4

DECIDABILITY

In Chapter 3 we introduced the Turing machine as a model of a general purpose computer and defined the notion of algorithm in terms of Turing machines by means of the Church–Turing thesis.

In this chapter we begin to investigate the power of algorithms to solve problems. We demonstrate certain problems that can be solved algorithmically and others that cannot. Our objective is to explore the limits of algorithmic solvability. You are probably familiar with solvability by algorithms because much of computer science is devoted to solving problems. The unsolvability of certain problems may come as a surprise.

Why should you study unsolvability? After all, showing that a problem is unsolvable doesn't appear to be of any use if you have to solve it. You need to study this phenomenon for two reasons. First, knowing when a problem is algorithmically unsolvable *is* useful because then you realize that the problem must be simplified or altered before you can find an algorithmic solution. Like any tool, computers have capabilities and limitations that must be appreciated if they are to be used well. The second reason is cultural. Even if you deal with problems that clearly are solvable, a glimpse of the unsolvable can stimulate your imagination and help you gain an important perspective on computation.

4.1 DECIDABLE LANGUAGES

In this section we give some examples of languages that are decidable by algorithms. We focus on languages concerning automata and grammars. For example, we present an algorithm that tests whether a string is a member of a context-free language (CFL). These languages are interesting for several reasons. First, certain problems of this kind are related to applications. This problem of testing whether a CFG generates a string is related to the problem of recognizing and compiling programs in a programming language. Second, certain other problems concerning automata and grammars are not decidable by algorithms. Starting with examples where decidability is possible helps you to appreciate the undecidable examples.

DECIDABLE PROBLEMS CONCERNING REGULAR LANGUAGES

We begin with certain computational problems concerning finite automata. We give algorithms for testing whether a finite automaton accepts a string, whether the language of a finite automaton is empty, and whether two finite automata are equivalent.

Note that we chose to represent various computational problems by languages. Doing so is convenient because we have already set up terminology for dealing with languages. For example, the ***acceptance problem*** for DFAs of testing whether a particular deterministic finite automaton accepts a given string can be expressed as a language, A_{DFA}. This language contains the encodings of all DFAs together with strings that the DFAs accept. Let

$$A_{\mathsf{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}.$$

The problem of testing whether a DFA B accepts an input w is the same as the problem of testing whether $\langle B, w\rangle$ is a member of the language A_{DFA}. Similarly, we can formulate other computational problems in terms of testing membership in a language. Showing that the language is decidable is the same as showing that the computational problem is decidable.

In the following theorem we show that A_{DFA} is decidable. Hence this theorem shows that the problem of testing whether a given finite automaton accepts a given string is decidable.

THEOREM 4.1

A_{DFA} is a decidable language.

PROOF IDEA We simply need to present a TM M that decides A_{DFA}.

M = "On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

PROOF We mention just a few implementation details of this proof. For those of you familiar with writing programs in any standard programming language, imagine how you would write a program to carry out the simulation.

First, let's examine the input $\langle B, w\rangle$. It is a representation of a DFA B together with a string w. One reasonable representation of B is simply a list of its five components: Q, Σ, δ, q_0, and F. When M receives its input, M first determines whether it properly represents a DFA B and a string w. If not, M rejects.

Then M carries out the simulation directly. It keeps track of B's current state and B's current position in the input w by writing this information down on its tape. Initially, B's current state is q_0 and B's current input position is the leftmost symbol of w. The states and position are updated according to the specified transition function δ. When M finishes processing the last symbol of w, M accepts the input if B is in an accepting state; M rejects the input if B is in a nonaccepting state.

We can prove a similar theorem for nondeterministic finite automata. Let

$$A_{\text{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}.$$

THEOREM 4.2

A_{NFA} is a decidable language.

PROOF We present a TM N that decides A_{NFA}. We could design N to operate like M, simulating an NFA instead of a DFA. Instead, we'll do it differently to illustrate a new idea: Have N use M as a subroutine. Because M is designed to work with DFAs, N first converts the NFA it receives as input to a DFA before passing it to M.

N = "On input $\langle B, w\rangle$, where B is an NFA and w is a string:

1. Convert NFA B to an equivalent DFA C, using the procedure for this conversion given in Theorem 1.39.
2. Run TM M from Theorem 4.1 on input $\langle C, w\rangle$.
3. If M accepts, *accept*; otherwise, *reject*."

Running TM M in stage 2 means incorporating M into the design of N as a subprocedure.

Similarly, we can determine whether a regular expression generates a given string. Let $A_{\mathsf{REX}} = \{\langle R, w\rangle | \ R \text{ is a regular expression that generates string } w\}$.

THEOREM 4.3

A_{REX} is a decidable language.

PROOF The following TM P decides A_{REX}.

P = "On input $\langle R, w\rangle$, where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent NFA A by using the procedure for this conversion given in Theorem 1.54.
2. Run TM N on input $\langle A, w\rangle$.
3. If N accepts, *accept*; if N rejects, *reject*."

Theorems 4.1, 4.2, and 4.3 illustrate that, for decidability purposes, it is equivalent to present the Turing machine with a DFA, an NFA, or a regular expression because the machine can convert one form of encoding to another.

Now we turn to a different kind of problem concerning finite automata: *emptiness testing* for the language of a finite automaton. In the preceding three theorems we had to determine whether a finite automaton accepts a particular string. In the next proof we must determine whether or not a finite automaton accepts any strings at all. Let

$$E_{\mathsf{DFA}} = \{\langle A\rangle | \ A \text{ is a DFA and } L(A) = \emptyset\}.$$

THEOREM 4.4

E_{DFA} is a decidable language.

PROOF A DFA accepts some string iff reaching an accept state from the start state by traveling along the arrows of the DFA is possible. To test this condition, we can design a TM T that uses a marking algorithm similar to that used in Example 3.23.

T = "On input $\langle A\rangle$, where A is a DFA:

1. Mark the start state of A.
2. Repeat until no new states get marked:
3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, *accept*; otherwise, *reject*."

The next theorem states that determining whether two DFAs recognize the same language is decidable. Let

$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}.$$

THEOREM 4.5

EQ_{DFA} is a decidable language.

PROOF To prove this theorem, we use Theorem 4.4. We construct a new DFA C from A and B, where C accepts only those strings that are accepted by either A or B but not by both. Thus, if A and B recognize the same language, C will accept nothing. The language of C is

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right).$$

This expression is sometimes called the ***symmetric difference*** of $L(A)$ and $L(B)$ and is illustrated in the following figure. Here, $\overline{L(A)}$ is the complement of $L(A)$. The symmetric difference is useful here because $L(C) = \emptyset$ iff $L(A) = L(B)$. We can construct C from A and B with the constructions for proving the class of regular languages closed under complementation, union, and intersection. These constructions are algorithms that can be carried out by Turing machines. Once we have constructed C, we can use Theorem 4.4 to test whether $L(C)$ is empty. If it is empty, $L(A)$ and $L(B)$ must be equal.

F = "On input $\langle A, B\rangle$, where A and B are DFAs:

1. Construct DFA C as described.
2. Run TM T from Theorem 4.4 on input $\langle C\rangle$.
3. If T accepts, *accept*. If T rejects, *reject*."

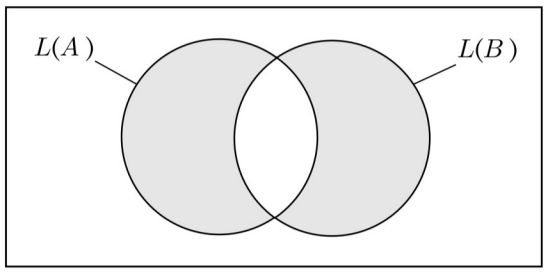


FIGURE 4.6
The symmetric difference of $L(A)$ and $L(B)$

DECIDABLE PROBLEMS CONCERNING CONTEXT-FREE LANGUAGES

Here, we describe algorithms to determine whether a CFG generates a particular string and to determine whether the language of a CFG is empty. Let

$$A_{\mathsf{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}.$$

THEOREM 4.7

A_{CFG} is a decidable language.

PROOF IDEA For CFG G and string w, we want to determine whether G generates w. One idea is to use G to go through all derivations to determine whether any is a derivation of w. This idea doesn't work, as infinitely many derivations may have to be tried. If G does not generate w, this algorithm would never halt. This idea gives a Turing machine that is a recognizer, but not a decider, for A_{CFG}.

To make this Turing machine into a decider, we need to ensure that the algorithm tries only finitely many derivations. In Problem 2.26 (page 157) we showed that if G were in Chomsky normal form, any derivation of w has $2n - 1$ steps, where n is the length of w. In that case, checking only derivations with $2n - 1$ steps to determine whether G generates w would be sufficient. Only finitely many such derivations exist. We can convert G to Chomsky normal form by using the procedure given in Section 2.1.

PROOF The TM S for A_{CFG} follows.

S = "On input $\langle G, w\rangle$, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n-1$ steps, where n is the length of w; except if $n = 0$, then instead list all derivations with one step.
3. If any of these derivations generate w, *accept*; if not, *reject*."

The problem of determining whether a CFG generates a particular string is related to the problem of compiling programming languages. The algorithm in TM S is very inefficient and would never be used in practice, but it is easy to describe and we aren't concerned with efficiency here. In Part Three of this book, we address issues concerning the running time and memory use of algorithms. In the proof of Theorem 7.16, we describe a more efficient algorithm for recognizing general context-free languages. Even greater efficiency is possible for recognizing deterministic context-free languages.

Recall that we have given procedures for converting back and forth between CFGs and PDAs in Theorem 2.20. Hence everything we say about the decidability of problems concerning CFGs applies equally well to PDAs.

Let's turn now to the emptiness testing problem for the language of a CFG. As we did for DFAs, we can show that the problem of determining whether a CFG generates any strings at all is decidable. Let

$$E_{\mathsf{CFG}} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}.$$

THEOREM 4.8

E_{CFG} is a decidable language.

PROOF IDEA To find an algorithm for this problem, we might attempt to use TM S from Theorem 4.7. It states that we can test whether a CFG generates some particular string w. To determine whether $L(G) = \emptyset$, the algorithm might try going through all possible w's, one by one. But there are infinitely many w's to try, so this method could end up running forever. We need to take a different approach.

In order to determine whether the language of a grammar is empty, we need to test whether the start variable can generate a string of terminals. The algorithm does so by solving a more general problem. It determines *for each variable* whether that variable is capable of generating a string of terminals. When the algorithm has determined that a variable can generate some string of terminals, the algorithm keeps track of this information by placing a mark on that variable.

First, the algorithm marks all the terminal symbols in the grammar. Then, it scans all the rules of the grammar. If it ever finds a rule that permits some variable to be replaced by some string of symbols, all of which are already marked, the algorithm knows that this variable can be marked, too. The algorithm continues in this way until it cannot mark any additional variables. The TM R implements this algorithm.

PROOF

$R =$ "On input $\langle G \rangle$, where G is a CFG:

1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked:
3. Mark any variable A where G has a rule $A \to U_1 U_2 \cdots U_k$ and each symbol $U_1, \ldots, U_k$ has already been marked.
4. If the start variable is not marked, *accept*; otherwise, *reject*."

Next, we consider the problem of determining whether two context-free grammars generate the same language. Let

$$EQ_{\mathsf{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}.$$

Theorem 4.5 gave an algorithm that decides the analogous language EQ_{DFA} for finite automata. We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable. Because E_{CFG} also is decidable, you might think that we can use a similar strategy to prove that EQ_{CFG} is decidable. But something is wrong with this idea! The class of context-free languages is *not* closed under complementation or intersection, as you proved in Exercise 2.2. In fact, EQ_{CFG} is not decidable. The technique for proving so is presented in Chapter 5.

Now we show that context-free languages are decidable by Turing machines.

THEOREM 4.9

Every context-free language is decidable.

PROOF IDEA Let A be a CFL. Our objective is to show that A is decidable. One (bad) idea is to convert a PDA for A directly into a TM. That isn't hard to do because simulating a stack with the TM's more versatile tape is easy. The PDA for A may be nondeterministic, but that seems okay because we can convert it into a nondeterministic TM and we know that any nondeterministic TM can be converted into an equivalent deterministic TM. Yet, there is a difficulty. Some branches of the PDA's computation may go on forever, reading and writing the stack without ever halting. The simulating TM then would also have some nonhalting branches in its computation, and so the TM would not be a decider. A different idea is necessary. Instead, we prove this theorem with the TM S that we designed in Theorem 4.7 to decide A_{CFG}.

PROOF Let G be a CFG for A and design a TM M_G that decides A. We build a copy of G into M_G. It works as follows.

M_G = "On input w:

1. Run TM S on input $\langle G, w\rangle$.
2. If this machine accepts, *accept*; if it rejects, *reject*."

Theorem 4.9 provides the final link in the relationship among the four main classes of languages that we have described so far: regular, context-free, decidable, and Turing-recognizable. Figure 4.10 depicts this relationship.

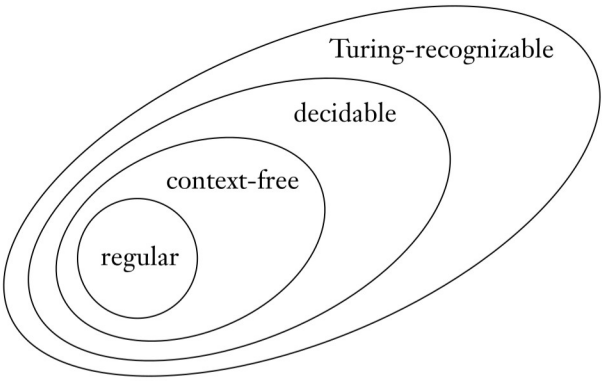


FIGURE **4.10**
The relationship among classes of languages

4.2 UNDECIDABILITY

In this section, we prove one of the most philosophically important theorems of the theory of computation: There is a specific problem that is algorithmically unsolvable. Computers appear to be so powerful that you may believe that all problems will eventually yield to them. The theorem presented here demonstrates that computers are limited in a fundamental way.

What sorts of problems are unsolvable by computer? Are they esoteric, dwelling only in the minds of theoreticians? No! Even some ordinary problems that people want to solve turn out to be computationally unsolvable.

In one type of unsolvable problem, you are given a computer program and a precise specification of what that program is supposed to do (e.g., sort a list of numbers). You need to verify that the program performs as specified (i.e., that it is correct). Because both the program and the specification are mathematically precise objects, you hope to automate the process of verification by feeding these objects into a suitably programmed computer. However, you will be disappointed. The general problem of software verification is not solvable by computer.

In this section and in Chapter 5, you will encounter several computationally unsolvable problems. We aim to help you develop a feeling for the types of problems that are unsolvable and to learn techniques for proving unsolvability.

Now we turn to our first theorem that establishes the undecidability of a specific language: the problem of determining whether a Turing machine accepts a

given input string. We call it A_{TM} by analogy with A_{DFA} and A_{CFG}. But, whereas A_{DFA} and A_{CFG} were decidable, A_{TM} is not. Let

$$A_{\mathsf{TM}} = \{\langle M, w\rangle | \ M \text{ is a } \mathsf{TM} \text{ and } M \text{ accepts } w\}.$$

THEOREM 4.11

A_{TM} is undecidable.

Before we get to the proof, let's first observe that A_{TM} is Turing-recognizable. Thus, this theorem shows that recognizers *are* more powerful than deciders. Requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize. The following Turing machine U recognizes A_{TM}.

U = "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

Note that this machine loops on input $\langle M, w\rangle$ if M loops on w, which is why this machine does not decide A_{TM}. If the algorithm had some way to determine that M was not halting on w, it could *reject* in this case. However, an algorithm has no way to make this determination, as we shall see.

The Turing machine U is interesting in its own right. It is an example of the *universal Turing machine* first proposed by Alan Turing in 1936. This machine is called universal because it is capable of simulating any other Turing machine from the description of that machine. The universal Turing machine played an important early role in the development of stored-program computers.

THE DIAGONALIZATION METHOD

The proof of the undecidability of A_{TM} uses a technique called *diagonalization*, discovered by mathematician Georg Cantor in 1873. Cantor was concerned with the problem of measuring the sizes of infinite sets. If we have two infinite sets, how can we tell whether one is larger than the other or whether they are of the same size? For finite sets, of course, answering these questions is easy. We simply count the elements in a finite set, and the resulting number is its size. But if we try to count the elements of an infinite set, we will never finish! So we can't use the counting method to determine the relative sizes of infinite sets.

For example, take the set of even integers and the set of all strings over {0,1}. Both sets are infinite and thus larger than any finite set, but is one of the two larger than the other? How can we compare their relative size?

Cantor proposed a rather nice solution to this problem. He observed that two finite sets have the same size if the elements of one set can be paired with the elements of the other set. This method compares the sizes without resorting to counting. We can extend this idea to infinite sets. Here it is more precisely.

DEFINITION 4.12

Assume that we have sets A and B and a function f from A to B. Say that f is ***one-to-one*** if it never maps two different elements to the same place—that is, if $f(a) \neq f(b)$ whenever $a \neq b$. Say that f is ***onto*** if it hits every element of B—that is, if for every $b \in B$ there is an $a \in A$ such that $f(a) = b$. Say that A and B are the ***same size*** if there is a one-to-one, onto function $f\colon A \longrightarrow B$. A function that is both one-to-one and onto is called a ***correspondence***. In a correspondence, every element of A maps to a unique element of B and each element of B has a unique element of A mapping to it. A correspondence is simply a way of pairing the elements of A with the elements of B.

Alternative common terminology for these types of functions is ***injective*** for one-to-one, ***surjective*** for onto, and ***bijective*** for one-to-one and onto.

EXAMPLE 4.13

Let $\mathcal{N}$ be the set of natural numbers $\{1, 2, 3, \ldots\}$ and let $\mathcal{E}$ be the set of even natural numbers $\{2, 4, 6, \ldots\}$. Using Cantor's definition of size, we can see that $\mathcal{N}$ and $\mathcal{E}$ have the same size. The correspondence f mapping $\mathcal{N}$ to $\mathcal{E}$ is simply $f(n) = 2n$. We can visualize f more easily with the help of a table.

n	$f(n)$
1	2
2	4
3	6
$\vdots$	$\vdots$

Of course, this example seems bizarre. Intuitively, $\mathcal{E}$ seems smaller than $\mathcal{N}$ because $\mathcal{E}$ is a proper subset of $\mathcal{N}$. But pairing each member of $\mathcal{N}$ with its own member of $\mathcal{E}$ is possible, so we declare these two sets to be the same size. ■

DEFINITION 4.14

A set A is ***countable*** if either it is finite or it has the same size as $\mathcal{N}$.

EXAMPLE 4.15

Now we turn to an even stranger example. If we let $\mathcal{Q} = \{\frac{m}{n} \mid m, n \in \mathcal{N}\}$ be the set of positive rational numbers, $\mathcal{Q}$ seems to be much larger than $\mathcal{N}$. Yet these two sets are the same size according to our definition. We give a correspondence with $\mathcal{N}$ to show that $\mathcal{Q}$ is countable. One easy way to do so is to list all the

elements of $\mathcal{Q}$. Then we pair the first element on the list with the number 1 from $\mathcal{N}$, the second element on the list with the number 2 from $\mathcal{N}$, and so on. We must ensure that every member of $\mathcal{Q}$ appears only once on the list.

To get this list, we make an infinite matrix containing all the positive rational numbers, as shown in Figure 4.16. The ith row contains all numbers with numerator i and the jth column has all numbers with denominator j. So the number $\frac{i}{j}$ occurs in the ith row and jth column.

Now we turn this matrix into a list. One (bad) way to attempt it would be to begin the list with all the elements in the first row. That isn't a good approach because the first row is infinite, so the list would never get to the second row. Instead we list the elements on the diagonals, which are superimposed on the diagram, starting from the corner. The first diagonal contains the single element $\frac{1}{1}$, and the second diagonal contains the two elements $\frac{2}{1}$ and $\frac{1}{2}$. So the first three elements on the list are $\frac{1}{1}$, $\frac{2}{1}$, and $\frac{1}{2}$. In the third diagonal, a complication arises. It contains $\frac{3}{1}$, $\frac{2}{2}$, and $\frac{1}{3}$. If we simply added these to the list, we would repeat $\frac{1}{1} = \frac{2}{2}$. We avoid doing so by skipping an element when it would cause a repetition. So we add only the two new elements $\frac{3}{1}$ and $\frac{1}{3}$. Continuing in this way, we obtain a list of all the elements of $\mathcal{Q}$.

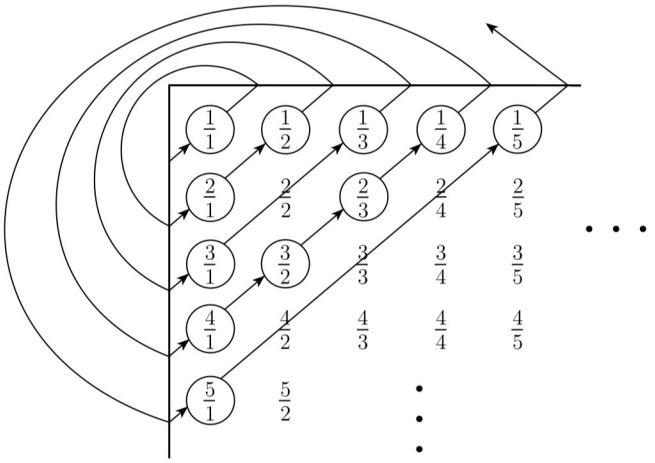


FIGURE 4.16
A correspondence of $\mathcal{N}$ and $\mathcal{Q}$

After seeing the correspondence of $\mathcal{N}$ and $\mathcal{Q}$, you might think that any two infinite sets can be shown to have the same size. After all, you need only demonstrate a correspondence, and this example shows that surprising correspondences do exist. However, for some infinite sets, no correspondence with $\mathcal{N}$ exists. These sets are simply too big. Such sets are called ***uncountable***.

The set of real numbers is an example of an uncountable set. A ***real number*** is one that has a decimal representation. The numbers $\pi = 3.1415926\ldots$ and

$\sqrt{2} = 1.4142135\ldots$ are examples of real numbers. Let $\mathcal{R}$ be the set of real numbers. Cantor proved that $\mathcal{R}$ is uncountable. In doing so, he introduced the diagonalization method.

THEOREM 4.17

$\mathcal{R}$ is uncountable.

PROOF In order to show that $\mathcal{R}$ is uncountable, we show that no correspondence exists between $\mathcal{N}$ and $\mathcal{R}$. The proof is by contradiction. Suppose that a correspondence f existed between $\mathcal{N}$ and $\mathcal{R}$. Our job is to show that f fails to work as it should. For it to be a correspondence, f must pair all the members of $\mathcal{N}$ with all the members of $\mathcal{R}$. But we will find an x in $\mathcal{R}$ that is not paired with anything in $\mathcal{N}$, which will be our contradiction.

The way we find this x is by actually constructing it. We choose each digit of x to make x different from one of the real numbers that is paired with an element of $\mathcal{N}$. In the end, we are sure that x is different from any real number that is paired.

We can illustrate this idea by giving an example. Suppose that the correspondence f exists. Let $f(1) = 3.14159\ldots$, $f(2) = 55.55555\ldots$, $f(3) = \ldots$, and so on, just to make up some values for f. Then f pairs the number 1 with $3.14159\ldots$, the number 2 with $55.55555\ldots$, and so on. The following table shows a few values of a hypothetical correspondence f between $\mathcal{N}$ and $\mathcal{R}$.

n	$f(n)$
1	3.14159...
2	55.55555...
3	0.12345...
4	0.50000...
⋮	⋮

We construct the desired x by giving its decimal representation. It is a number between 0 and 1, so all its significant digits are fractional digits following the decimal point. Our objective is to ensure that $x \neq f(n)$ for any n. To ensure that $x \neq f(1)$, we let the first digit of x be anything different from the first fractional digit 1 of $f(1) = 3.14159\ldots$. Arbitrarily, we let it be 4. To ensure that $x \neq f(2)$, we let the second digit of x be anything different from the second fractional digit 5 of $f(2) = 55.55555\ldots$. Arbitrarily, we let it be 6. The third fractional digit of $f(3) = 0.12345\ldots$ is 3, so we let x be anything different—say, 4. Continuing in this way down the diagonal of the table for f, we obtain all the digits of x, as shown in the following table. We know that x is not $f(n)$ for any n because it differs from $f(n)$ in the nth fractional digit. (A slight problem arises because certain numbers, such as $0.1999\ldots$ and $0.2000\ldots$, are equal even though their decimal representations are different. We avoid this problem by never selecting the digits 0 or 9 when we construct x.)

n	$f(n)$
1	3.14159...
2	55.55555...
3	0.12345...
4	0.50000...
⋮	⋮

$x = 0.4641\ldots$

The preceding theorem has an important application to the theory of computation. It shows that some languages are not decidable or even Turing-recognizable, for the reason that there are uncountably many languages yet only countably many Turing machines. Because each Turing machine can recognize a single language and there are more languages than Turing machines, some languages are not recognized by any Turing machine. Such languages are not Turing-recognizable, as we state in the following corollary.

COROLLARY 4.18

Some languages are not Turing-recognizable.

PROOF To show that the set of all Turing machines is countable, we first observe that the set of all strings Σ^* is countable for any alphabet Σ. With only finitely many strings of each length, we may form a list of Σ^* by writing down all strings of length 0, length 1, length 2, and so on.

The set of all Turing machines is countable because each Turing machine M has an encoding into a string $\langle M \rangle$. If we simply omit those strings that are not legal encodings of Turing machines, we can obtain a list of all Turing machines.

To show that the set of all languages is uncountable, we first observe that the set of all infinite binary sequences is uncountable. An *infinite binary sequence* is an unending sequence of 0s and 1s. Let $\mathcal{B}$ be the set of all infinite binary sequences. We can show that $\mathcal{B}$ is uncountable by using a proof by diagonalization similar to the one we used in Theorem 4.17 to show that $\mathcal{R}$ is uncountable.

Let $\mathcal{L}$ be the set of all languages over alphabet Σ. We show that $\mathcal{L}$ is uncountable by giving a correspondence with $\mathcal{B}$, thus showing that the two sets are the same size. Let $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$. Each language $A \in \mathcal{L}$ has a unique sequence in $\mathcal{B}$. The ith bit of that sequence is a 1 if $s_i \in A$ and is a 0 if $s_i \notin A$, which is called the ***characteristic sequence*** of A. For example, if A were the language of all strings starting with a 0 over the alphabet {0,1}, its characteristic sequence χ_A would be

$$\begin{array}{rccccccccccc} \Sigma^* = \{ & \varepsilon, & 0, & 1, & 00, & 01, & 10, & 11, & 000, & 001, & \cdots & \}\ ; \\ A = \{ & & 0, & & 00, & 01, & & & 000, & 001, & \cdots & \}\ ; \\ \chi_A = & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \cdots & . \end{array}$$

The function $f: \mathcal{L} \longrightarrow \mathcal{B}$, where $f(A)$ equals the characteristic sequence of A, is one-to-one and onto, and hence is a correspondence. Therefore, as $\mathcal{B}$ is uncountable, $\mathcal{L}$ is uncountable as well.

Thus we have shown that the set of all languages cannot be put into a correspondence with the set of all Turing machines. We conclude that some languages are not recognized by any Turing machine.

AN UNDECIDABLE LANGUAGE

Now we are ready to prove Theorem 4.11, the undecidability of the language

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a } \mathsf{TM} \text{ and } M \text{ accepts } w\}.$$

PROOF We assume that A_{TM} is decidable and obtain a contradiction. Suppose that H is a decider for A_{TM}. On input $\langle M, w\rangle$, where M is a TM and w is a string, H halts and accepts if M accepts w. Furthermore, H halts and rejects if M fails to accept w. In other words, we assume that H is a TM, where

$$H(\langle M, w\rangle) = \begin{cases} \mathit{accept} & \text{if } M \text{ accepts } w \\ \mathit{reject} & \text{if } M \text{ does not accept } w. \end{cases}$$

Now we construct a new Turing machine D with H as a subroutine. This new TM calls H to determine what M does when the input to M is its own description $\langle M\rangle$. Once D has determined this information, it does the opposite. That is, it rejects if M accepts and accepts if M does not accept. The following is a description of D.

D = "On input $\langle M\rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M\rangle\rangle$.
2. Output the opposite of what H outputs. That is, if H accepts, *reject*; and if H rejects, *accept*."

Don't be confused by the notion of running a machine on its own description! That is similar to running a program with itself as input, something that does occasionally occur in practice. For example, a compiler is a program that translates other programs. A compiler for the language Python may itself be written in Python, so running that program on itself would make sense. In summary,

$$D(\langle M\rangle) = \begin{cases} \mathit{accept} & \text{if } M \text{ does not accept } \langle M\rangle \\ \mathit{reject} & \text{if } M \text{ accepts } \langle M\rangle. \end{cases}$$

What happens when we run D with its own description $\langle D\rangle$ as input? In that case, we get

$$D(\langle D\rangle) = \begin{cases} \mathit{accept} & \text{if } D \text{ does not accept } \langle D\rangle \\ \mathit{reject} & \text{if } D \text{ accepts } \langle D\rangle. \end{cases}$$

No matter what D does, it is forced to do the opposite, which is obviously a contradiction. Thus, neither TM D nor TM H can exist.

Let's review the steps of this proof. Assume that a TM H decides A_{TM}. Use H to build a TM D that takes an input $\langle M \rangle$, where D accepts its input $\langle M \rangle$ exactly when M does not accept its input $\langle M \rangle$. Finally, run D on itself. Thus, the machines take the following actions, with the last line being the contradiction.

- H accepts $\langle M, w \rangle$ exactly when M accepts w.
- D rejects $\langle M \rangle$ exactly when M accepts $\langle M \rangle$.
- D rejects $\langle D \rangle$ exactly when D accepts $\langle D \rangle$.

Where is the diagonalization in the proof of Theorem 4.11? It becomes apparent when you examine tables of behavior for TMs H and D. In these tables we list all TMs down the rows, M_1, M_2, ..., and all their descriptions across the columns, $\langle M_1 \rangle$, $\langle M_2 \rangle$, ... The entries tell whether the machine in a given row accepts the input in a given column. The entry is *accept* if the machine accepts the input but is blank if it rejects or loops on that input. We made up the entries in the following figure to illustrate the idea.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*		*accept*		
M_2	*accept*	*accept*	*accept*	*accept*	
M_3					$\cdots$
M_4	*accept*	*accept*			
$\vdots$			$\vdots$		

FIGURE 4.19
Entry i, j is *accept* if M_i accepts $\langle M_j \rangle$

In the following figure, the entries are the results of running H on inputs corresponding to Figure 4.19. So if M_3 does not accept input $\langle M_2 \rangle$, the entry for row M_3 and column $\langle M_2 \rangle$ is *reject* because H rejects input $\langle M_3, \langle M_2 \rangle \rangle$.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*	
M_2	*accept*	*accept*	*accept*	*accept*	
M_3	*reject*	*reject*	*reject*	*reject*	$\cdots$
M_4	*accept*	*accept*	*reject*	*reject*	
$\vdots$			$\vdots$		

FIGURE 4.20
Entry i, j is the value of H on input $\langle M_i, \langle M_j \rangle \rangle$

In the following figure, we added D to Figure 4.20. By our assumption, H is a TM and so is D. Therefore, it must occur on the list $M_1, M_2, \ldots$ of all TMs. Note that D computes the opposite of the diagonal entries. The contradiction occurs at the point of the question mark where the entry must be the opposite of itself.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	*reject*		*accept*	
$\vdots$			$\vdots$		$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		?	
$\vdots$			$\vdots$				$\ddots$

FIGURE 4.21
If D is in the figure, a contradiction occurs at "?"

A TURING-UNRECOGNIZABLE LANGUAGE

In the preceding section, we exhibited a language—namely, A_{TM}—that is undecidable. Now we exhibit a language that isn't even Turing-recognizable. Note that A_{TM} will not suffice for this purpose because we showed that A_{TM} is Turing-recognizable (page 202). The following theorem shows that if both a language and its complement are Turing-recognizable, the language is decidable. Hence for any undecidable language, either it or its complement is not Turing-recognizable. Recall that the complement of a language is the language consisting of all strings that are not in the language. We say that a language is ***co-Turing-recognizable*** if it is the complement of a Turing-recognizable language.

THEOREM 4.22

A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.

In other words, a language is decidable exactly when both it and its complement are Turing-recognizable.

PROOF We have two directions to prove. First, if A is decidable, we can easily see that both A and its complement $\overline{A}$ are Turing-recognizable. Any decidable language is Turing-recognizable, and the complement of a decidable language also is decidable.

For the other direction, if both A and $\overline{A}$ are Turing-recognizable, we let M_1 be the recognizer for A and M_2 be the recognizer for $\overline{A}$. The following Turing machine M is a decider for A.

$M =$ "On input w:

1. Run both M_1 and M_2 on input w in parallel.
2. If M_1 accepts, *accept*; if M_2 accepts, *reject*."

Running the two machines in parallel means that M has two tapes, one for simulating M_1 and the other for simulating M_2. In this case, M takes turns simulating one step of each machine, which continues until one of them accepts.

Now we show that M decides A. Every string w is either in A or $\overline{A}$. Therefore, either M_1 or M_2 must accept w. Because M halts whenever M_1 or M_2 accepts, M always halts and so it is a decider. Furthermore, it accepts all strings in A and rejects all strings not in A. So M is a decider for A, and thus A is decidable.

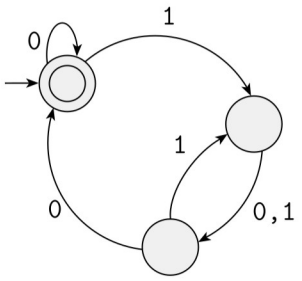
COROLLARY 4.23

$\overline{A_{\mathsf{TM}}}$ is not Turing-recognizable.

PROOF We know that A_{TM} is Turing-recognizable. If $\overline{A_{\mathsf{TM}}}$ also were Turing-recognizable, A_{TM} would be decidable. Theorem 4.11 tells us that A_{TM} is not decidable, so $\overline{A_{\mathsf{TM}}}$ must not be Turing-recognizable.

EXERCISES

[A]**4.1** Answer all parts for the following DFA M and give reasons for your answers.



a. Is $\langle M, 0100\rangle \in A_{\mathsf{DFA}}$?
b. Is $\langle M, 011\rangle \in A_{\mathsf{DFA}}$?
c. Is $\langle M\rangle \in A_{\mathsf{DFA}}$?
d. Is $\langle M, 0100\rangle \in A_{\mathsf{REX}}$?
e. Is $\langle M\rangle \in E_{\mathsf{DFA}}$?
f. Is $\langle M, M\rangle \in EQ_{\mathsf{DFA}}$?

4.2 Consider the problem of determining whether a DFA and a regular expression are equivalent. Express this problem as a language and show that it is decidable.

4.3 Let $ALL_{\mathsf{DFA}} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \Sigma^*\}$. Show that ALL_{DFA} is decidable.

4.4 Let $A\varepsilon_{\mathsf{CFG}} = \{\langle G \rangle \mid G \text{ is a CFG that generates } \varepsilon\}$. Show that $A\varepsilon_{\mathsf{CFG}}$ is decidable.

[A]**4.5** Let $E_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$. Show that $\overline{E_{\mathsf{TM}}}$, the complement of E_{TM}, is Turing-recognizable.

4.6 Let X be the set $\{1, 2, 3, 4, 5\}$ and Y be the set $\{6, 7, 8, 9, 10\}$. We describe the functions $f\colon X \longrightarrow Y$ and $g\colon X \longrightarrow Y$ in the following tables. Answer each part and give a reason for each negative answer.

n	$f(n)$
1	6
2	7
3	6
4	7
5	6

n	$g(n)$
1	10
2	9
3	8
4	7
5	6

[A]**a.** Is f one-to-one?
b. Is f onto?
c. Is f a correspondence?
[A]**d.** Is g one-to-one?
e. Is g onto?
f. Is g a correspondence?

4.7 Let $\mathcal{B}$ be the set of all infinite sequences over $\{0,1\}$. Show that $\mathcal{B}$ is uncountable using a proof by diagonalization.

4.8 Let $T = \{(i, j, k) \mid i, j, k \in \mathcal{N}\}$. Show that T is countable.

4.9 Review the way that we define sets to be the same size in Definition 4.12 (page 203). Show that "is the same size" is an equivalence relation.

PROBLEMS

[A]**4.10** Let $INFINITE_{\mathsf{DFA}} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) \text{ is an infinite language}\}$. Show that $INFINITE_{\mathsf{DFA}}$ is decidable.

4.11 Let $INFINITE_{\mathsf{PDA}} = \{\langle M \rangle \mid M \text{ is a PDA and } L(M) \text{ is an infinite language}\}$. Show that $INFINITE_{\mathsf{PDA}}$ is decidable.

[A]**4.12** Let $A = \{\langle M \rangle \mid M \text{ is a DFA that doesn't accept any string containing an odd number of 1s}\}$. Show that A is decidable.

4.13 Let $A = \{\langle R, S \rangle \mid R \text{ and } S \text{ are regular expressions and } L(R) \subseteq L(S)\}$. Show that A is decidable.

[A]**4.14** Let $\Sigma = \{0,1\}$. Show that the problem of determining whether a CFG generates some string in 1^* is decidable. In other words, show that

$$\{\langle G \rangle \mid G \text{ is a CFG over } \{0,1\} \text{ and } 1^* \cap L(G) \neq \emptyset\}$$

is a decidable language.

*4.15 Show that the problem of determining whether a CFG generates all strings in 1^* is decidable. In other words, show that $\{\langle G\rangle \mid G \text{ is a CFG over } \{0,1\} \text{ and } 1^* \subseteq L(G)\}$ is a decidable language.

4.16 Let $A = \{\langle R\rangle \mid R$ is a regular expression describing a language containing at least one string w that has 111 as a substring (i.e., $w = x111y$ for some x and $y)\}$. Show that A is decidable.

4.17 Prove that EQ_{DFA} is decidable by testing the two DFAs on all strings up to a certain size. Calculate a size that works.

*4.18 Let C be a language. Prove that C is Turing-recognizable iff a decidable language D exists such that $C = \{x \mid \exists y\ (\langle x, y\rangle \in D)\}$.

*4.19 Prove that the class of decidable languages is not closed under homomorphism.

4.20 Let A and B be two disjoint languages. Say that language C ***separates*** A and B if $A \subseteq C$ and $B \subseteq \overline{C}$. Show that any two disjoint co-Turing-recognizable languages are separable by some decidable language.

4.21 Let $S = \{\langle M\rangle \mid M$ is a DFA that accepts $w^{\mathcal{R}}$ whenever it accepts $w\}$. Show that S is decidable.

4.22 Let $\mathit{PREFIX\text{-}FREE}_{\mathsf{REX}} = \{\langle R\rangle \mid R$ is a regular expression and $L(R)$ is prefix-free$\}$. Show that $\mathit{PREFIX\text{-}FREE}_{\mathsf{REX}}$ is decidable. Why does a similar approach fail to show that $\mathit{PREFIX\text{-}FREE}_{\mathsf{CFG}}$ is decidable?

A*4.23 Say that an NFA is ***ambiguous*** if it accepts some string along two different computation branches. Let $\mathit{AMBIG}_{\mathsf{NFA}} = \{\langle N\rangle \mid N$ is an ambiguous NFA$\}$. Show that $\mathit{AMBIG}_{\mathsf{NFA}}$ is decidable. (Suggestion: One elegant way to solve this problem is to construct a suitable DFA and then run E_{DFA} on it.)

4.24 A ***useless state*** in a pushdown automaton is never entered on any input string. Consider the problem of determining whether a pushdown automaton has any useless states. Formulate this problem as a language and show that it is decidable.

A*4.25 Let $\mathit{BAL}_{\mathsf{DFA}} = \{\langle M\rangle \mid M$ is a DFA that accepts some string containing an equal number of 0s and 1s$\}$. Show that $\mathit{BAL}_{\mathsf{DFA}}$ is decidable. (Hint: Theorems about CFLs are helpful here.)

*4.26 Let $\mathit{PAL}_{\mathsf{DFA}} = \{\langle M\rangle \mid M$ is a DFA that accepts some palindrome$\}$. Show that $\mathit{PAL}_{\mathsf{DFA}}$ is decidable. (Hint: Theorems about CFLs are helpful here.)

*4.27 Let $E = \{\langle M\rangle \mid M$ is a DFA that accepts some string with more 1s than 0s$\}$. Show that E is decidable. (Hint: Theorems about CFLs are helpful here.)

4.28 Let $C = \{\langle G, x\rangle \mid G$ is a CFG x is a substring of some $y \in L(G)\}$. Show that C is decidable. (Hint: An elegant solution to this problem uses the decider for E_{CFG}.)

4.29 Let $C_{\mathsf{CFG}} = \{\langle G, k\rangle \mid G$ is a CFG and $L(G)$ contains exactly k strings where $k \geq 0$ or $k = \infty\}$. Show that C_{CFG} is decidable.

4.30 Let A be a Turing-recognizable language consisting of descriptions of Turing machines, $\{\langle M_1\rangle, \langle M_2\rangle, \ldots\}$, where every M_i is a decider. Prove that some decidable language D is not decided by any decider M_i whose description appears in A. (Hint: You may find it helpful to consider an enumerator for A.)

4.31 Say that a variable A in CFL G is ***usable*** if it appears in some derivation of some string $w \in G$. Given a CFG G and a variable A, consider the problem of testing whether A is usable. Formulate this problem as a language and show that it is decidable.

4.32 The proof of Lemma 2.41 says that (q, x) is a *looping situation* for a DPDA P if when P is started in state q with $x \in \Gamma$ on the top of the stack, it never pops anything below x and it never reads an input symbol. Show that F is decidable, where $F = \{\langle P, q, x\rangle \mid (q, x) \text{ is a looping situation for } P\}$.

SELECTED SOLUTIONS

4.1 **(a)** Yes. The DFA M accepts 0100.

(b) No. M doesn't accept 011.

(c) No. This input has only a single component and thus is not of the correct form.

(d) No. The first component is not a regular expression and so the input is not of the correct form.

(e) No. M's language isn't empty.

(f) Yes. M accepts the same language as itself.

4.5 Let $s_1, s_2, \ldots$ be a list of all strings in Σ^*. The following TM recognizes $\overline{E_{\text{TM}}}$.

"On input $\langle M\rangle$, where M is a TM:

1. Repeat the following for $i = 1, 2, 3, \ldots$.
2. Run M for i steps on each input, $s_1, s_2, \ldots, s_i$.
3. If M has accepted any of these, *accept*. Otherwise, continue."

4.6 **(a)** No, f is not one-to-one because $f(1) = f(3)$.

(d) Yes, g is one-to-one.

4.10 The following TM I decides $INFINITE_{\text{DFA}}$.

I = "On input $\langle A\rangle$, where A is a DFA:

1. Let k be the number of states of A.
2. Construct a DFA D that accepts all strings of length k or more.
3. Construct a DFA M such that $L(M) = L(A) \cap L(D)$.
4. Test $L(M) = \emptyset$ using the E_{DFA} decider T from Theorem 4.4.
5. If T accepts, *reject*; if T rejects, *accept*."

This algorithm works because a DFA that accepts infinitely many strings must accept arbitrarily long strings. Therefore, this algorithm accepts such DFAs. Conversely, if the algorithm accepts a DFA, the DFA accepts some string of length k or more, where k is the number of states of the DFA. This string may be pumped in the manner of the pumping lemma for regular languages to obtain infinitely many accepted strings.

4.12 The following TM decides A.

"On input $\langle M \rangle$:

1. Construct a DFA O that accepts every string containing an odd number of 1s.
2. Construct a DFA B such that $L(B) = L(M) \cap L(O)$.
3. Test whether $L(B) = \emptyset$ using the E_{DFA} decider T from Theorem 4.4.
4. If T accepts, *accept*; if T rejects, *reject*."

4.14 You showed in Problem 2.18 that if C is a context-free language and R is a regular language, then $C \cap R$ is context free. Therefore, $\mathtt{1}^* \cap L(G)$ is context free. The following TM decides the language of this problem.

"On input $\langle G \rangle$:

1. Construct CFG H such that $L(H) = \mathtt{1}^* \cap L(G)$.
2. Test whether $L(H) = \emptyset$ using the E_{CFG} decider R from Theorem 4.8.
3. If R accepts, *reject*; if R rejects, *accept*."

4.23 The following procedure decides $AMBIG_{\text{NFA}}$. Given an NFA N, we design a DFA D that simulates N and accepts a string iff it is accepted by N along two different computational branches. Then we use a decider for E_{DFA} to determine whether D accepts any strings.

Our strategy for constructing D is similar to the NFA-to-DFA conversion in the proof of Theorem 1.39. We simulate N by keeping a pebble on each active state. We begin by putting a red pebble on the start state and on each state reachable from the start state along ε transitions. We move, add, and remove pebbles in accordance with N's transitions, preserving the color of the pebbles. Whenever two or more pebbles are moved to the same state, we replace its pebbles with a blue pebble. After reading the input, we accept if a blue pebble is on an accept state of N or if two different accept states of N have red pebbles on them.

The DFA D has a state corresponding to each possible position of pebbles. For each state of N, three possibilities occur: It can contain a red pebble, a blue pebble, or no pebble. Thus, if N has n states, D will have 3^n states. Its start state, accept states, and transition function are defined to carry out the simulation.

4.25 The language of all strings with an equal number of 0s and 1s is a context-free language, generated by the grammar $S \to \mathtt{1}S\mathtt{0}S \mid \mathtt{0}S\mathtt{1}S \mid \varepsilon$. Let P be the PDA that recognizes this language. Build a TM M for BAL_{DFA}, which operates as follows. On input $\langle B \rangle$, where B is a DFA, use B and P to construct a new PDA R that recognizes the intersection of the languages of B and P. Then test whether R's language is empty. If its language is empty, *reject*; otherwise, *accept*.

5

REDUCIBILITY

In Chapter 4 we established the Turing machine as our model of a general purpose computer. We presented several examples of problems that are solvable on a Turing machine and gave one example of a problem, A_{TM}, that is computationally unsolvable. In this chapter we examine several additional unsolvable problems. In doing so, we introduce the primary method for proving that problems are computationally unsolvable. It is called ***reducibility***.

A ***reduction*** is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem. Such reducibilities come up often in everyday life, even if we don't usually refer to them in this way.

For example, suppose that you want to find your way around a new city. You know that doing so would be easy if you had a map. Thus, you can reduce the problem of finding your way around the city to the problem of obtaining a map of the city.

Reducibility always involves two problems, which we call A and B. If A reduces to B, we can use a solution to B to solve A. So in our example, A is the problem of finding your way around the city and B is the problem of obtaining a map. Note that reducibility says nothing about solving A or B alone, but only about the solvability of A in the presence of a solution to B.

The following are further examples of reducibilities. The problem of traveling from Boston to Paris reduces to the problem of buying a plane ticket between the two cities. That problem in turn reduces to the problem of earning the money for the ticket. And that problem reduces to the problem of finding a job.

Reducibility also occurs in mathematical problems. For example, the problem of measuring the area of a rectangle reduces to the problem of measuring its length and width. The problem of solving a system of linear equations reduces to the problem of inverting a matrix.

Reducibility plays an important role in classifying problems by decidability, and later in complexity theory as well. When A is reducible to B, solving A cannot be harder than solving B because a solution to B gives a solution to A. In terms of computability theory, if A is reducible to B and B is decidable, A also is decidable. Equivalently, if A is undecidable and reducible to B, B is undecidable. This last version is key to proving that various problems are undecidable.

In short, our method for proving that a problem is undecidable will be to show that some other problem already known to be undecidable reduces to it.

5.1

UNDECIDABLE PROBLEMS FROM LANGUAGE THEORY

We have already established the undecidability of A_{TM}, the problem of determining whether a Turing machine accepts a given input. Let's consider a related problem, $HALT_{\mathsf{TM}}$, the problem of determining whether a Turing machine halts (by accepting or rejecting) on a given input. This problem is widely known as the ***halting problem***. We use the undecidability of A_{TM} to prove the undecidability of the halting problem by reducing A_{TM} to $HALT_{\mathsf{TM}}$. Let

$$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}.$$

THEOREM 5.1

$HALT_{\mathsf{TM}}$ is undecidable.

PROOF IDEA This proof is by contradiction. We assume that $HALT_{\mathsf{TM}}$ is decidable and use that assumption to show that A_{TM} is decidable, contradicting Theorem 4.11. The key idea is to show that A_{TM} is reducible to $HALT_{\mathsf{TM}}$.

Let's assume that we have a TM R that decides $HALT_{\mathsf{TM}}$. Then we use R to construct S, a TM that decides A_{TM}. To get a feel for the way to construct S, pretend that you are S. Your task is to decide A_{TM}. You are given an input of the form $\langle M, w\rangle$. You must output *accept* if M accepts w, and you must output *reject* if M loops or rejects on w. Try simulating M on w. If it accepts or rejects, do the same. But you may not be able to determine whether M is looping, and in that case your simulation will not terminate. That's bad because you are a decider and thus never permitted to loop. So this idea by itself does not work.

Instead, use the assumption that you have TM R that decides $HALT_{\mathsf{TM}}$. With R, you can test whether M halts on w. If R indicates that M doesn't halt on w, reject because $\langle M, w\rangle$ isn't in A_{TM}. However, if R indicates that M does halt on w, you can do the simulation without any danger of looping.

Thus, if TM R exists, we can decide A_{TM}, but we know that A_{TM} is undecidable. By virtue of this contradiction, we can conclude that R does not exist. Therefore, $HALT_{\mathsf{TM}}$ is undecidable.

PROOF Let's assume for the purpose of obtaining a contradiction that TM R decides $HALT_{\mathsf{TM}}$. We construct TM S to decide A_{TM}, with S operating as follows.

S = "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*."

Clearly, if R decides $HALT_{\mathsf{TM}}$, then S decides A_{TM}. Because A_{TM} is undecidable, $HALT_{\mathsf{TM}}$ also must be undecidable.

Theorem 5.1 illustrates our strategy for proving that a problem is undecidable. This strategy is common to most proofs of undecidability, except for the undecidability of A_{TM} itself, which is proved directly via the diagonalization method.

We now present several other theorems and their proofs as further examples of the reducibility method for proving undecidability. Let

$$E_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}.$$

THEOREM 5.2

E_{TM} is undecidable.

PROOF IDEA We follow the pattern adopted in Theorem 5.1. We assume that E_{TM} is decidable and then show that A_{TM} is decidable—a contradiction. Let R be a TM that decides E_{TM}. We use R to construct TM S that decides A_{TM}. How will S work when it receives input $\langle M, w\rangle$?

One idea is for S to run R on input $\langle M\rangle$ and see whether it accepts. If it does, we know that $L(M)$ is empty and therefore that M does not accept w. But if R rejects $\langle M\rangle$, all we know is that $L(M)$ is not empty and therefore that M accepts some string—but we still do not know whether M accepts the particular string w. So we need to use a different idea.

Instead of running R on $\langle M \rangle$, we run R on a modification of $\langle M \rangle$. We modify $\langle M \rangle$ to guarantee that M rejects all strings except w, but on input w it works as usual. Then we use R to determine whether the modified machine recognizes the empty language. The only string the machine can now accept is w, so its language will be nonempty iff it accepts w. If R accepts when it is fed a description of the modified machine, we know that the modified machine doesn't accept anything and that M doesn't accept w.

PROOF Let's write the modified machine described in the proof idea using our standard notation. We call it M_1.

M_1 = "On input x:

1. If $x \neq w$, *reject*.
2. If $x = w$, run M on input w and *accept* if M does."

This machine has the string w as part of its description. It conducts the test of whether $x = w$ in the obvious way, by scanning the input and comparing it character by character with w to determine whether they are the same.

Putting all this together, we assume that TM R decides E_{TM} and construct TM S that decides A_{TM} as follows.

S = "On input $\langle M, w \rangle$, an encoding of a TM M and a string w:

1. Use the description of M and w to construct the TM M_1 just described.
2. Run R on input $\langle M_1 \rangle$.
3. If R accepts, *reject*; if R rejects, *accept*."

Note that S must actually be able to compute a description of M_1 from a description of M and w. It is able to do so because it only needs to add extra states to M that perform the $x = w$ test.

If R were a decider for E_{TM}, S would be a decider for A_{TM}. A decider for A_{TM} cannot exist, so we know that E_{TM} must be undecidable.

Another interesting computational problem regarding Turing machines concerns determining whether a given Turing machine recognizes a language that also can be recognized by a simpler computational model. For example, we let $REGULAR_{\mathsf{TM}}$ be the problem of determining whether a given Turing machine has an equivalent finite automaton. This problem is the same as determining whether the Turing machine recognizes a regular language. Let

$$REGULAR_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}.$$

THEOREM 5.3

$REGULAR_{\mathsf{TM}}$ is undecidable.

PROOF IDEA As usual for undecidability theorems, this proof is by reduction from A_{TM}. We assume that $REGULAR_{\mathsf{TM}}$ is decidable by a TM R and use this assumption to construct a TM S that decides A_{TM}. Less obvious now is how to use R's ability to assist S in its task. Nonetheless, we can do so.

The idea is for S to take its input $\langle M, w\rangle$ and modify M so that the resulting TM recognizes a regular language if and only if M accepts w. We call the modified machine M_2. We design M_2 to recognize the nonregular language $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ if M does not accept w, and to recognize the regular language Σ^* if M accepts w. We must specify how S can construct such an M_2 from M and w. Here, M_2 works by automatically accepting all strings in $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$. In addition, if M accepts w, M_2 accepts all other strings.

Note that the TM M_2 is *not* constructed for the purposes of actually running it on some input—a common confusion. We construct M_2 only for the purpose of feeding its description into the decider for $REGULAR_{\mathsf{TM}}$ that we have assumed to exist. Once this decider returns its answer, we can use it to obtain the answer to whether M accepts w. Thus, we can decide A_{TM}, a contradiction.

PROOF We let R be a TM that decides $REGULAR_{\mathsf{TM}}$ and construct TM S to decide A_{TM}. Then S works in the following manner.

S = "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Construct the following TM M_2.
   M_2 = "On input x:
   1. If x has the form $\mathtt{0}^n\mathtt{1}^n$, *accept*.
   2. If x does not have this form, run M on input w and *accept* if M accepts w."
2. Run R on input $\langle M_2\rangle$.
3. If R accepts, *accept*; if R rejects, *reject*."

Similarly, the problems of testing whether the language of a Turing machine is a context-free language, a decidable language, or even a finite language can be shown to be undecidable with similar proofs. In fact, a general result, called Rice's theorem, states that determining *any property* of the languages recognized by Turing machines is undecidable. We give Rice's theorem in Problem 5.28.

So far, our strategy for proving languages undecidable involves a reduction from A_{TM}. Sometimes reducing from some other undecidable language, such as E_{TM}, is more convenient when we are showing that certain languages are undecidable. Theorem 5.4 shows that testing the equivalence of two Turing

machines is an undecidable problem. We could prove it by a reduction from A_{TM}, but we use this opportunity to give an example of an undecidability proof by reduction from E_{TM}. Let

$$EQ_{\mathsf{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}.$$

THEOREM 5.4

EQ_{TM} is undecidable.

PROOF IDEA Show that if EQ_{TM} were decidable, E_{TM} also would be decidable by giving a reduction from E_{TM} to EQ_{TM}. The idea is simple. E_{TM} is the problem of determining whether the language of a TM is empty. EQ_{TM} is the problem of determining whether the languages of two TMs are the same. If one of these languages happens to be $\emptyset$, we end up with the problem of determining whether the language of the other machine is empty—that is, the E_{TM} problem. So in a sense, the E_{TM} problem is a special case of the EQ_{TM} problem wherein one of the machines is fixed to recognize the empty language. This idea makes giving the reduction easy.

PROOF We let TM R decide EQ_{TM} and construct TM S to decide E_{TM} as follows.

S = "On input $\langle M\rangle$, where M is a TM:

1. Run R on input $\langle M, M_1\rangle$, where M_1 is a TM that rejects all inputs.
2. If R accepts, *accept*; if R rejects, *reject*."

If R decides EQ_{TM}, S decides E_{TM}. But E_{TM} is undecidable by Theorem 5.2, so EQ_{TM} also must be undecidable.

REDUCTIONS VIA COMPUTATION HISTORIES

The computation history method is an important technique for proving that A_{TM} is reducible to certain languages. This method is often useful when the problem to be shown undecidable involves testing for the existence of something. For example, this method is used to show the undecidability of Hilbert's tenth problem, testing for the existence of integral roots in a polynomial.

The computation history for a Turing machine on an input is simply the sequence of configurations that the machine goes through as it processes the input. It is a complete record of the computation of this machine.

DEFINITION 5.5

Let M be a Turing machine and w an input string. An ***accepting computation history*** for M on w is a sequence of configurations, $C_1, C_2, \ldots, C_l$, where C_1 is the start configuration of M on w, C_l is an accepting configuration of M, and each C_i legally follows from C_{i-1} according to the rules of M. A ***rejecting computation history*** for M on w is defined similarly, except that C_l is a rejecting configuration.

Computation histories are finite sequences. If M doesn't halt on w, no accepting or rejecting computation history exists for M on w. Deterministic machines have at most one computation history on any given input. Nondeterministic machines may have many computation histories on a single input, corresponding to the various computation branches. For now, we continue to focus on deterministic machines. Our first undecidability proof using the computation history method concerns a type of machine called a linear bounded automaton.

DEFINITION 5.6

A ***linear bounded automaton*** is a restricted type of Turing machine wherein the tape head isn't permitted to move off the portion of the tape containing the input. If the machine tries to move its head off either end of the input, the head stays where it is—in the same way that the head will not move off the left-hand end of an ordinary Turing machine's tape.

A linear bounded automaton is a Turing machine with a limited amount of memory, as shown schematically in the following figure. It can only solve problems requiring memory that can fit within the tape used for the input. Using a tape alphabet larger than the input alphabet allows the available memory to be increased up to a constant factor. Hence we say that for an input of length n, the amount of memory available is linear in n—thus the name of this model.

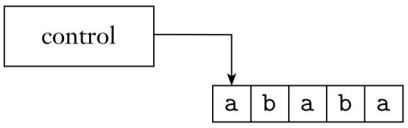


FIGURE 5.7
Schematic of a linear bounded automaton

Despite their memory constraint, linear bounded automata (LBAs) are quite powerful. For example, the deciders for A_{DFA}, A_{CFG}, E_{DFA}, and E_{CFG} all are LBAs. Every CFL can be decided by an LBA. In fact, coming up with a decidable language that can't be decided by an LBA takes some work. We develop the techniques to do so in Chapter 9.

Here, A_{LBA} is the problem of determining whether an LBA accepts its input. Even though A_{LBA} is the same as the undecidable problem A_{TM} where the Turing machine is restricted to be an LBA, we can show that A_{LBA} is decidable. Let

$$A_{\mathsf{LBA}} = \{\langle M, w\rangle \mid M \text{ is an LBA that accepts string } w\}.$$

Before proving the decidability of A_{LBA}, we find the following lemma useful. It says that an LBA can have only a limited number of configurations when a string of length n is the input.

LEMMA 5.8

Let M be an LBA with q states and g symbols in the tape alphabet. There are exactly qng^n distinct configurations of M for a tape of length n.

PROOF Recall that a configuration of M is like a snapshot in the middle of its computation. A configuration consists of the state of the control, position of the head, and contents of the tape. Here, M has q states. The length of its tape is n, so the head can be in one of n positions, and g^n possible strings of tape symbols appear on the tape. The product of these three quantities is the total number of different configurations of M with a tape of length n.

THEOREM 5.9

A_{LBA} is decidable.

PROOF IDEA In order to decide whether LBA M accepts input w, we simulate M on w. During the course of the simulation, if M halts and accepts or rejects, we accept or reject accordingly. The difficulty occurs if M loops on w. We need to be able to detect looping so that we can halt and reject.

The idea for detecting when M is looping is that as M computes on w, it goes from configuration to configuration. If M ever repeats a configuration, it would go on to repeat this configuration over and over again and thus be in a loop. Because M is an LBA, the amount of tape available to it is limited. By Lemma 5.8, M can be in only a limited number of configurations on this amount of tape. Therefore, only a limited amount of time is available to M before it will enter some configuration that it has previously entered. Detecting that M is looping is possible by simulating M for the number of steps given by Lemma 5.8. If M has not halted by then, it must be looping.

PROOF The algorithm that decides A_{LBA} is as follows.

L = "On input $\langle M, w\rangle$, where M is an LBA and w is a string:

1. Simulate M on w for qng^n steps or until it halts.
2. If M has halted, *accept* if it has accepted and *reject* if it has rejected. If it has not halted, *reject*."

If M on w has not halted within qng^n steps, it must be repeating a configuration according to Lemma 5.8 and therefore looping. That is why our algorithm rejects in this instance.

Theorem 5.9 shows that LBAs and TMs differ in one essential way: For LBAs the acceptance problem is decidable, but for TMs it isn't. However, certain other problems involving LBAs remain undecidable. One is the emptiness problem $E_{\mathsf{LBA}} = \{\langle M\rangle |\ M \text{ is an LBA where } L(M) = \emptyset\}$. To prove that E_{LBA} is undecidable, we give a reduction that uses the computation history method.

THEOREM 5.10

E_{LBA} is undecidable.

PROOF IDEA This proof is by reduction from A_{TM}. We show that if E_{LBA} were decidable, A_{TM} would also be. Suppose that E_{LBA} is decidable. How can we use this supposition to decide A_{TM}?

For a TM M and an input w, we can determine whether M accepts w by constructing a certain LBA B and then testing whether $L(B)$ is empty. The language that B recognizes comprises all accepting computation histories for M on w. If M accepts w, this language contains one string and so is nonempty. If M does not accept w, this language is empty. If we can determine whether B's language is empty, clearly we can determine whether M accepts w.

Now we describe how to construct B from M and w. Note that we need to show more than the mere existence of B. We have to show how a Turing machine can obtain a description of B, given descriptions of M and w.

As in the previous reductions we've given for proving undecidability, we construct B only to feed its description into the presumed E_{LBA} decider, but not to run B on some input.

We construct B to accept its input x if x is an accepting computation history for M on w. Recall that an accepting computation history is the sequence of configurations, $C_1, C_2, \ldots, C_l$ that M goes through as it accepts some string w. For the purposes of this proof, we assume that the accepting computation history is presented as a single string with the configurations separated from each other by the # symbol, as shown in Figure 5.11.

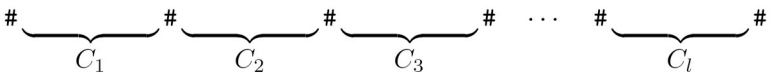


FIGURE 5.11
A possible input to B

The LBA B works as follows. When it receives an input x, B is supposed to accept if x is an accepting computation history for M on w. First, B breaks up x according to the delimiters into strings $C_1, C_2, \ldots, C_l$. Then B determines whether the C_i's satisfy the three conditions of an accepting computation history.

1. C_1 is the start configuration for M on w.
2. Each C_{i+1} legally follows from C_i.
3. C_l is an accepting configuration for M.

The start configuration C_1 for M on w is the string $q_0 w_1 w_2 \cdots w_n$, where q_0 is the start state for M on w. Here, B has this string directly built in, so it is able to check the first condition. An accepting configuration is one that contains the q_{accept} state, so B can check the third condition by scanning C_l for q_{accept}. The second condition is the hardest to check. For each pair of adjacent configurations, B checks on whether C_{i+1} legally follows from C_i. This step involves verifying that C_i and C_{i+1} are identical except for the positions under and adjacent to the head in C_i. These positions must be updated according to the transition function of M. Then B verifies that the updating was done properly by zig-zagging between corresponding positions of C_i and C_{i+1}. To keep track of the current positions while zig-zagging, B marks the current position with dots on the tape. Finally, if conditions 1, 2, and 3 are satisfied, B accepts its input.

By inverting the decider's answer, we obtain the answer to whether M accepts w. Thus we can decide A_{TM}, a contradiction.

PROOF Now we are ready to state the reduction of A_{TM} to E_{LBA}. Suppose that TM R decides E_{LBA}. Construct TM S to decide A_{TM} as follows.

S = "On input $\langle M, w \rangle$, where M is a TM and w is a string:

1. Construct LBA B from M and w as described in the proof idea.
2. Run R on input $\langle B \rangle$.
3. If R rejects, *accept*; if R accepts, *reject*."

If R accepts $\langle B \rangle$, then $L(B) = \emptyset$. Thus, M has no accepting computation history on w and M doesn't accept w. Consequently, S rejects $\langle M, w \rangle$. Similarly, if R rejects $\langle B \rangle$, the language of B is nonempty. The only string that B can accept is an accepting computation history for M on w. Thus, M must accept w. Consequently, S accepts $\langle M, w \rangle$. Figure 5.12 illustrates LBA B.

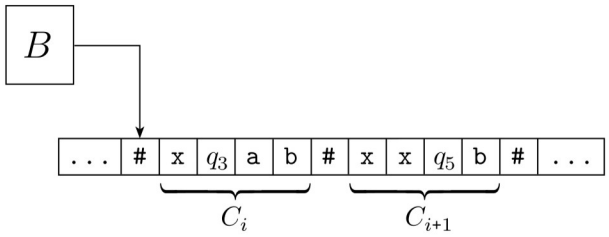


FIGURE 5.12
LBA B checking a TM computation history

We can also use the technique of reduction via computation histories to establish the undecidability of certain problems related to context-free grammars and pushdown automata. Recall that in Theorem 4.8 we presented an algorithm to decide whether a context-free grammar generates any strings—that is, whether $L(G) = \emptyset$. Now we show that a related problem is undecidable. It is the problem of determining whether a context-free grammar generates all possible strings. Proving that this problem is undecidable is the main step in showing that the equivalence problem for context-free grammars is undecidable. Let

$$ALL_{\mathsf{CFG}} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \Sigma^*\}.$$

THEOREM 5.13

ALL_{CFG} is undecidable.

PROOF This proof is by contradiction. To get the contradiction, we assume that ALL_{CFG} is decidable and use this assumption to show that A_{TM} is decidable. This proof is similar to that of Theorem 5.10 but with a small extra twist: It is a reduction from A_{TM} via computation histories, but we modify the representation of the computation histories slightly for a technical reason that we will explain later.

We now describe how to use a decision procedure for ALL_{CFG} to decide A_{TM}. For a TM M and an input w, we construct a CFG G that generates all strings if and only if M does not accept w. So if M does accept w, G does *not* generate some particular string. This string is—guess what—the accepting computation history for M on w. That is, G is designed to generate all strings that are *not* accepting computation histories for M on w.

To make the CFG G generate all strings that fail to be an accepting computation history for M on w, we utilize the following strategy. A string may fail to be an accepting computation history for several reasons. An accepting computation history for M on w appears as $\#C_1\#C_2\#\cdots\#C_l\#$, where C_i is the configuration of M on the ith step of the computation on w. Then, G generates all strings

1. that *do not* start with C_1,
2. that *do not* end with an accepting configuration, or
3. in which some C_i *does not* properly yield C_{i+1} under the rules of M.

If M does not accept w, no accepting computation history exists, so *all* strings fail in one way or another. Therefore, G would generate all strings, as desired.

Now we get down to the actual construction of G. Instead of constructing G, we construct a PDA D. We know that we can use the construction given in Theorem 2.20 (page 117) to convert D to a CFG. We do so because, for our purposes, designing a PDA is easier than designing a CFG. In this instance, D will start by nondeterministically branching to guess which of the preceding three conditions to check. One branch checks on whether the beginning of the input string is C_1 and accepts if it isn't. Another branch checks on whether the input string ends with a configuration containing the accept state, q_{accept}, and accepts if it isn't.

The third branch is supposed to accept if some C_i does not properly yield C_{i+1}. It works by scanning the input until it nondeterministically decides that it has come to C_i. Next, it pushes C_i onto the stack until it comes to the end as marked by the # symbol. Then D pops the stack to compare with C_{i+1}. They are supposed to match except around the head position, where the difference is dictated by the transition function of M. Finally, D accepts if it discovers a mismatch or an improper update.

The problem with this idea is that when D pops C_i off the stack, it is in reverse order and not suitable for comparison with C_{i+1}. At this point, the twist in the proof appears: We write the accepting computation history differently. Every other configuration appears in reverse order. The odd positions remain written in the forward order, but the even positions are written backward. Thus, an accepting computation history would appear as shown in the following figure.

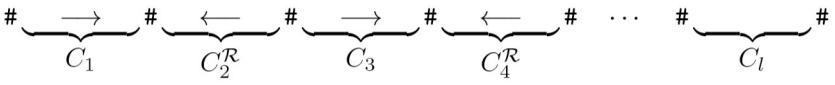


FIGURE 5.14
Every other configuration written in reverse order

In this modified form, the PDA is able to push a configuration so that when it is popped, the order is suitable for comparison with the next one. We design D to accept any string that is not an accepting computation history in the modified form.

In Exercise 5.1 you can use Theorem 5.13 to show that EQ_{CFG} is undecidable.

5.2

A SIMPLE UNDECIDABLE PROBLEM

In this section we show that the phenomenon of undecidability is not confined to problems concerning automata. We give an example of an undecidable problem concerning simple manipulations of strings. It is called the ***Post Correspondence Problem***, or ***PCP***.

We can describe this problem easily as a type of puzzle. We begin with a collection of dominos, each containing two strings, one on each side. An individual domino looks like

$$\left[\frac{\texttt{a}}{\texttt{ab}}\right]$$

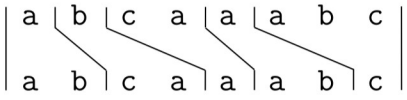
and a collection of dominos looks like

$$\left\{\left[\frac{\texttt{b}}{\texttt{ca}}\right], \left[\frac{\texttt{a}}{\texttt{ab}}\right], \left[\frac{\texttt{ca}}{\texttt{a}}\right], \left[\frac{\texttt{abc}}{\texttt{c}}\right]\right\}.$$

The task is to make a list of these dominos (repetitions permitted) so that the string we get by reading off the symbols on the top is the same as the string of symbols on the bottom. This list is called a ***match***. For example, the following list is a match for this puzzle.

$$\left[\frac{\texttt{a}}{\texttt{ab}}\right]\left[\frac{\texttt{b}}{\texttt{ca}}\right]\left[\frac{\texttt{ca}}{\texttt{a}}\right]\left[\frac{\texttt{a}}{\texttt{ab}}\right]\left[\frac{\texttt{abc}}{\texttt{c}}\right]$$

Reading off the top string we get `abcaaabc`, which is the same as reading off the bottom. We can also depict this match by deforming the dominos so that the corresponding symbols from top and bottom line up.



For some collections of dominos, finding a match may not be possible. For example, the collection

$$\left\{\left[\frac{\texttt{abc}}{\texttt{ab}}\right], \left[\frac{\texttt{ca}}{\texttt{a}}\right], \left[\frac{\texttt{acc}}{\texttt{ba}}\right]\right\}$$

cannot contain a match because every top string is longer than the corresponding bottom string.

The Post Correspondence Problem is to determine whether a collection of dominos has a match. This problem is unsolvable by algorithms.

Before getting to the formal statement of this theorem and its proof, let's state the problem precisely and then express it as a language. An instance of the PCP is a collection P of dominos

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\},$$

and a match is a sequence $i_1, i_2, \ldots, i_l$, where $t_{i_1} t_{i_2} \cdots t_{i_l} = b_{i_1} b_{i_2} \cdots b_{i_l}$. The problem is to determine whether P has a match. Let

$$PCP = \{\langle P \rangle \mid P \text{ is an instance of the Post Correspondence Problem with a match}\}.$$

THEOREM 5.15

PCP is undecidable.

PROOF IDEA Conceptually this proof is simple, though it involves many details. The main technique is reduction from A_{TM} via accepting computation histories. We show that from any TM M and input w, we can construct an instance P where a match is an accepting computation history for M on w. If we could determine whether the instance has a match, we would be able to determine whether M accepts w.

How can we construct P so that a match is an accepting computation history for M on w? We choose the dominos in P so that making a match forces a simulation of M to occur. In the match, each domino links a position or positions in one configuration with the corresponding one(s) in the next configuration.

Before getting to the construction, we handle three small technical points. (Don't worry about them too much on your initial reading through this construction.) First, for convenience in constructing P, we assume that M on w never attempts to move its head off the left-hand end of the tape. That requires first altering M to prevent this behavior. Second, if $w = \varepsilon$, we use the string $\sqcup$ in place of w in the construction. Third, we modify the PCP to require that a match starts with the first domino,

$$\left[\frac{t_1}{b_1}\right].$$

Later we show how to eliminate this requirement. We call this problem the Modified Post Correspondence Problem (MPCP). Let

$$MPCP = \{\langle P \rangle \mid P \text{ is an instance of the Post Correspondence Problem with a match that starts with the first domino}\}.$$

Now let's move into the details of the proof and design P to simulate M on w.

PROOF We let TM R decide the PCP and construct S deciding A_{TM}. Let

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

where Q, Σ, Γ, and δ are the state set, input alphabet, tape alphabet, and transition function of M, respectively.

In this case, S constructs an instance of the PCP P that has a match iff M accepts w. To do that, S first constructs an instance P' of the MPCP. We describe the construction in seven parts, each of which accomplishes a particular aspect of simulating M on w. To explain what we are doing, we interleave the construction with an example of the construction in action.

Part 1. The construction begins in the following manner.

$$\text{Put } \left[\frac{\#}{\#q_0w_1w_2\cdots w_n\#}\right] \text{ into } P' \text{ as the first domino } \left[\frac{t_1}{b_1}\right].$$

Because P' is an instance of the MPCP, the match must begin with this domino. Thus, the bottom string begins correctly with $C_1 = q_0w_1w_2\cdots w_n$, the first configuration in the accepting computation history for M on w, as shown in the following figure.

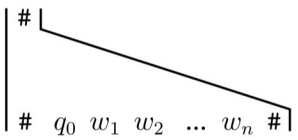


FIGURE 5.16
Beginning of the MPCP match

In this depiction of the partial match achieved so far, the bottom string consists of $\#q_0w_1w_2\cdots w_n\#$ and the top string consists only of #. To get a match, we need to extend the top string to match the bottom string. We provide additional dominos to allow this extension. The additional dominos cause M's next configuration to appear at the extension of the bottom string by forcing a single-step simulation of M.

In parts 2, 3, and 4, we add to P' dominos that perform the main part of the simulation. Part 2 handles head motions to the right, part 3 handles head motions to the left, and part 4 handles the tape cells not adjacent to the head.

Part 2. For every $a, b \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{\text{reject}}$,

$$\text{if } \delta(q, a) = (r, b, \text{R}), \text{ put } \left[\frac{qa}{br}\right] \text{ into } P'.$$

Part 3. For every $a, b, c \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{\text{reject}}$,

$$\text{if } \delta(q, a) = (r, b, \text{L}), \text{ put } \left[\frac{cqa}{rcb}\right] \text{ into } P'.$$

Part 4. For every $a \in \Gamma$,

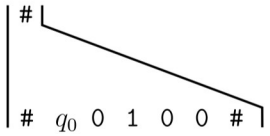
$$\text{put } \left[\frac{a}{a}\right] \text{ into } P'.$$

Now we make up a hypothetical example to illustrate what we have built so far. Let $\Gamma = \{0, 1, 2, \sqcup\}$. Say that w is the string 0100 and that the start state of M is q_0. In state q_0, upon reading a 0, let's say that the transition function dictates that M enters state q_7, writes a 2 on the tape, and moves its head to the right. In other words, $\delta(q_0, 0) = (q_7, 2, \mathrm{R})$.

Part 1 places the domino

$$\left[\frac{\#}{\#q_0 0100\#}\right] = \left[\frac{t_1}{b_1}\right]$$

in P', and the match begins



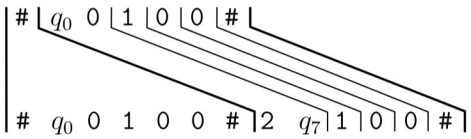
In addition, part 2 places the domino

$$\left[\frac{q_0 0}{2 q_7}\right]$$

as $\delta(q_0, 0) = (q_7, 2, \mathrm{R})$ and part 4 places the dominos

$$\left[\frac{0}{0}\right], \left[\frac{1}{1}\right], \left[\frac{2}{2}\right], \text{ and } \left[\frac{\sqcup}{\sqcup}\right]$$

in P', as 0, 1, 2, and $\sqcup$ are the members of Γ. Together with part 5, that allows us to extend the match to



Thus, the dominos of parts 2, 3, and 4 let us extend the match by adding the second configuration after the first one. We want this process to continue, adding the third configuration, then the fourth, and so on. For it to happen, we need to add one more domino for copying the # symbol.

Part 5.

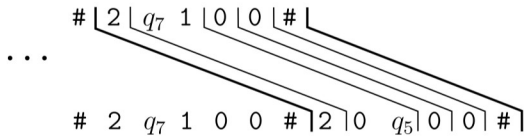
$$\text{Put } \left[\frac{\#}{\#}\right] \text{ and } \left[\frac{\#}{\sqcup\#}\right] \text{ into } P'.$$

The first of these dominos allows us to copy the # symbol that marks the separation of the configurations. In addition to that, the second domino allows us to add a blank symbol $\sqcup$ at the end of the configuration to simulate the infinitely many blanks to the right that are suppressed when we write the configuration.

Continuing with the example, let's say that in state q_7, upon reading a 1, M goes to state q_5, writes a 0, and moves the head to the right. That is, $\delta(q_7, 1) = (q_5, 0, \mathrm{R})$. Then we have the domino

$$\left[\frac{q_7 1}{0 q_5}\right] \text{ in } P'.$$

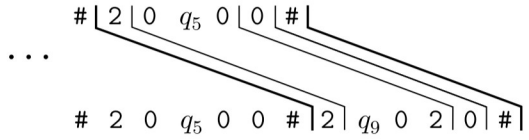
So the latest partial match extends to



Then, suppose that in state q_5, upon reading a 0, M goes to state q_9, writes a 2, and moves its head to the left. So $\delta(q_5, 0) = (q_9, 2, \mathrm{L})$. Then we have the dominos

$$\left[\frac{0 q_5 0}{q_9 0 2}\right], \left[\frac{1 q_5 0}{q_9 1 2}\right], \left[\frac{2 q_5 0}{q_9 2 2}\right], \text{ and } \left[\frac{\sqcup q_5 0}{q_9 \sqcup 2}\right].$$

The first one is relevant because the symbol to the left of the head is a 0. The preceding partial match extends to

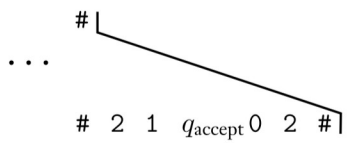


Note that as we construct a match, we are forced to simulate M on input w. This process continues until M reaches a halting state. If the accept state occurs, we want to let the top of the partial match "catch up" with the bottom so that the match is complete. We can arrange for that to happen by adding additional dominos.

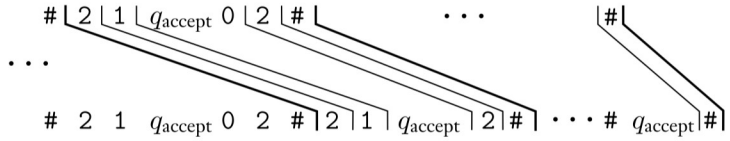
Part 6. For every $a \in \Gamma$,

$$\text{put } \left[\frac{a\ q_{\text{accept}}}{q_{\text{accept}}}\right] \text{ and } \left[\frac{q_{\text{accept}}\ a}{q_{\text{accept}}}\right] \text{ into } P'.$$

This step has the effect of adding "pseudo-steps" of the Turing machine after it has halted, where the head "eats" adjacent symbols until none are left. Continuing with the example, if the partial match up to the point when the machine halts in the accept state is



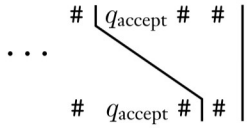
The dominos we have just added allow the match to continue:



Part 7. Finally, we add the domino

$$\left[\frac{q_{\text{accept}}\#\#}{\#}\right]$$

and complete the match:



That concludes the construction of P'. Recall that P' is an instance of the MPCP whereby the match simulates the computation of M on w. To finish the proof, we recall that the MPCP differs from the PCP in that the match is required to start with the first domino in the list. If we view P' as an instance of

the PCP instead of the MPCP, it obviously has a match, regardless of whether M accepts w. Can you find it? (Hint: It is very short.)

We now show how to convert P' to P, an instance of the PCP that still simulates M on w. We do so with a somewhat technical trick. The idea is to take the requirement that the match starts with the first domino and build it directly into the problem instance itself so that it becomes enforced automatically. After that, the requirement isn't needed. We introduce some notation to implement this idea.

Let $u = u_1 u_2 \cdots u_n$ be any string of length n. Define $\star u$, $u\star$, and $\star u\star$ to be the three strings

$$\begin{aligned} \star u &= *u_1 * u_2 * u_3 * \cdots * u_n \\ u\star &= u_1 * u_2 * u_3 * \cdots * u_n * \\ \star u\star &= *u_1 * u_2 * u_3 * \cdots * u_n *. \end{aligned}$$

Here, $\star u$ adds the symbol $*$ before every character in u, $u\star$ adds one after each character in u, and $\star u\star$ adds one both before and after each character in u.

To convert P' to P, an instance of the PCP, we do the following. If P' were the collection

$$\left\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \left[\frac{t_3}{b_3}\right], \cdots, \left[\frac{t_k}{b_k}\right]\right\},$$

we let P be the collection

$$\left\{\left[\frac{\star t_1}{\star b_1\star}\right], \left[\frac{\star t_1}{b_1\star}\right], \left[\frac{\star t_2}{b_2\star}\right], \left[\frac{\star t_3}{b_3\star}\right], \cdots, \left[\frac{\star t_k}{b_k\star}\right], \left[\frac{*\diamond}{\diamond}\right]\right\}.$$

Considering P as an instance of the PCP, we see that the only domino that could possibly start a match is the first one,

$$\left[\frac{\star t_1}{\star b_1\star}\right],$$

because it is the only one where both the top and the bottom start with the same symbol—namely, $*$. Besides forcing the match to start with the first domino, the presence of the $*$s doesn't affect possible matches because they simply interleave with the original symbols. The original symbols now occur in the even positions of the match. The domino

$$\left[\frac{*\diamond}{\diamond}\right]$$

is there to allow the top to add the extra $*$ at the end of the match.

5.3 MAPPING REDUCIBILITY

We have shown how to use the reducibility technique to prove that various problems are undecidable. In this section we formalize the notion of reducibility. Doing so allows us to use reducibility in more refined ways, such as for proving that certain languages are not Turing-recognizable and for applications in complexity theory.

The notion of reducing one problem to another may be defined formally in one of several ways. The choice of which one to use depends on the application. Our choice is a simple type of reducibility called ***mapping reducibility***.[1]

Roughly speaking, being able to reduce problem A to problem B by using a mapping reducibility means that a computable function exists that converts instances of problem A to instances of problem B. If we have such a conversion function, called a *reduction*, we can solve A with a solver for B. The reason is that any instance of A can be solved by first using the reduction to convert it to an instance of B and then applying the solver for B. A precise definition of mapping reducibility follows shortly.

COMPUTABLE FUNCTIONS

A Turing machine computes a function by starting with the input to the function on the tape and halting with the output of the function on the tape.

DEFINITION 5.17

A function $f: \Sigma^* \longrightarrow \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

EXAMPLE 5.18

All usual, arithmetic operations on integers are computable functions. For example, we can make a machine that takes input $\langle m, n \rangle$ and returns $m + n$, the sum of m and n. We don't give any details here, leaving them as exercises. ■

EXAMPLE 5.19

Computable functions may be transformations of machine descriptions. For example, one computable function f takes input w and returns the description of a Turing machine $\langle M' \rangle$ if $w = \langle M \rangle$ is an encoding of a Turing machine M.

[1] It is called ***many–one reducibility*** in some other textbooks.

The machine M' is a machine that recognizes the same language as M, but never attempts to move its head off the left-hand end of its tape. The function f accomplishes this task by adding several states to the description of M. The function returns ε if w is not a legal encoding of a Turing machine. ■

FORMAL DEFINITION OF MAPPING REDUCIBILITY

Now we define mapping reducibility. As usual, we represent computational problems by languages.

DEFINITION 5.20

Language A is ***mapping reducible*** to language B, written $A \leq_{\mathrm{m}} B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***reduction*** from A to B.

The following figure illustrates mapping reducibility.

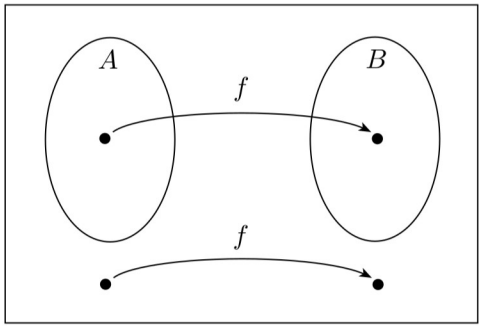


FIGURE 5.21
Function f reducing A to B

A mapping reduction of A to B provides a way to convert questions about membership testing in A to membership testing in B. To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$. The term *mapping reduction* comes from the function or mapping that provides the means of doing the reduction.

If one problem is mapping reducible to a second, previously solved problem, we can thereby obtain a solution to the original problem. We capture this idea in Theorem 5.22.

THEOREM 5.22

If $A \leq_{\mathrm{m}} B$ and B is decidable, then A is decidable.

PROOF We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

N = "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

Clearly, if $w \in A$, then $f(w) \in B$ because f is a reduction from A to B. Thus, M accepts $f(w)$ whenever $w \in A$. Therefore, N works as desired.

The following corollary of Theorem 5.22 has been our main tool for proving undecidability.

COROLLARY 5.23

If $A \leq_{\mathrm{m}} B$ and A is undecidable, then B is undecidable.

Now we revisit some of our earlier proofs that used the reducibility method to get examples of mapping reducibilities.

EXAMPLE 5.24

In Theorem 5.1 we used a reduction from A_{TM} to prove that $HALT_{\mathsf{TM}}$ is undecidable. This reduction showed how a decider for $HALT_{\mathsf{TM}}$ could be used to give a decider for A_{TM}. We can demonstrate a mapping reducibility from A_{TM} to $HALT_{\mathsf{TM}}$ as follows. To do so, we must present a computable function f that takes input of the form $\langle M, w\rangle$ and returns output of the form $\langle M', w'\rangle$, where

$$\langle M, w\rangle \in A_{\mathsf{TM}} \text{ if and only if } \langle M', w'\rangle \in HALT_{\mathsf{TM}}.$$

The following machine F computes a reduction f.

F = "On input $\langle M, w\rangle$:

1. Construct the following machine M'.
   M' = "On input x:
   1. Run M on x.
   2. If M accepts, *accept*.
   3. If M rejects, enter a loop."
2. Output $\langle M', w\rangle$."

A minor issue arises here concerning improperly formed input strings. If TM F determines that its input is not of the correct form as specified in the input line "On input $\langle M, w\rangle$:" and hence that the input is not in A_{TM}, the TM outputs a

string not in $HALT_{\mathsf{TM}}$. Any string not in $HALT_{\mathsf{TM}}$ will do. In general, when we describe a Turing machine that computes a reduction from A to B, improperly formed inputs are assumed to map to strings outside of B. ■

EXAMPLE 5.25

The proof of the undecidability of the Post Correspondence Problem in Theorem 5.15 contains two mapping reductions. First, it shows that $A_{\mathsf{TM}} \leq_{\mathrm{m}} MPCP$ and then it shows that $MPCP \leq_{\mathrm{m}} PCP$. In both cases, we can easily obtain the actual reduction function and show that it is a mapping reduction. As Exercise 5.6 shows, mapping reducibility is transitive, so these two reductions together imply that $A_{\mathsf{TM}} \leq_{\mathrm{m}} PCP$. ■

EXAMPLE 5.26

A mapping reduction from E_{TM} to EQ_{TM} lies in the proof of Theorem 5.4. In this case, the reduction f maps the input $\langle M \rangle$ to the output $\langle M, M_1 \rangle$, where M_1 is the machine that rejects all inputs. ■

EXAMPLE 5.27

The proof of Theorem 5.2 showing that E_{TM} is undecidable illustrates the difference between the formal notion of mapping reducibility that we have defined in this section and the informal notion of reducibility that we used earlier in this chapter. The proof shows that E_{TM} is undecidable by reducing A_{TM} to it. Let's see whether we can convert this reduction to a mapping reduction.

From the original reduction, we may easily construct a function f that takes input $\langle M, w \rangle$ and produces output $\langle M_1 \rangle$, where M_1 is the Turing machine described in that proof. But M accepts w iff $L(M_1)$ is *not* empty so f is a mapping reduction from A_{TM} to $\overline{E_{\mathsf{TM}}}$. It still shows that E_{TM} is undecidable because decidability is not affected by complementation, but it doesn't give a mapping reduction from A_{TM} to E_{TM}. In fact, no such reduction exists, as you are asked to show in Exercise 5.5. ■

The sensitivity of mapping reducibility to complementation is important in the use of reducibility to prove nonrecognizability of certain languages. We can also use mapping reducibility to show that problems are not Turing-recognizable. The following theorem is analogous to Theorem 5.22.

THEOREM 5.28

If $A \leq_{\mathrm{m}} B$ and B is Turing-recognizable, then A is Turing-recognizable.

The proof is the same as that of Theorem 5.22, except that M and N are recognizers instead of deciders.

COROLLARY 5.29

If $A \leq_{\mathrm{m}} B$ and A is not Turing-recognizable, then B is not Turing-recognizable.

In a typical application of this corollary, we let A be $\overline{A_{\mathsf{TM}}}$, the complement of A_{TM}. We know that $\overline{A_{\mathsf{TM}}}$ is not Turing-recognizable from Corollary 4.23. The definition of mapping reducibility implies that $A \leq_{\mathrm{m}} B$ means the same as $\overline{A} \leq_{\mathrm{m}} \overline{B}$. To prove that B isn't recognizable, we may show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} \overline{B}$. We can also use mapping reducibility to show that certain problems are neither Turing-recognizable nor co-Turing-recognizable, as in the following theorem.

THEOREM 5.30

EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.

PROOF First we show that EQ_{TM} is not Turing-recognizable. We do so by showing that A_{TM} is reducible to $\overline{EQ_{\mathsf{TM}}}$. The reducing function f works as follows.

F = "On input $\langle M, w\rangle$, where M is a TM and w a string:

1. Construct the following two machines, M_1 and M_2.
   M_1 = "On any input:
      1. *Reject*."
   M_2 = "On any input:
      1. Run M on w. If it accepts, *accept*."
2. Output $\langle M_1, M_2\rangle$."

Here, M_1 accepts nothing. If M accepts w, M_2 accepts everything, and so the two machines are not equivalent. Conversely, if M doesn't accept w, M_2 accepts nothing, and they are equivalent. Thus f reduces A_{TM} to $\overline{EQ_{\mathsf{TM}}}$, as desired.

To show that $\overline{EQ_{\mathsf{TM}}}$ is not Turing-recognizable, we give a reduction from A_{TM} to the complement of $\overline{EQ_{\mathsf{TM}}}$—namely, EQ_{TM}. Hence we show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} EQ_{\mathsf{TM}}$. The following TM G computes the reducing function g.

G = "On input $\langle M, w\rangle$, where M is a TM and w a string:

1. Construct the following two machines, M_1 and M_2.
   M_1 = "On any input:
      1. *Accept*."
   M_2 = "On any input:
      1. Run M on w.
      2. If it accepts, *accept*."
2. Output $\langle M_1, M_2\rangle$."

The only difference between f and g is in machine M_1. In f, machine M_1 always rejects, whereas in g it always accepts. In both f and g, M accepts w iff M_2 always accepts. In g, M accepts w iff M_1 and M_2 are equivalent. That is why g is a reduction from A_{TM} to EQ_{TM}.

EXERCISES

5.1 Show that EQ_{CFG} is undecidable.

5.2 Show that EQ_{CFG} is co-Turing-recognizable.

5.3 Find a match in the following instance of the Post Correspondence Problem.

$$\left\{\left[\frac{\texttt{ab}}{\texttt{abab}}\right], \left[\frac{\texttt{b}}{\texttt{a}}\right], \left[\frac{\texttt{aba}}{\texttt{b}}\right], \left[\frac{\texttt{aa}}{\texttt{a}}\right]\right\}$$

5.4 If $A \leq_{\mathrm{m}} B$ and B is a regular language, does that imply that A is a regular language? Why or why not?

[A]**5.5** Show that A_{TM} is not mapping reducible to E_{TM}. In other words, show that no computable function reduces A_{TM} to E_{TM}. (Hint: Use a proof by contradiction, and facts you already know about A_{TM} and E_{TM}.)

[A]**5.6** Show that $\leq_{\mathrm{m}}$ is a transitive relation.

[A]**5.7** Show that if A is Turing-recognizable and $A \leq_{\mathrm{m}} \overline{A}$, then A is decidable.

[A]**5.8** In the proof of Theorem 5.15, we modified the Turing machine M so that it never tries to move its head off the left-hand end of the tape. Suppose that we did not make this modification to M. Modify the PCP construction to handle this case.

PROBLEMS

5.9 Let $T = \{\langle M \rangle | \ M \text{ is a } \mathsf{TM} \text{ that accepts } w^{\mathcal{R}} \text{ whenever it accepts } w\}$. Show that T is undecidable.

[A]**5.10** Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape when it is run on input w. Formulate this problem as a language and show that it is undecidable.

[A]**5.11** Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape during the course of its computation on any input string. Formulate this problem as a language and show that it is undecidable.

5.12 Consider the problem of determining whether a single-tape Turing machine ever writes a blank symbol over a nonblank symbol during the course of its computation on any input string. Formulate this problem as a language and show that it is undecidable.

5.13 A ***useless state*** in a Turing machine is one that is never entered on any input string. Consider the problem of determining whether a Turing machine has any useless states. Formulate this problem as a language and show that it is undecidable.

5.14 Consider the problem of determining whether a Turing machine M on an input w ever attempts to move its head left when its head is on the left-most tape cell. Formulate this problem as a language and show that it is undecidable.

5.15 Consider the problem of determining whether a Turing machine M on an input w ever attempts to move its head left at any point during its computation on w. Formulate this problem as a language and show that it is decidable.

5.16 Let $\Gamma = \{0, 1, \sqcup\}$ be the tape alphabet for all TMs in this problem. Define the ***busy beaver function*** $BB: \mathcal{N} \longrightarrow \mathcal{N}$ as follows. For each value of k, consider all k-state TMs that halt when started with a blank tape. Let $BB(k)$ be the maximum number of 1s that remain on the tape among all of these machines. Show that BB is not a computable function.

5.17 Show that the Post Correspondence Problem is decidable over the unary alphabet $\Sigma = \{1\}$.

5.18 Show that the Post Correspondence Problem is undecidable over the binary alphabet $\Sigma = \{0,1\}$.

5.19 In the *silly Post Correspondence Problem*, *SPCP*, the top string in each pair has the same length as the bottom string. Show that the *SPCP* is decidable.

5.20 Prove that there exists an undecidable subset of $\{1\}^*$.

5.21 Let $AMBIG_{\mathsf{CFG}} = \{\langle G \rangle \mid G \text{ is an ambiguous CFG}\}$. Show that $AMBIG_{\mathsf{CFG}}$ is undecidable. (Hint: Use a reduction from *PCP*. Given an instance

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\}$$

of the Post Correspondence Problem, construct a CFG G with the rules

$$\begin{aligned} S &\to T \mid B \\ T &\to t_1 T \mathtt{a}_1 \mid \cdots \mid t_k T \mathtt{a}_k \mid t_1 \mathtt{a}_1 \mid \cdots \mid t_k \mathtt{a}_k \\ B &\to b_1 B \mathtt{a}_1 \mid \cdots \mid b_k B \mathtt{a}_k \mid b_1 \mathtt{a}_1 \mid \cdots \mid b_k \mathtt{a}_k , \end{aligned}$$

where $\mathtt{a}_1, \ldots, \mathtt{a}_k$ are new terminal symbols. Prove that this reduction works.)

5.22 Show that A is Turing-recognizable iff $A \leq_{\mathrm{m}} A_{\mathsf{TM}}$.

5.23 Show that A is decidable iff $A \leq_{\mathrm{m}} 0^*1^*$.

5.24 Let $J = \{w \mid \text{either } w = 0x \text{ for some } x \in A_{\mathsf{TM}}, \text{ or } w = 1y \text{ for some } y \in \overline{A_{\mathsf{TM}}}\}$. Show that neither J nor $\overline{J}$ is Turing-recognizable.

5.25 Give an example of an undecidable language B, where $B \leq_{\mathrm{m}} \overline{B}$.

5.26 Define a ***two-headed finite automaton*** (2DFA) to be a deterministic finite automaton that has two read-only, bidirectional heads that start at the left-hand end of the input tape and can be independently controlled to move in either direction. The tape of a 2DFA is finite and is just large enough to contain the input plus two additional blank tape cells, one on the left-hand end and one on the right-hand end, that serve as delimiters. A 2DFA accepts its input by entering a special accept state. For example, a 2DFA can recognize the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$.

a. Let $A_{\mathsf{2DFA}} = \{\langle M, x \rangle \mid M \text{ is a 2DFA and } M \text{ accepts } x\}$. Show that A_{2DFA} is decidable.

b. Let $E_{\mathsf{2DFA}} = \{\langle M \rangle \mid M \text{ is a 2DFA and } L(M) = \emptyset\}$. Show that E_{2DFA} is not decidable.

5.27 A ***two-dimensional finite automaton*** (2DIM-DFA) is defined as follows. The input is an $m \times n$ rectangle, for any $m, n \geq 2$. The squares along the boundary of the rectangle contain the symbol # and the internal squares contain symbols over the input alphabet Σ. The transition function $\delta\colon Q \times (\Sigma \cup \{\#\}) \longrightarrow Q \times \{\mathrm{L}, \mathrm{R}, \mathrm{U}, \mathrm{D}\}$ indicates the next state and the new head position (Left, Right, Up, Down). The machine accepts when it enters one of the designated accept states. It rejects if it tries to move off the input rectangle or if it never halts. Two such machines are equivalent if they accept the same rectangles. Consider the problem of determining whether two of these machines are equivalent. Formulate this problem as a language and show that it is undecidable.

A* **5.28** **Rice's theorem**. Let P be any nontrivial property of the language of a Turing machine. Prove that the problem of determining whether a given Turing machine's language has property P is undecidable.

In more formal terms, let P be a language consisting of Turing machine descriptions where P fulfills two conditions. First, P is nontrivial—it contains some, but not all, TM descriptions. Second, P is a property of the TM's language—whenever $L(M_1) = L(M_2)$, we have $\langle M_1 \rangle \in P$ iff $\langle M_2 \rangle \in P$. Here, M_1 and M_2 are any TMs. Prove that P is an undecidable language.

5.29 Show that both conditions in Problem 5.28 are necessary for proving that P is undecidable.

5.30 Use Rice's theorem, which appears in Problem 5.28, to prove the undecidability of each of the following languages.

A **a.** $\mathit{INFINITE}_{\mathsf{TM}} = \{\langle M \rangle |\ M \text{ is a TM and } L(M) \text{ is an infinite language}\}$.

b. $\{\langle M \rangle |\ M \text{ is a TM and } \mathtt{1011} \in L(M)\}$.

c. $\mathit{ALL}_{\mathsf{TM}} = \{\langle M \rangle |\ M \text{ is a TM and } L(M) = \Sigma^*\}$.

5.31 Let

$$f(x) = \begin{cases} 3x + 1 & \text{for odd } x \\ x/2 & \text{for even } x \end{cases}$$

for any natural number x. If you start with an integer x and iterate f, you obtain a sequence, $x, f(x), f(f(x)), \ldots$. Stop if you ever hit 1. For example, if $x = 17$, you get the sequence 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Extensive computer tests have shown that every starting point between 1 and a large positive integer gives a sequence that ends in 1. But the question of whether all positive starting points end up at 1 is unsolved; it is called the $3x + 1$ problem.

Suppose that A_{TM} were decidable by a TM H. Use H to describe a TM that is guaranteed to state the answer to the $3x + 1$ problem.

5.32 Prove that the following two languages are undecidable.

a. $\mathit{OVERLAP}_{\mathsf{CFG}} = \{\langle G, H \rangle |\ G \text{ and } H \text{ are CFGs where } L(G) \cap L(H) \neq \emptyset\}$. (Hint: Adapt the hint in Problem 5.21.)

b. $\mathit{PREFIX\text{-}FREE}_{\mathsf{CFG}} = \{\langle G \rangle |\ G \text{ is a CFG where } L(G) \text{ is prefix-free}\}$.

5.33 Consider the problem of determining whether a PDA accepts some string of the form $\{ww |\ w \in \{\mathtt{0},\mathtt{1}\}^*\}$. Use the computation history method to show that this problem is undecidable.

5.34 Let $X = \{\langle M, w \rangle |\ M \text{ is a single-tape TM that never modifies the portion of the tape that contains the input } w\}$. Is X decidable? Prove your answer.

5.35 Say that a variable A in CFG G is ***necessary*** if it appears in every derivation of some string $w \in G$. Let $NECESSARY_{\mathsf{CFG}} = \{\langle G, A\rangle \mid A \text{ is a necessary variable in } G\}$.

a. Show that $NECESSARY_{\mathsf{CFG}}$ is Turing-recognizable.

b. Show that $NECESSARY_{\mathsf{CFG}}$ is undecidable.

***5.36** Say that a CFG is *minimal* if none of its rules can be removed without changing the language generated. Let $MIN_{\mathsf{CFG}} = \{\langle G\rangle \mid G \text{ is a minimal CFG}\}$.

a. Show that MIN_{CFG} is T-recognizable.

b. Show that MIN_{CFG} is undecidable.

SELECTED SOLUTIONS

5.5 Suppose for a contradiction that $A_{\mathsf{TM}} \leq_{\mathrm{m}} E_{\mathsf{TM}}$ via reduction f. It follows from the definition of mapping reducibility that $\overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} \overline{E_{\mathsf{TM}}}$ via the same reduction function f. However, $\overline{E_{\mathsf{TM}}}$ is Turing-recognizable (see the solution to Exercise 4.5) and $\overline{A_{\mathsf{TM}}}$ is not Turing-recognizable, contradicting Theorem 5.28.

5.6 Suppose $A \leq_{\mathrm{m}} B$ and $B \leq_{\mathrm{m}} C$. Then there are computable functions f and g such that $x \in A \iff f(x) \in B$ and $y \in B \iff g(y) \in C$. Consider the composition function $h(x) = g(f(x))$. We can build a TM that computes h as follows: First, simulate a TM for f (such a TM exists because we assumed that f is computable) on input x and call the output y. Then simulate a TM for g on y. The output is $h(x) = g(f(x))$. Therefore, h is a computable function. Moreover, $x \in A \iff h(x) \in C$. Hence $A \leq_{\mathrm{m}} C$ via the reduction function h.

5.7 Suppose that $A \leq_{\mathrm{m}} \overline{A}$. Then $\overline{A} \leq_{\mathrm{m}} A$ via the same mapping reduction. Because A is Turing-recognizable, Theorem 5.28 implies that $\overline{A}$ is Turing-recognizable, and then Theorem 4.22 implies that A is decidable.

5.8 You need to handle the case where the head is at the leftmost tape cell and attempts to move left. To do so, add dominos

$$\left[\frac{\#qa}{\#rb}\right]$$

for every $q, r \in Q$ and $a, b \in \Gamma$, where $\delta(q, a) = (r, b, \mathrm{L})$. Additionally, replace the first domino with

$$\left[\frac{\#}{\#\#q_0 w_1 w_2 \cdots w_n}\right]$$

to handle the case where the head attempts to move left in the very first move.

5.10 Let $B = \{\langle M, w\rangle | \ M$ is a two-tape TM that writes a nonblank symbol on its second tape when it is run on $w\}$. Show that A_{TM} reduces to B. Assume for the sake of contradiction that TM R decides B. Then construct a TM S that uses R to decide A_{TM}.

S = "On input $\langle M, w\rangle$:

1. Use M to construct the following two-tape TM T.
   T = "On input x:
   1. Simulate M on x using the first tape.
   2. If the simulation shows that M accepts, write a nonblank symbol on the second tape."
2. Run R on $\langle T, w\rangle$ to determine whether T on input w writes a nonblank symbol on its second tape.
3. If R accepts, M accepts w, so *accept*. Otherwise, *reject*."

5.11 Let $C = \{\langle M\rangle | \ M$ is a two-tape TM that writes a nonblank symbol on its second tape when it is run on some input$\}$. Show that A_{TM} reduces to C. Assume for the sake of contradiction that TM R decides C. Construct a TM S that uses R to decide A_{TM}.

S = "On input $\langle M, w\rangle$:

1. Use M and w to construct the following two-tape TM T_w.
   T_w = "On any input:
   1. Simulate M on w using the first tape.
   2. If the simulation shows that M accepts, write a nonblank symbol on the second tape."
2. Run R on $\langle T_w\rangle$ to determine whether T_w ever writes a nonblank symbol on its second tape.
3. If R accepts, M accepts w, so *accept*. Otherwise, *reject*."

5.28 Assume for the sake of contradiction that P is a decidable language satisfying the properties and let R_P be a TM that decides P. We show how to decide A_{TM} using R_P by constructing TM S. First, let $T_\emptyset$ be a TM that always rejects, so $L(T_\emptyset) = \emptyset$. You may assume that $\langle T_\emptyset\rangle \notin P$ without loss of generality because you could proceed with $\overline{P}$ instead of P if $\langle T_\emptyset\rangle \in P$. Because P is not trivial, there exists a TM T with $\langle T\rangle \in P$. Design S to decide A_{TM} using R_P's ability to distinguish between $T_\emptyset$ and T.

S = "On input $\langle M, w\rangle$:

1. Use M and w to construct the following TM M_w.
   M_w = "On input x:
   1. Simulate M on w. If it halts and rejects, *reject*. If it accepts, proceed to stage 2.
   2. Simulate T on x. If it accepts, *accept*."
2. Use TM R_P to determine whether $\langle M_w\rangle \in P$. If YES, *accept*. If NO, *reject*."

TM M_w simulates T if M accepts w. Hence $L(M_w)$ equals $L(T)$ if M accepts w and $\emptyset$ otherwise. Therefore, $\langle M_w\rangle \in P$ iff M accepts w.

5.30 **(a)** $INFINITE_{\mathsf{TM}}$ is a language of TM descriptions. It satisfies the two conditions of Rice's theorem. First, it is nontrivial because some TMs have infinite languages and others do not. Second, it depends only on the language. If two TMs recognize the same language, either both have descriptions in $INFINITE_{\mathsf{TM}}$ or neither do. Consequently, Rice's theorem implies that $INFINITE_{\mathsf{TM}}$ is undecidable.