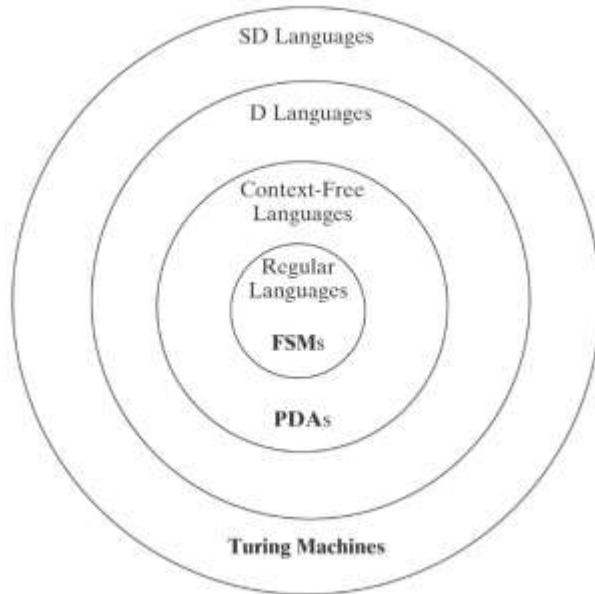


Part IV: Turing Machines and Undecidability

We are about to begin our exploration of the two outer circles of the language hierarchy, as well as the background, the area outside all of the circles.

Up until now, we have been placing limitations on what we could do in order that we could discover simple solutions when they exist.

Now we are going to tear down all the barriers and explore the full power of formal computation. We will discover a whole range of problems that become solvable once we do that. We will also discover that there are fundamental limitations on what we can compute, regardless of the specific model with which we choose to work.



17 Turing Machines

We need a new kind of automaton that has two properties:

- It must be powerful enough to describe all computable things. In this respect, it should be like real computers and unlike FSMs and PDAs.
- It must be simple enough that we can reason formally about it. In this respect, it should be like FSMs and PDAs and unlike real computers.

17.1 Definition, Notation and Examples

In our discussion of pushdown automata, it became clear that a finite state controller augmented by a single stack was not powerful enough to be able to execute even some very simple programs. What else must be added in order to acquire the necessary power? One answer is a second stack. We will explore that idea in Section 17.5.2.

17.1.1 What Is a Turing Machine?

A more straightforward approach is to eliminate the stack and replace it by a more flexible form of infinite storage, a writeable tape. When we do that, we get a Turing machine \square . Figure 17.1 shows a simple schematic diagram of a Turing machine M . M 's tape is infinite in both directions. The input to M is written on the tape, one character per square, before M is started. All other squares of the tape are initially blank (\square). As we have done for both FSMs and PDAs, M 's behavior will be defined only on input strings that are:

- finite, and
- contain only characters in M 's input alphabet.

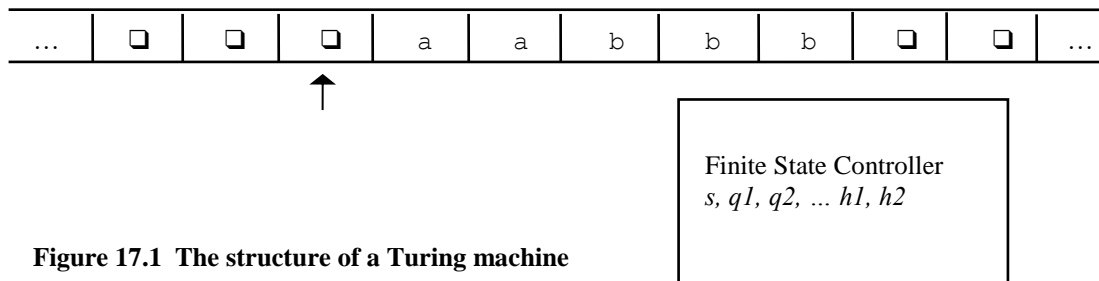


Figure 17.1 The structure of a Turing machine

M has a single read/write head, shown here with an arrow. We will almost always use the convention that, when M starts, its read/write head will be over the blank immediately to the left of the leftmost character of the input. But occasionally, when we are designing a machine to be used as a subroutine by some other machine, we may choose a different initial specification.

M begins in its start state. At each step of its operation, M must:

- choose its next state,
- write on the current square, and
- move the read/write head left or right one square.

M can move back and forth on its tape, so there is no longer the idea that it consumes all of its input characters one at a time and then halts. M will continue to execute until it reaches a special state called a halting state. It is possible that M may never reach a halting state, in which case it will execute forever.

Notice that there will always be a finite number of nonblank squares on M 's tape. This follows from the fact that, before M starts, only a finite number of squares are nonblank. Then, at each step of its operation, M can write on at most one additional square. So, after any finite number of steps, only a finite number of squares can be nonblank. And, even if M never halts, at any point in its computation it will have executed only a finite number of steps.

In Chapter 18 we are going to argue that the Turing machine, as we have just described it, is as powerful as any other reasonable model of computation, including modern computers. So, in the rest of this discussion, although our examples will be simple, remember that we are now talking about computation, broadly conceived.

We are now ready to provide a formal definition: a **Turing machine** (or **TM**) M is a sextuple $(K, \Sigma, \Gamma, \delta, s, H)$:

- K is a finite set of states,
- Σ is the input alphabet, which does not contain \square ,
- Γ is the tape alphabet, which must, at a minimum, contain \square and have Σ as a subset,
- $s \in K$ is the start state,
- $H \subseteq K$ is the set of halting states, and
- δ is the transition function. It maps from:

$$\begin{array}{ccccccc} (K - H) & \times & \Gamma & \text{to} & K & \times & \Gamma & \times & \{\rightarrow, \leftarrow\}. \\ \text{non-halting state} & \times & \text{tape character} & & \text{state} & \times & \text{tape character} & \times & \text{action (R or L)} \end{array}$$

If δ contains the transition $((q_0, a), (q_1, b, A))$ then, whenever M is in state q_0 and the character under the read/write head is a , M will go to state q_1 , write b , and then move the read/write head as specified by A (either one square to the right or one square to the left).

Notice that the tape symbol \square is special in two ways:

- Initially, all tape squares except those that contain the input string contain \square .
- The input string may not contain \square .

But those are the only ways in which \square is special. A Turing machine may write \square just as it writes any other symbol in its tape alphabet Γ . Be careful, though, if you design a Turing machine M that does write \square . Make sure that M can tell the difference between running off the end of the input and hitting a patch of \square 's within the part of the tape it is working on. Some books use a definition of a Turing machine that does not allow writing \square . But we allow it because it can be quite useful if you are careful. In addition, this definition allows a Turing machine to output a string that is shorter than its input by writing \square 's as necessary.

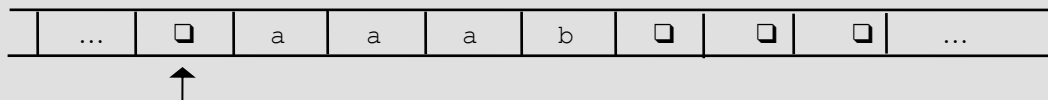
Define the **active tape** of a Turing machine M to be the shortest fragment of M 's tape that includes the square under the read/write head and all the nonblank squares.

We require that δ be defined for all (state, input) pairs unless the state is a halting state. Notice that δ is a function, not a relation. So this is a definition for deterministic Turing machines.

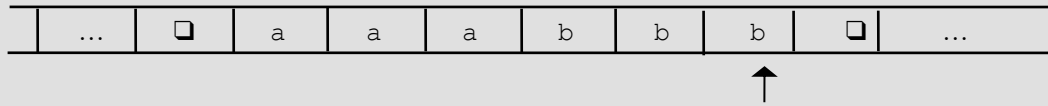
One other important observation: a Turing machine can produce output, namely the contents of its tape when it halts.

Example 17.1 Add b's to Make Them Match the a's

Design a Turing machine M that takes as input a string in the language $\{a^i b^j : 0 \leq j \leq i\}$ and adds b's as required to make the number of b's equal the number of a's. The input to M will look like this:



On that input, the output (the contents of the tape when M halts) should be:



M will operate as follows:

1. Move one square to the right. If the character under the read/write head is □, halt. Otherwise, continue.
2. Loop:
 - 2.1 Mark off an a with a \$.
 - 2.2 Scan rightward to the first b or □.
 - If b, mark it off with a # and get ready to go back and find the next matching a, b pair.
 - If □, then there are no more b's but there are still a's that need matches. So it is necessary to write another b on the tape. But that b must be marked so that it cannot match another a. So write a #. Then get ready to go back and look for remaining unmarked a's.
 - 2.3 Scan back leftward looking for a or □. If a, then go back to the top of the loop and repeat. If □, then all a's have been handled. Exit the loop. (Notice that the specification for M guarantees that there will not be more b's than a's.)
3. Make one last pass all the way through the nonblank area of the tape, from left to right, changing each \$ to an a and each # to a b.
4. Halt.

$M = \{ \{1, 2, 3, 4, 5, 6\}, \{a, b\}, \{a, b, \square, \$, \#\}, \delta, 1, \{6\} \}$, where $\delta =$

- | | |
|--|---|
| (((1, □), (2, □, →)),
((1, a), (2, □, →)),
((1, b), (2, □, →)),
((1, \$), (2, □, →)),
((1, #), (2, □, →)),
((2, □), (6, \$, →)),
((2, a), (3, \$, →)),
((2, b), (3, \$, →)),
((2, \$), (3, \$, →)),
((2, #), (3, \$, →)),
((3, □), (4, #, ←)),
((3, a), (3, a, →)),
((3, b), (4, #, ←)),
((3, \$), (3, \$, →)),
((3, #), (3, #, →)),
((4, □), (5, □, →)),
((4, a), (3, \$, →)),
((4, \$), (4, \$, ←)),
((4, #), (4, #, ←)),
((5, □), (6, □, ←)),
((5, \$), (5, a, →)),
((5, #), (5, b, →))) | {These four transitions are required because M must be defined for every state/ input pair, but since it isn't possible to see anything except □ in state 1, it doesn't matter what they do. }

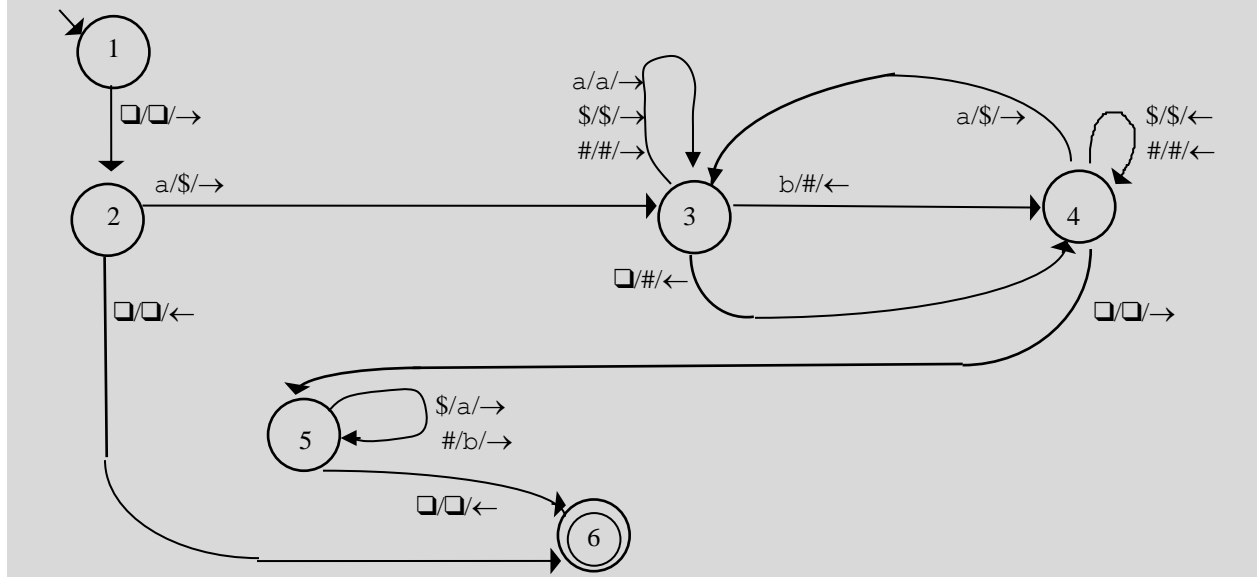
{Three more unusable elements of δ . We'll omit the rest here for clarity.}

{State 6 is a halting state and so has no transitions out of it.} |
|--|---|

People find it nearly impossible to read transition tables like this one, even for very simple machines. So we will adopt a graphical notation similar to the one we used for both FSMs and PDAs. Since each element of δ has five components, we need a notation for labeling arcs that includes all the required information. Let $x/t/a$ on an arc of M mean that the transition can be taken if the character currently under the read/write head is x . If it is taken, write t and then move the read/write head as specified by a . We will also adopt the convention that we will omit unusable transitions, such as $((1, a), (2, \square, \rightarrow))$ in the example above, from our diagrams so that they are easier to read.

Example 17.2 Using the Graphical Language

Here is a graphical description, using the notation we just described, of the machine from Example 17.1:



17.1.2 Programming Turing Machines

Although there is a lot less practical motivation for learning to program a Turing machine than there is for learning to build FSMs, regular expressions, context-free grammars, and parsers, it is interesting to see how a device that is so simple can actually be made to compute whatever we can compute using the fastest machines in our labs today. It seems to highlight the essence of what it takes to compute.

In Chapter 18, we will argue that anything computable can be computed by a Turing machine. So we should not expect to find simple nuggets that capture everything a Turing machine programmer needs to know. But, at least for the fairly straightforward language recognition problems that we will focus on, there are a few common programming idioms. The example we have just shown illustrates them:

- A computation will typically occur in phases: when phase 1 finishes, phase 2 begins, and so forth.
- One phase checks for corresponding substrings by moving back and forth, marking off corresponding characters.
- There are two common ways to go back and forth. Suppose the input string is `aaaabbbb` and we want to mark off the `a`'s and make sure they have corresponding `b`'s. Almost any sensible procedure marks the first `a`, scans right, and marks the first `b`. There are then two ways to approach doing the rest. We could:
 - Scan left to the first `a` we find and process the rest of the `a`'s right to left. That is the approach we took in Example 17.1 above.
 - Scan all the way left until we find the first marked `a`. Bounce back one square to the right and mark the next `a`. In this approach, we process all the `a`'s left to right.
 Both ways work. Sometimes it seems easier to use one, sometimes the other.
- If we care about the machine's output (as opposed to caring just about whether it accepts or rejects), then there is a final phase that makes one last pass over the tape and converts the marked characters back to their proper form.

17.1.3 Halting

We make the following important observations about the three kinds of automata that we have so far considered:

- A DFSM M , on input w , is guaranteed to halt in $|w|$ steps. We proved this result as Theorem 5.1. An arbitrary NDFSM can be simulated by *ndfmsimulate* and that simulation will also halt in $|w|$ steps.

- An arbitrary PDA, on input w , is not guaranteed to halt. But, as we saw in Chapter 14, for any context-free language L there exists a PDA M that accepts L and that is guaranteed to halt.
- A Turing machine M , on input w , is not guaranteed to halt. It could, instead, bounce back and forth forever on its tape. Or it could just blast its way, in a single direction, through the input and off forever into the infinite sequence of blanks on the tape. And now, unlike with PDAs, there exists no algorithm to find an equivalent Turing machine that is guaranteed to halt.

This fundamental property of Turing machines, that they cannot be guaranteed to halt, will drive a good deal of our discussion about them.

17.1.4 Formalizing the Operation of a Turing Machine

In this section we will describe formally the computation process that we outlined in the last section.

A **configuration** of a Turing machine $M = (K, \Sigma, \Gamma, \delta, s, H)$ is a 4-tuple that is an element of:

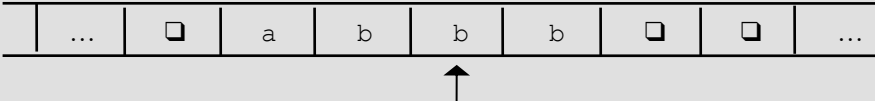
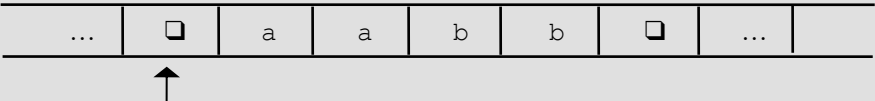
K	×	$(\Gamma - \{\square\}) \Gamma^* \cup \{\varepsilon\}$	×	Γ	×	$(\Gamma^* (\Gamma - \{\square\})) \cup \{\varepsilon\}$.
state		includes all of M 's active tape to the left of the read/write head		square under the read/write head		includes all of M 's active tape to the right of the read/write head

Notice that, although M 's tape is infinite, the description of any configuration is finite because we include in that description the smallest contiguous tape fragment that includes all the nonblank squares and the square under the read/write head.

We will use the following shorthand for configurations: (q, s_1, a, s_2) will be written as $(q, s_1 \underline{a} s_2)$.

The initial configuration of any Turing machine M with start state s and input w is $(s, \square w)$. Any configuration whose state is an element of H is a halting configuration.

Example 17.3 Using the 4-Tuple Notation and the Shorthand

	As a 4-tuple	Shorthand
	(q, ab, b, b)	$(q, ab\underline{b}b)$
	$(q, \varepsilon, \square, aabb)$	$(q, \underline{\square}aabb)$

The transition function δ defines the operation of a Turing machine M one step at a time. We can use it to define the sequence of configurations that M will enter. We start by defining the relation *yields-in-one-step*, written \vdash_M , which relates configuration c_1 to configuration c_2 iff M can move from configuration c_1 to configuration c_2 in one step. So, just as we did with FSMs and PDAs, we define:

$$(q_1, w_1) \vdash_M (q_2, w_2) \text{ iff } (q_2, w_2) \text{ is derivable, via } \delta, \text{ in one step.}$$

We can now define the relation *yields*, written \vdash_M^* , to be the reflexive, transitive closure of \vdash_M . So configuration C_1 yields configuration C_2 iff:

$$C_1 \vdash_M^* C_2.$$

A **path** through M is a sequence of configurations C_0, C_1, C_2, \dots such that C_0 is an initial configuration of M and:

$$C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \dots$$

A **computation** by M is a path that halts. So it is a sequence of configurations C_0, C_1, \dots, C_n for some $n \geq 0$, such that C_0 is an initial configuration of M , C_n is a halting configuration of M , and:

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

If a computation halts in n steps, we will say that it has length n and we will write:

$$C_0 \vdash_{M^n} C_n$$

17.1.5 A Macro Notation for Turing Machines

Writing even very simple Turing machines is time consuming and reading a description of one and making sense of it is even harder. Sometimes we will simply describe, at a high level, how a Turing machine should operate. But there are times when we would like to be able to specify a machine precisely. So, in this section, we present a macro language that will make the task somewhat easier. If you don't care about the details of how Turing machines work, you can skip this section. In most of the rest of our examples, we will give the high level description first, followed (when it's feasible) by a description in this macro language.

The key idea behind this language is the observation that we can combine smaller Turing machines to build more complex ones. We begin by defining simple machines that perform the basic operations of writing on the tape, moving the read/write head, and halting:

- Symbol writing machines: for each $x \in \Gamma$, define M_x , written just x , to be a Turing machine that writes x on the current square of the tape and then halts. So, if $\Gamma = \{a, b, \square\}$, there will be three simple machines: a , b , and \square . (A technical note: Given our definition of a Turing machine, each of these machines must actually make two moves. In the first move, it writes the new symbol on the tape and moves right. In the next move, it rewrites whatever character was there and then moves left. These two moves are necessary because our machines must move at each step. But this is a detail with which we do not want to be concerned when we are writing Turing machine programs. This notation hides it from us.)
- Head moving machines: there are two of these: R rewrites whatever character was on the tape and moves one square to the right. L rewrites whatever character was on the tape and moves one square to the left.
- Machines that simply halt: each of our machines halts when it has nothing further to do (i.e., it has entered a state on which δ is undefined), but there are times when we'll need to indicate halting explicitly. We will use three simple halting machines:
 - h , which simply halts. We will use h when we want to make it clear that some path of a machine halts, but we do not care about accepting or rejecting.
 - n , which halts and rejects.
 - y , which halts and accepts.

Next we need to describe how to:

- Check the tape and branch based on what character we see, and
- Combine the basic machines to form larger ones.

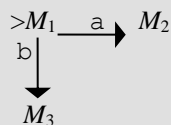
We can do both of these things with a notation that is very similar to the one we have used for all of our state machines so far. We will use two basic forms:

- $M_1 M_2$: Begin in the start state of M_1 . Run M_1 until it halts. If it does, begin M_2 in its start state (without moving the read/write head) and run M_2 until it halts. If it does, then halt. If either M_1 or M_2 fails to halt, $M_1 M_2$ will fail to halt.
- $M_1 \xrightarrow{\text{condition}} M_2$: Begin in the start state of M_1 . Run M_1 until it halts. If it does, check *condition*. If it is true, then begin M_2 in its start state (without moving the read/write head) and run M_2 until it halts. The simplest condition will be the presence of a specific character under the read/write head, although we will introduce some others as well. A machine with this structure will fail to halt if either:
 - M_1 fails to halt, or
 - *condition* is true and M_2 fails to halt.

We will use the symbol $>$ to indicate where the combination machine begins.

Example 17.4 The Macro Language Lets Machines be Composed

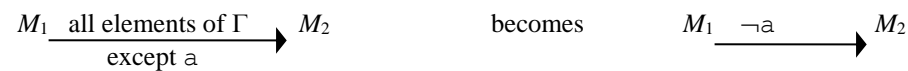
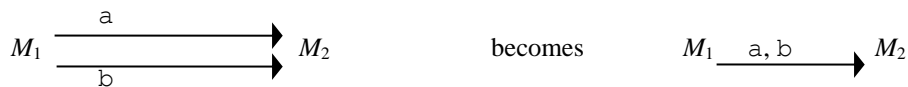
Let $M =$



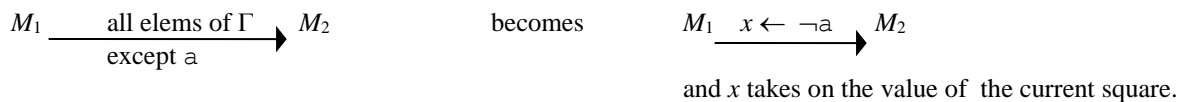
So M :

- Starts in the start state of M_1 .
- Computes until M_1 reaches a halting state.
- Examines the tape. If the current symbol is a , then it transfers control to M_2 . If the current symbol is b , it transfers control to M_3 .

To make writing our machines a bit easier, we introduce some shorthands:



Next we provide a simple mechanism for storing values in variables. Each variable will hold just a single character. A standard Turing machine can remember values for any finite number of such variables either by writing them someplace on its tape or by branching to a different state for each possible value. This second solution avoids having to scan back and forth on the tape, but it can lead to an explosion in the number of states since there must be effectively a new copy of the machine for each combination of values that a set of variables can have. We will hide the mechanism by which variables are implemented by allowing them to be named and explicitly referenced in the conditions on the arcs of our machines. So we have:



We can use the value of a variable in two ways. The first is as a condition on a transition. So we can write:

$$M_1 \xrightarrow{x=y} M_2$$

if $x = y$ then take the transition.

Note that we use \leftarrow for assignment and $=$ and \neq for Boolean comparison.

We can also write the value of a variable. We'll indicate that with the variable's name.

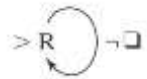
Example 17.5 Using Variables to Remember Single Characters

Let $M =$

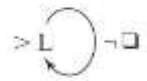
$$\triangleright \xrightarrow{x \leftarrow \neg \square} R_x$$

If the current square is not blank, M remembers its value in the variable x , goes right one square, and copies it by writing the value of x . (If the current square is blank, M has nothing to do. So it halts.)

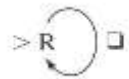
Next we define some very useful machines that we can build from the primitives we have so far:



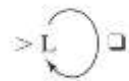
Move right. If the character under the read/write head is not \square , repeat. If it is \square , no further action is specified, so halt. In other words, find the first blank square to the right of the current square. We will abbreviate this R_{\square} .



Move left. If the character under the read/write head is not \square , repeat. If it is \square , no further action is specified, so halt. In other words, find the first blank square to the left of the current square. We will abbreviate this L_{\square} .



Similarly, but find the first nonblank square to the right of the current square. We will abbreviate this $R_{\neg \square}$.



Similarly, but find the first nonblank square to the left of the current square. We will abbreviate this $L_{\neg \square}$.

We can do the same thing we have just done for \square with any other character in Γ . So we can write:

L_a Find the first occurrence of a to the left of the current square.

$R_{a,b}$ Find the first occurrence of a or b to the right of the current square.

$L_{a,b} \xrightarrow{a} M_1$ Find the first occurrence of a or b to the left of the current square, then go to M_1 if the detected character is a ; go to M_2 if the detected character is b .

\downarrow

M_2

$L_{x \leftarrow a,b}$ Find the first occurrence of a or b to the left of the current square and set x to the value found.

$L_{x \leftarrow a,b} R_x$ Find the first occurrence of a or b to the left of the current square, set x to the value found, move one square to the right, and write x (a or b).

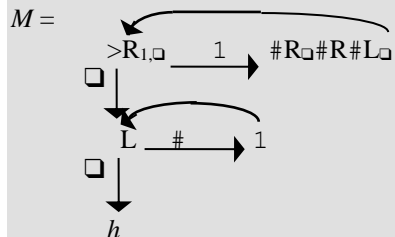
Example 17.6 Triplicating a String

We wish to build M with the following specification: Input: $\square w$ $w \in \{1\}^*$
 Output: $\square w^3$

Example: Input: $\square 111$ Output: $\square 111111111$

M will operate as follows on input w :

1. Loop
 - 1.1 Move right to the first 1 or \square .
 - 1.2 If the current character is \square , all 1s have been copied. Exit the loop. Else, the current character must be a 1. Mark it off with # (so it won't get copied again), move right to the first blank, and write two more #'s.
 - 1.3 Go left back to the blank in front of the string.
2. Make one final pass through the string converting the #'s back to 1's.



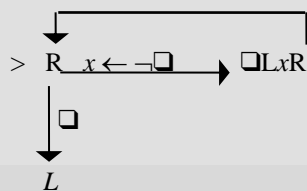
Example 17.7 Shifting Left One Square

We wish to build a shifting machine S_{\leftarrow} with the following specification, where u and w are strings that do not contain any \square 's:

Input: $\square u \square w \square$
 Output: $\square u w \square$

Example: Input: $11 \square 00$
 Output: $1100 \square$

S_{\leftarrow} moves left to right through w , copying each character onto the square immediately to its left:



17.2 Computing With Turing Machines

Now that we know how Turing machines work, we can describe how to use a Turing machine to:

- recognize a language, or
- compute a function.

17.2.1 Turing Machines as Language Recognizers

Given a language L , we would like to be able to design a Turing machine M that takes as input (on its tape) some string w and tells us whether or not $w \in L$. There are many languages for which it is going to be possible to do this. Among these are all of the noncontext-free languages that we discussed in Part III (as well as all the regular and context-free languages for which we have built FSMs and PDAs).

However, as we will see in Chapter 19 and others that follow it, there are many languages for which even the power of the Turing machine is not enough. In some of those cases, there is absolutely nothing better that we can do. There exists no Turing machine that can distinguish between strings that are in L and strings that are not. But there are other languages for which we can solve part of the problem. For each of these languages we can build a Turing machine M that looks for the property P (whatever it is) of being in L . If M discovers that its input possesses P , it halts and accepts. But if P does not hold for some input string w , then M may keep looking forever. It may not be able to tell that it is not going to find P and thus it should halt and reject.

In this section we will define what it means for a Turing machine to decide a language (i.e., for every string, accept or reject as appropriate) and for a Turing machine to semidecide a language (i.e., to accept when it should).

Deciding a Language

Let M be a Turing machine with start state s and two halting states that we will call y and n . Let w be an element of Σ^* . Then we will say that:

- M **accepts** w iff $(s, \sqcup w) \vdash_{-M^*} (y, w')$ for some string w' . We call any configuration (y, w') an **accepting configuration**.
- M **rejects** w iff $(s, \sqcup w) \vdash_{-M^*} (n, w')$ for some string w' . We call any configuration (n, w') a **rejecting configuration**.

Notice that we do not care what the contents of M 's tape are when it halts. Also note that if M does not halt, it neither accepts nor rejects.

Let Σ be the input alphabet of M . Then M **decides** a language $L \subseteq \Sigma^*$ iff, for any string $w \in \Sigma^*$, it is true that:

- If $w \in L$ then M accepts w , and
- If $w \notin L$ then M rejects w .

Since every string in Σ^* is either in L or not in L , any deciding machine M must halt on all inputs. A language L is **decidable** iff there is a Turing machine M that decides it.

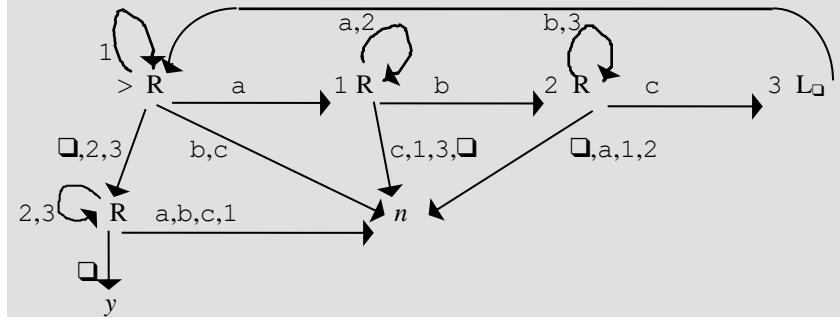
We define the set D to be the set of all decidable languages. So a language L is in D iff there is a Turing machine that decides it. In some books, the set D is called R , or the set of **recursive languages**.

Example 17.8 $A^n B^n C^n$

Recall the language $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$, which we showed was not context-free and so could not be recognized with a PDA. $A^n B^n C^n$ is decidable. We can build a straightforward Turing machine M to decide it. M will work as follows on input w :

1. Move right onto w . If the first character is \sqcup , halt and accept.
2. Loop:
 - 2.1. Mark off an a with a 1.
 - 2.2. Move right to the first b and mark it off with a 2. If there isn't one, or if there is a c first, halt and reject.
 - 2.3. Move right to the first c and mark it off with a 3. If there isn't one, or if there is an a first, halt and reject.
 - 2.4. Move all the way back to the left, then right again past all the 1's (the marked off a 's). If there is another a , go back to the top of the loop. If there isn't, exit the loop.
3. All a 's have found matching b 's and c 's and the read/write head is just to the right of the region of marked off a 's. Continue moving left to right to verify that all b 's and c 's have been marked. If they have, halt and accept. Otherwise halt and reject.

In our macro language, M is:

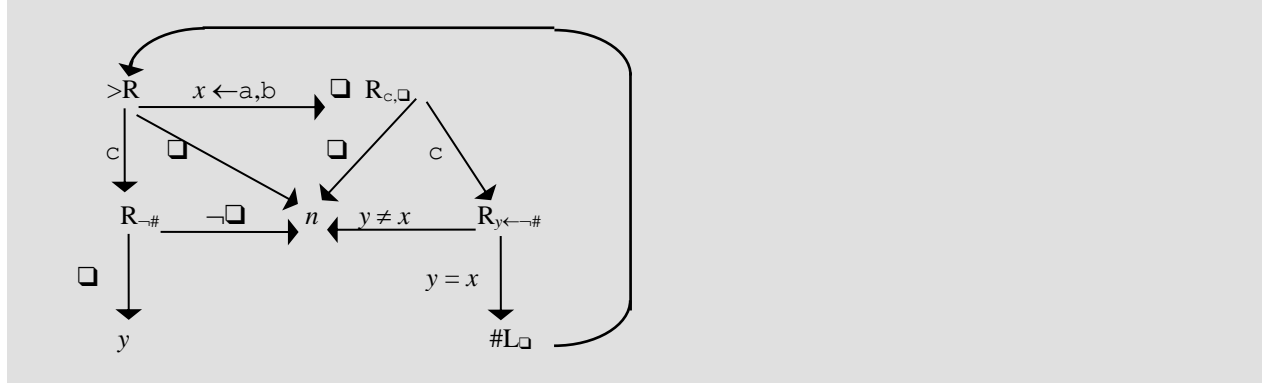


Example 17.9 WcW

Consider again $WcW = \{wcw : w \in \{a, b\}^*\}$. We can build M to decide WcW as follows:

1. Loop:
 - 1.1. Move right to the first character. If it is c , exit the loop. Otherwise, overwrite it with \square and remember what it is.
 - 1.2. Move right to the c . Then continue right to the first unmarked character. If it is \square , halt and reject. (This will happen if the string to the right of c is shorter than the string to the left.) If it is anything else, check to see whether it matches the remembered character from the previous step. If it does not, halt and reject. If it does, mark it off with $\#$.
 - 1.3. Move back leftward to the first \square .
2. There are no characters remaining before the c . Make one last sweep left to right checking that there are no unmarked characters after the c and before the first blank. If there are, halt and reject. Otherwise, halt and accept.

In our macro language, M is;



Semideciding a Language

Let Σ be the input alphabet to a Turing machine M . Let $L \subseteq \Sigma^*$. Then we will say that M *semidecides* L iff, for any string $w \in \Sigma^*$:

- If $w \in L$ then M accepts w , and
- If $w \notin L$ then M does not accept w . In this case, M may explicitly reject or it may loop.

A language L is *semidecidable* iff there is a Turing machine that semidecides it. We define the set SD to be the set of all semidecidable languages. So a language L is in SD iff there is a Turing machine that semidecides it. In some books, the set SD is called RE , or the set of *recursively enumerable languages* or the set of *Turing-recognizable languages*.

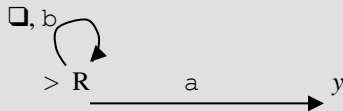
Example 17.10 Semideciding by Running Off the Tape

Let $L = b^*a (a \cup b)^*$. So, any machine that accepts L must look for at least one a .

We can build M to semidecide L :

1. Loop:
Move one square to the right. If the character under the read/write head is an a , halt and accept.

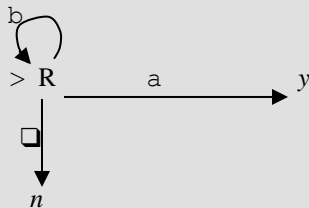
In our macro language, M is:



Of course, for L , we can do better than M . $M\#$ decides L :

1. Loop:
Move one square to the right. If the character under the read/write head is an a , halt and accept. If it is \square , halt and reject.

In our macro language, $M\#$ is:



As we will prove later, there are languages that are in SD but not D and so a semideciding Turing machine is the best we will be able to build for those languages.

17.2.2 Turing Machines Compute Functions

When a Turing machine halts, there is a value on its tape. When we build deciding and semideciding Turing machines, we ignore that value. But we don't have to. Instead, we can define what it means for a Turing machine to compute a function. We'll begin by defining what it means for a Turing machine to compute a function whose domain and range are sets of strings. Then we'll see that, by using appropriate encodings of other data types and of multiple input values, we can define Turing machines to compute a wide variety of functions.

In this section, we consider only Turing machines that always halt. In Chapter 25 we will expand this discussion to include Turing machines that sometimes fail to halt.

Let M be a Turing machine with start state s , halting state h , and input alphabet Σ . The initial configuration of M will be $(s, \square w)$, where $w \in \Sigma^*$.

Define $M(w) = z$ iff $(s, \square w) \vdash_{-M}^* (h, \square z)$. In other words $M(w) = z$ iff M , when started on a string w in Σ^* , halts with z on its tape and its read/write head is just to the left of z .

Let $\Sigma' \subseteq \Gamma$ be M 's output alphabet (i.e., the set of symbols that M may leave on its tape when it halts).

Now, let f be any function that maps from Σ^* to Σ'^* . We say that a Turing machine M **computes** a function f iff, for all $w \in \Sigma^*$:

- If w is an input on which f is defined, $M(w) = f(w)$. In other words, M halts with $f(w)$ on its tape.
- Otherwise $M(w)$ does not halt.

A function f is **recursive** or **computable** iff there is a Turing machine M that computes it and that always halts. The term *computable* more clearly describes the essence of these functions. The traditional name for them, however, is *recursive*. We will see why that is in Chapter 25. In the meantime, we will use the term *computable*.⁷

There is a natural correspondence between the use of Turing machines to compute functions and their use as language deciders. A language is decidable iff its characteristic function is computable. In other words, a language L is decidable iff there exists a Turing machine that always halts and that outputs *True* if its input is in L and *False* otherwise.

Example 17.11 Duplicating a String

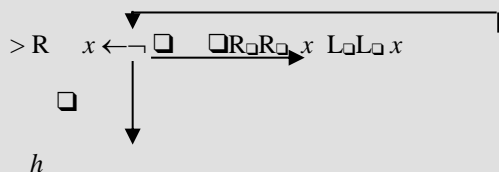
Let $duplicate(w) = ww$, where w is a string that does not contain \square .

A Turing machine to compute $duplicate$ can be built easily if we have two subroutines:

- The copy machine C , which will perform the following operation:

$\square w \square \square \square \square \square \square \rightarrow \square w \square w \square$

C will work by moving back and forth on the tape, copying characters one at a time:



We define C this way because the copy process is straightforward if there is a character (we use \square) to delimit the two copies.

- The S_{\leftarrow} machine, which we described in Section 17.1.5. We will use S_{\leftarrow} to shift the second copy of w one square to the left.

M , defined as follows, computes $duplicate$:

$> C S_{\leftarrow} L \square$

Now suppose that we want to compute functions on values other than strings. All we need to do is to encode those values as strings. To make it easy to describe functions on such values, define a family of functions, $value_k(n)$. For any positive integer k , $value_k(n)$ returns the nonnegative integer that is encoded, base k , by the string n . For example, $value_2(101) = 5$ and $value_8(101) = 65$. We will say that a Turing machine M computes a function f from \mathbb{N}^m to \mathbb{N} provided that, for some k , $value_k(M(n_1;n_2;\dots;n_m)) = f(value_k(n_1), \dots, value_k(n_m))$.

Not all functions with straightforward definitions are computable. For example, the busy beaver functions described in Section 25.1.4 measure the “productivity” of Turing machines by returning

⁷ In some other treatments of this subject, a function f is **computable** iff there is some Turing machine M (which may not always halt) that computes it. Specifically, if there are values for which f is undefined, M will fail to halt on those values. We will say that such a function is **partially computable** and we will reserve the term *computable* for that subset of the partially computable functions that can be computed by a Turing machine that always halts.

the maximum amount of work (measured in steps or in number of symbols on the tape) that can be done by a Turing machine with n states. The busy beaver functions are not computable.

Example 17.12 The Successor Function

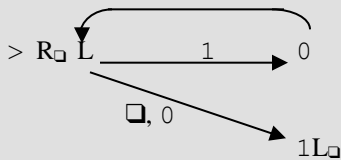
Consider the successor function $succ(n) = n + 1$. On input $\square^n \square \square \square \square \square$, M should output $\square^{n+1} \square$.

We will represent n in binary without leading zeros. So $n \in 0 \cup 1\{0,1\}^*$ and $f(n) = m$, where $value_2(m) = value_2(n) + 1$.

We can now define the Turing machine M to compute $succ$:

1. Scan right until the first \square . Then move one square back left so that the read/write head is on the last digit of n .
2. Loop:
 - 2.1. If the digit under the read/write head is a 0, write a 1, move the read/write head left to the first blank, and halt.
 - 2.2. If the digit under the read/write head is a 1, we need to carry. So write a 0, move one square to the left, and go back to the top of the loop.
 - 2.3. If the digit under the read/write head is a \square , we have carried all the way to the left. Write a 1, move one square to the left, and halt.

In our macro language, M is:



We can build Turing machines to compute functions of two or more arguments by encoding each of the arguments as a string and then concatenating them together, separated by a delimiter.

Example 17.13 Binary Addition

Consider the *plus* function defined on the integers. On input $\square^x; y \square \square \square \square \square$, M should output the sum of x and y .

We will represent x and y in binary without leading zeros. So, for example, we'll encode the problem $5 + 8$ as the input string $101; 1000$. On this input, M should halt with 1101 on its tape. More generally, M should compute $f(n_1, n_2) = m$, where $value_2(m) = value_2(n_1) + value_2(n_2)$.

We leave the design of M as an exercise.

17.3 Adding Multiple Tapes and Nondeterminism

We have started with a very simple definition of a Turing machine. In this section we will consider two important extensions to that basic model. Our goal in describing the extensions is to make Turing machines easier to program. But we don't want to do that if it forces us to give up the simple model that we carefully chose because it would be easy to prove things about. So we are not going to add any fundamental power to the model. For each of the extensions we consider, we will prove that, given a Turing machine M that exploits the extension, there exists a Turing machine M' that is equivalent to M and that does not exploit the new feature. Each of these proofs will be by construction, from M to M' . This will enable us to place a bound on any change in time complexity that occurs when we transform M to M' .

There will be a bottom line at the end of this chapter. The details of the definition of a Turing machine don't matter in the sense that they don't affect what can be computed. In fact, there is a large family of other computational models

that look even more unlike the basic definition than our extended machines do but that are still equivalent in power. We will articulate that principle in the following chapter. We will see, however, that the details may matter if we are concerned about the efficiency of the computations that we do. Even here, though, the details matter less than one might initially think. With one exception (the addition of nondeterminism), we'll see that adding features changes the time complexity of the resulting programs by at most a polynomial factor.

17.3.1 Multiple Tapes

The first extension that we will propose is additional tapes. Suppose we could build a Turing machine with two or three or more tapes, each with its own read/write head, as shown in Figure 17.2. What could we do with such a machine? One answer is, “a lot less going back and forth on the tape.”

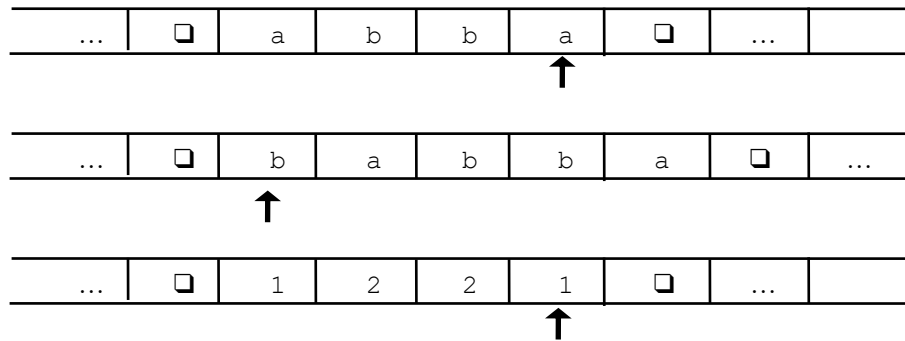


Figure 17.2 A multiple tape Turing machine

A k -tape Turing machine, just like a 1-tape Turing machine, is a sextuple $M = (K, \Sigma, \Gamma, \delta, s, H)$. A configuration of a k -tape machine M is a $k+1$ tuple: $(state, tape_1, \dots, tape_k)$, where each tape description is identical to the description we gave in Section 17.1.4 for a 1-tape machine. M 's initial configuration will be $(s, \square_w, \square, \dots, \square)$. In other words, its input will be on tape 1; all other tapes will initially be blank, with their read/write heads positioned on some blank square. If M halts, we will define its output to be the contents of tape 1; the contents of the other tapes will be ignored.

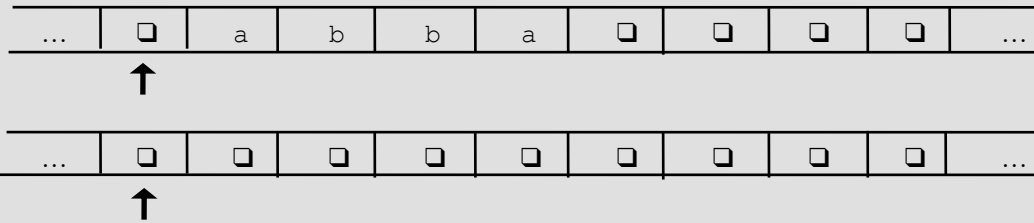
At each step, M will examine the square under each of its read/write heads. The set of values so obtained, along with the current state, determines M 's next action. It will write and then move on each of the tapes simultaneously. Sometimes M will want to move along one or more of its tapes without moving on others. So we will now allow the move action, stay put, which we will write as \uparrow . So δ is a function from:

$$\begin{array}{l}
 ((K-H) \times \Gamma_1 \quad \text{to} \quad (K \times \Gamma_1 \times \{\leftarrow, \rightarrow, \uparrow\} \\
 \times \Gamma_2 \quad \times \Gamma_2 \times \{\leftarrow, \rightarrow, \uparrow\} \\
 \times \dots \quad \times \dots \\
 \times \dots \quad \times \dots \\
 \times \Gamma_k) \quad \times \Gamma_k \times \{\leftarrow, \rightarrow, \uparrow\}).
 \end{array}$$

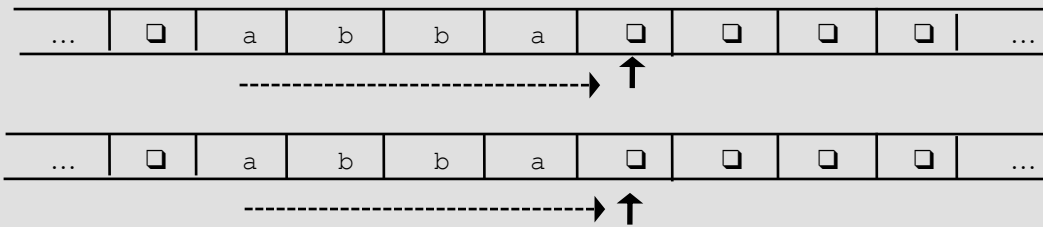
Example 17.14 Exploiting Two Tapes to Duplicate a String

Suppose that we want to build a Turing machine that, on input $\square_w\square$, outputs $\square_ww\square$. In Example 17.11 we saw how we could do this with a conventional, one-tape machine that went back and forth copying each character of w one at a time. To copy a string of length n took n passes, each of which took n steps, for a total of n^2 steps. But to make that process straightforward, we left a blank between the two copies. So then we had to do a second pass in which we shifted the copy one square to the left. That took an additional n steps. So the entire process took $\mathcal{O}(n^2)$ steps. We now show how to do the same thing with a two tape machine M_C in $\mathcal{O}(n)$ steps.

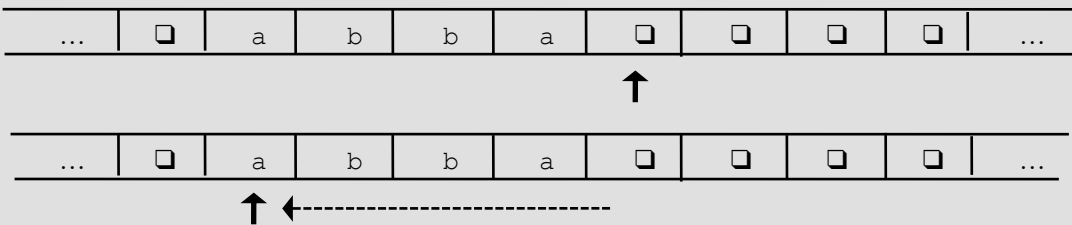
Let w be the string to be copied. Initially, w is on tape 1 with the read/write head just to its left. The second tape is empty. The operation of M_C is shown in the following series of snapshots:



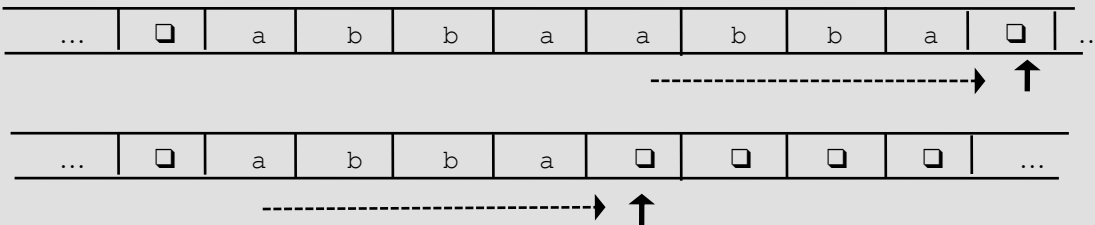
The first thing M_C will do is to move to the right on both tapes, one square at a time, copying the character from tape 1 onto the corresponding square of tape 2. This phase of the processing takes $|w|$ steps. At the end of this phase, the tapes will look like this, with both read/write heads on the blank just to the right of w :



Next M_C moves tape 2's read/write head all the way back to the left. This phase also takes $|w|$ steps. At the end of it, the tapes will look like this:



In its final phase, M_C will sweep to the right, copying w from tape 2 to tape 1. This phase also takes $|w|$ steps. At the end of it, the tapes will look like this:

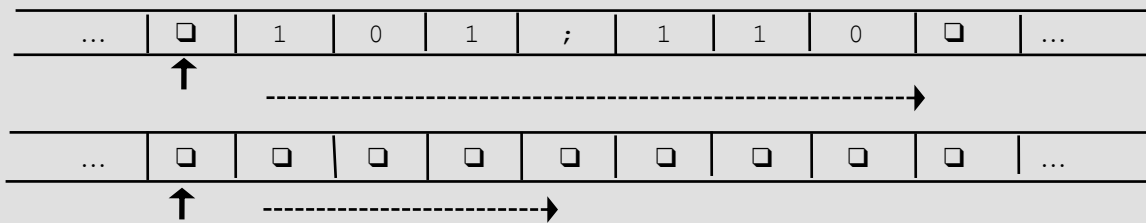


M_C takes $3 \cdot |w| = \mathcal{O}(|w|)$ steps.

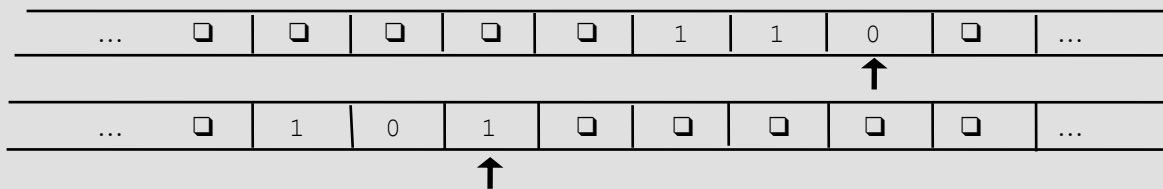
Example 17.15 Exploiting Two Tapes for Addition

Exercise 17.3)a) asked you to construct a standard one-tape Turing machine to add two binary numbers. Let's now build a 2-tape Turing machine M_A to do that. Let x and y be arbitrary binary strings. On input $\sqcup x; y$, M_A should output $\sqcup z$, where z is the binary encoding of the sum of the numbers represented by x and y .

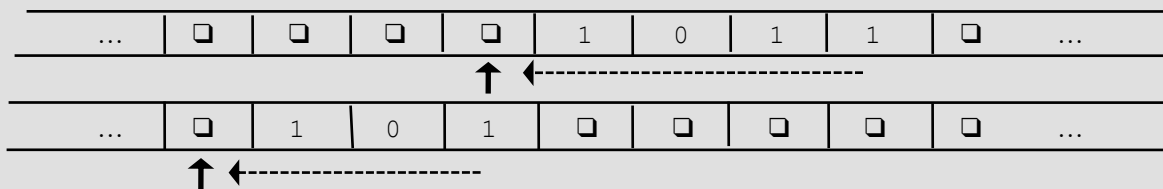
For example, let $x = 5$ and $y = 6$. The initial configuration of tape 1 will be $\sqcup 101; 110$. The second tape is empty:



In its first phase, M_A moves the read/write head of tape 1 all the way to the right, copying x onto tape 2 and replacing it, on tape 1, with \square 's. It also replaces the $;$ with a \square . It then moves both read/write heads rightward to the last nonblank square. At the end of this phase, y is on tape 1; x is on tape 2, and each read/write head is pointing to the low-order digit of its number:



In its second phase, M_A moves back to the left, considering one pair of digits at a time. It sums them, treating a \square on either tape as a 0, records the result on tape 1, and remembers the carry digit for the next sum. Once it has encountered a blank on both tapes, it writes the carry digit if necessary and then it halts. At that point, its tapes are:



Theorem 17.1 Equivalence of Multitape and Single-Tape Turing Machines

Theorem: Let $M = (K, \Sigma, \Gamma, \delta, s, H)$ be a k -tape Turing machine, for some $k > 1$. Then there is a standard Turing machine $M' = (K', \Sigma', \Gamma', \delta', s', H')$ such that $\Gamma \subseteq \Gamma'$, and each of the following conditions holds:

- For any input string x , M on input x halts with output z on the first tape iff M' on input x halts at the same halting state (y , n , or h) and with z on its tape.
- If, on input x , M halts after n steps, then M' halts in $\mathcal{O}(n^2)$ steps.

Proof: The proof is by construction. The idea behind the construction is that M' will simulate M 's k tapes by treating its single tape as though it were divided into tracks. Suppose M has k tapes. Then an ordered k -tuple of values describes the contents of each of the tapes at some particular location. We also need to record the position of each of the k read/write heads. We do this by assigning two tracks to each of M 's tapes. The first track contains the value on the corresponding square of the tape. The second track contains a 1 if the read/write head is over that square and a 0 otherwise. Because all of M 's tapes are infinite, we need a way to line them up in order to be able to represent a slice through them. We will do this by starting with M 's initial configuration and then lining up the tapes so that all the read/write heads form a single column.

To see how this works, let $k = 2$. Then M 's initial configuration is shown in Figure 17.3 (a). M' will encode that pair of tapes on its single tape as shown in Figure 17.3 (b).

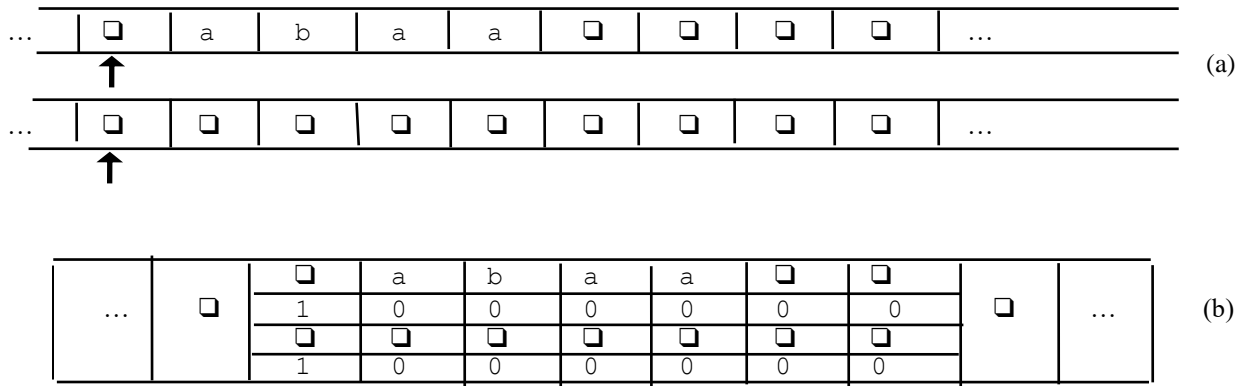


Figure 17.3 Encoding multiple tapes as multiple tracks

M 's tape, like every Turing machine tape, will contain \square 's on all but some finite number of squares, initially equal to the length of the input string w . But, if any of the read/write heads of M moves either left or right into the blank area, M will pause and encode the next square on its tape into tracks.

Like all standard Turing machines, when M' starts, its tape will contain its input. The first thing it will do is to reformat its tape so that it is encoded as k tracks, as shown above. It will then compute with the reformatted tape until it halts. Its final step will be to reformat the tape again so that its result (the string that is written on its simulated tape 1) is written, without the track encoding, on the tape. So M' will need a tape alphabet that can encode both the initial and final situations (a single character per tape square) and the encoding of k tapes (with k values plus k read/write head bits per tape square). So M' needs a tape alphabet that has a unique symbol for each element of $\Gamma \cup (\Gamma \times \{0, 1\})^k$. Thus $|\Gamma'| = |\Gamma| + (2 \cdot |\Gamma|)^k$. For example, to do the encoding shown above requires that Γ' contain symbols for \square , a , b , $(\square, 1, \square, 1)$, $(a, 0, \square, 0)$, $(b, 0, \square, 0)$, and so forth. $|\Gamma'| = 3 + 6^2 = 39$.

M' operates as follows:

1. Set up the multitrack tape:
 - 1.1. Move one square to the right to the first nonblank character on the tape.
 - 1.2. While the read/write head is positioned over some non- \square character c do:

Write onto the square the symbol that corresponds to a c on tape 1 and \square 's on every other track. On the first square, use the encoding that places a 1 on each even-numbered track (corresponding to the simulated read/write heads). On every other square, use the encoding that places a 0 on each even-numbered track.
2. Simulate the computation of M until (if) M would halt: (Each step will start with M' 's read/write head on the \square immediately to the right of the divided tape.)
 - 2.1. Scan left and store in the state the k -tuple of characters under the simulated read/write heads. Move back to the \square immediately to the right of the divided tape.
 - 2.2. Scan left and update each track as required by the appropriate transition of M . If necessary, subdivide a new square into tracks.
 - 2.3. Move back right.
3. When M would halt, reformat the tape to throw away all but track 1, position the read/write head correctly, and then go to M' 's halting state.

The construction that we just presented proves that any computation that can be performed by a k -tape Turing machine can be performed by a 1-tape machine. So adding any finite number of tapes adds no power to the Turing machine model. But there is a difference: The 1-tape machine must execute multiple steps for each single step taken by the k -tape machine. How many more? This question is only well defined if M (and so M') halts. So, if M halts, let:

- w be the input string to M , and
- n be the number of steps M executes before it halts.

Each time M' executes step 2, it must make two passes over the nonblank segment of its tape. How long is that segment? It starts out with length $|w|$ but if M ever moves off its input then M' will extend the encoded area and have to sweep over the new section on each succeeding pass. So we do not know exactly the length of the nonblank (encoded part) of M' 's tape, but we can put an upper bound on it by observing that M (and thus M') can write on at most one additional square at each step. So an upper bound on the length of encoded tape is $|w| + n$.

We can now compute an upper bound on the number of steps it will take M' to simulate the execution of M on w :

Step 1 (initialization):	=	$\mathcal{O}(w)$.
Step 2 (computation):		
Number of passes = n .		
Steps at each pass:		
For step 2.1	=	$2 \cdot (\text{length of tape})$.
	=	$2 \cdot (w + n)$.
For step 2.2	=	$2 \cdot (w + n)$.
Total	=	$\mathcal{O}(n \cdot (w + n))$.
Step 3 (clean up):	=	$\mathcal{O}(\text{length of tape})$.
Total:	=	$\mathcal{O}(n \cdot (w + n))$.

If $n \geq |w|$ (which it will be most of the time, including in all cases in which M looks at each square of its input at least once), then the total number of steps executed by M' is $\mathcal{O}(n^2)$. ■

17.3.2 Nondeterministic Turing Machines

So far, all of our Turing machines have been deterministic. What happens if we relax that restriction? Before we answer that question, let's review what we know so far about nondeterminism:

- With FSMs, we saw that nondeterminism is a very useful programming tool. It makes the task of designing certain classes of machines, including pattern matchers, easy. So it reduces the likelihood of programmer error. But nondeterminism adds no real power. For any NDFSM M , there exists an equivalent deterministic one M' . Furthermore, although the number of states in M' may be as many as 2^K , where K is the number of states in M , the time it takes to execute M' on some input string w is $\mathcal{O}(|w|)$, just as it is for M .
- With PDAs, on the other hand, we saw that nondeterminism adds power. There are context-free languages that can be recognized by a nondeterministic PDA for which no equivalent deterministic PDA exists.

So, now, what about Turing machines? The answer here is mixed:

- Nondeterminism adds no power in the sense that any computation that can be performed by a nondeterministic Turing machine can be performed by a corresponding deterministic one.
- But complexity is an issue. It may take exponentially more steps to solve a problem using a deterministic Turing machine than it does to solve the same problem with a nondeterministic Turing machine.

A **nondeterministic Turing machine** is a sextuple $(K, \Sigma, \Gamma, \Delta, s, H)$, where $K, \Sigma, \Gamma, s,$ and H are as for standard Turing machines, and Δ is a subset of $((K - H) \times \Gamma) \times (K \times \Gamma \times \{\leftarrow, \rightarrow\})$. In other words, we have replaced the transition function δ by the transition relation Δ , in much the same way we did when we defined nondeterministic FSMs and PDAs. The primary difference between our definition of nondeterminism for FSMs and PDAs and our definition of nondeterminism for Turing machines is that, since the operation of a Turing machine is not tied to the read-only, one-at-a-time consumption of its input characters, the notion of an ϵ -transition no longer makes sense.

But, just as before, we now allow multiple competing moves from a single configuration. And, as before, the easiest way to envision the operation of a nondeterministic Turing machine M is as a tree, as shown in Figure 17.4. Each node in the tree corresponds to a configuration of M and each path from the root corresponds to one sequence of configurations that M might enter.

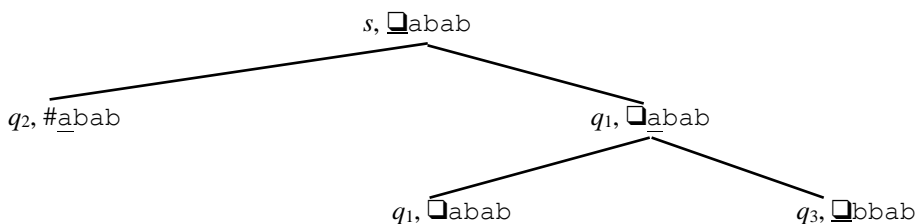


Figure 17.4 Viewing nondeterminism as search through a space of computation paths

Just as with PDAs, both the state and the data (in this case the tape) can be different along different paths.

Next we must define what it means for a nondeterministic Turing machine to:

- Decide a language.
- Semidecide a language.
- Compute a function.

We will consider each of these in turn.

Nondeterministic Deciding

What does it mean for a nondeterministic Turing machine to decide a language? What happens if the various paths disagree? The definition we will use is analogous to the one we used for both FSMs and PDAs. Recall that a computation of M is a sequence of configurations, starting in an initial configuration and ending in a halting configuration.

Let $M = (K, \Sigma, \Gamma, \Delta, s, H)$ be a nondeterministic Turing machine. Let w be an element of Σ^* . Then we will say that:

- M *accepts* w iff *at least one* of its computations accepts.
- M *rejects* w iff all of its computations reject.

M *decides* a language $L \subseteq \Sigma^*$ iff, $\forall w \in \Sigma^*$:

- There is a finite number of paths that M can follow on input w ,
- All of those paths are computations (i.e., they halt), and
- $w \in L$ iff M accepts w .

Example 17.16 Exploiting Nondeterminism For Finding Factors

Let $\text{COMPOSITES} = \{w \in \{0, 1\}^* : w \text{ is the binary encoding of a composite number}\}$. We can build a nondeterministic Turing machine M to decide COMPOSITES. M operates as follows on input w :

1. Nondeterministically choose two binary numbers p and q , both greater than 1, such that $|p|$ and $|q| \leq |w|$. Write them on the tape, after w , separated by ;. For example, consider the input string 110011. After this step, M 's tape, along one of its paths, will look like:

□110011;111;1111□□

2. Multiply p and q and put the answer, A , on the tape, in place of p and q . At this point, M 's tape will look like:

□110011;1101001□□

3. Compare A and w . If they are equal, accept (i.e., go to y); else reject (i.e., go to n).

Nondeterministic Semideciding

Next we must decide what it means for a nondeterministic Turing machine to semidecide a language. What happens if the various paths disagree? In particular, what happens if some paths halt and others don't. Again, the definition that we will use requires only that there exist at least one accepting path. We don't care how many nonaccepting (looping or rejecting) paths there are. So we will say:

A nondeterministic Turing machine $M = (K, \Sigma, \Gamma, \Delta, s, H)$ **semidecides** a language $L \subseteq \Sigma^*$ iff, $\forall w \in \Sigma^*$:

- $w \in L$ iff $(s, \sqcup w)$ yields at least one accepting configuration. In other words, there exists at least one path that halts and accepts w .

In the next example, as well as many others to follow, we will consider Turing machines whose inputs are strings that represent descriptions of Turing machines. We will describe later exactly how we can encode a Turing machine as a string. For now, imagine it simply as a program written out as we have been doing. We will use the notation $\langle M \rangle$ to mean the string that describes some Turing machine M (as opposed to the abstract machine M , which we might actually encode in a variety of different ways).

Example 17.17 Semideciding by Simulation

Let $L = \{\langle M \rangle : M \text{ is a Turing machine that halts on at least one string}\}$. We will describe later how one Turing machine can simulate another. Assuming that we can in fact do that, a Turing machine S to semidecide L will work as follows on input $\langle M \rangle$:

1. Nondeterministically choose a string w in Σ^* and write it on the tape.
2. Run M on w .
3. Accept.

Any individual branch of S will halt iff M halts on that branch's string. If a branch halts, it accepts. So at least one branch of S will halt and accept iff there is at least one string on which M halts.

As we will see in Chapter 21, semideciding is the best we are going to be able to do for L . We will also see that the approach that we have taken to designing S , namely to simulate some other machine and see whether it halts, will be one that we will use a lot when semideciding is the best that we can do.

Nondeterministic Function Computation

What about Turing machines that compute functions? Suppose, for example, that there are two paths through some Turing machine M on input w and they each return a different value. What value should M return? The first one it finds? Some sort of average of the two? Neither of these definitions seems to capture what we mean by a computation. And what if one path halts and the other doesn't? Should we say that M halts and returns a value? We choose a strict definition:

A nondeterministic Turing machine $M = (K, \Sigma, \Gamma, \Delta, s, H)$ **computes** a function f iff, $\forall w \in \Sigma^*$:

- All paths that M can follow on input w halt (i.e., all paths are computations), and
- All of M 's computations result in $f(w)$.

Does Nondeterminism Add Power?


One of the most important results that we will prove about Turing machines is that nondeterminism adds no power to the original model. Nondeterministic machines may be easier to design and they may run substantially faster, but there is nothing that they can do that cannot be done with some equivalent deterministic machine.

Theorem 17.2 Nondeterminism in Deciding and Semideciding Turing Machines

Theorem: If a nondeterministic Turing machine $M = (K, \Sigma, \Gamma, \Delta, s, H)$ decides a language L , then there exists a deterministic Turing machine M' that decides L . If a nondeterministic Turing machine M semidecides a language L , then there exists a deterministic Turing machine M' that semidecides L .

Proof Strategy: The proof will be by construction. The first idea we consider is the one we used to show that nondeterminism does not add power to FSMs. There we showed how to construct a new FSM M' that simulated the parallel execution of all of the paths of the original FSM M . Since M had a finite number of states, the number of sets of states that M' could be in was finite. So we simply constructed M' so that its states corresponded to sets of states from M . But that simple technique will not work for Turing machines because we must now consider the tape. Each path will need its own copy of the tape. Perhaps we could solve that problem by exploiting the technique from Section 17.3.1, where we used a single tape to encode multiple tapes. But that technique depended on advance knowledge of k , the number of tapes to be encoded. Since each path of M' will need a new copy of the tape, it isn't possible to put a bound on k . So we must reject this idea.

A second idea we might consider is simple depth-first search. If any path rejects, M' will back up and try an alternative. If any path accepts, M' will halt and accept. If M' explores the entire tree and all paths have rejected, then it rejects. But there is a big problem with this approach. What if one of the early paths is one that doesn't halt? Then M' will get stuck and never find some accepting path later in the tree. If we are concerned only with finding deterministic equivalents for nondeterministic *deciding* Turing machines, this is not an issue since all paths of any deciding machine must halt. But we must also show that every nondeterministic *semideciding* Turing machine has an equivalent deterministic machine. So we must abandon the idea of a depth-first search.

But we can build an M' that conducts a breadth-first search of the tree of computational paths that M generates. Suppose that there are never more than b competing moves available from any configuration of M . And suppose that h is the length of the longest path that M might have to follow before it can accept. Then M' may require $\mathcal{O}(b^{h+1})$ moves to find a solution since it may have to explore an entire tree of height h . Is an exponential increase in the time it takes a deterministic machine to simulate the computation of a nondeterministic one the best we can do? No one knows. Most people will bet yes. Yet no one has been able to prove that no better approach exists. A proof of the correctness of either a yes or a no answer to this question is worth \$1,000,000 . We will return to this question in Part V. There we will see that the standard way in which this question is asked is, "Does $P = NP$?"

For now though we will continue with the search-based approach. To complete this proof with such a construction requires that we show how to implement the search process on a Turing machine. Because breadth-first search requires substantial bookkeeping that is difficult to describe, we'll use an alternative but computationally similar technique, iterative deepening. We describe the construction in detail in [§ 643](#). ■

Theorem 17.3 Nondeterminism in Turing Machines that Compute Functions

Theorem: If a nondeterministic Turing machine $M = (K, \Sigma, \Gamma, \Delta, s, H)$ computes a function f then there exists a deterministic Turing machine M' that computes f .

Proof: The proof is by construction. It is very similar to the proof of Theorem 17.2 and is left as an exercise. ■

17.4 Simulating a “Real” Computer ♦

We’ve now seen that adding multiple tapes does not increase the power of Turing Machines. Neither does adding nondeterminism. What about adding features that would make a Turing Machine look more like a standard computer? Consider, for example, a simple computer that is composed of:

- An unbounded number of memory cells addressed by the integers starting at 0. These memory cells may be used to contain both program instructions and data. We’ll encode both in binary. Assume no limit on the number of bits that are stored in each cell.
- An instruction set composed of basic operations including read (R), move input pointer right or left (MIR, MIL), load (L), store (ST), add (A), subtract (S), jump (JUMP), conditional jump (CJUMP), and halt (H). Here’s a simple example program:

```
R      10      /* Read 2 bits from the input tape and put them into the accumulator.
MIR    10      /* Move the input pointer two bits to the right.
CJUMP 1001    /* If the value in the accumulator is 0, jump to location 1001.
A      10111   /* Add to the value in the accumulator the value at location 10111.
ST     10111   /* Store the result back in location 10111.
```

- A program counter.
- An address register.
- An accumulator in which operations are performed.
- A small fixed number of special purpose registers.
- An input file.
- An output file.

Can a Turing machine simulate the operation of such a computer? The answer is yes.

Theorem 17.4 A Real Computer Can be Simulated by a Turing Machine

Theorem: A random-access, stored program computer can be simulated by a Turing Machine. If the computer requires n steps to perform some operation, the Turing Machine simulation will require $\mathcal{O}(n^6)$ steps.

Proof: The proof is by construction of a simulator we’ll call *simcomputer*.

The simulator *simcomputer* will use 7 tapes:

- Tape 1 will hold the computer’s memory. It will be organized as a series of (address, value) pairs, separated by the delimiter #. The addresses will be represented in binary. The values will also be represented in binary. This means that we need a binary encoding of programs such as the addition one we saw above. We’ll use the first 4 bits of any instruction word for the operation code. The remainder of the word will store the address. So tape 1 will look like this:

```
#0,value_0#1,value_1#10,value_2#11,value_3#100,value_4# ...#
```

With an appropriate assignment of operations to binary encodings, our example program, if stored starting at location 0, would look like:

```
#0,000110010#1,11111001#10,001110011#11,001010111#....
```

Notice that we must explicitly delimit the words because there is no bound on their length. Addresses may get longer as the simulated program uses more words of its memory. Numeric values may increase as old values are added to produce new ones.

- Tape 2 will hold the program counter, which is just an index into the memory stored on tape 1.
- Tape 3 will hold the address register.
- Tape 4 will hold the accumulator.

- Tape 5 will hold the operation code of the current instruction.
- Tape 6 will hold the input file.
- Tape 7 will hold the output file, which will initially be blank.

Like all other multitape Turing machines, *simcomputer* will begin with its input on tape 1 and all other tapes blank. *Simcomputer* requires two inputs, the program to be simulated and the input on which the simulation is to be run. So we will encode them both on tape 1, separated by a special character that we will write as %.

We will assume that the program is stored starting in memory location 0, so the program counter will initially need to be initialized to 0. The simulator *simcomputer* operates as follows:

```

simcomputer(program) =
  /* Initialize.
  1. Move the input string to tape 6.
  2. Initialize the program counter (tape 2) to 0.

  /* Execute one pass through this loop for every instruction executed by program.
  3. Loop:
    3.1. Starting at the left of the nonblank portion of tape 1, scan to the right looking for an index that matches
        the contents of tape 2 (the program counter).

    /* Decode the current instruction and increment the program counter.
    3.2. Copy the operation code to tape 5.
    3.3. Copy the address to tape 3.
    3.4. Add 1 to the value on tape 2.

    /* Retrieve the operand.
    3.5. Starting at the left again, scan to the right looking for the address that is stored on tape 3.

    /* Execute the instruction.
    3.6. If the operation is Load, copy the operand to tape 4 (the accumulator).
    3.7. If the operation is Add, add the operand to the value on tape 4.
    3.8. If the operation is Jump, copy the value on tape 3 to tape 2 (the program counter).
    3.9. And so forth for the other operations.

```

How many steps must *simcomputer* execute to simulate a program that runs in n steps? It executes the outer loop of step 3 n times. How many steps are required at each pass through the loop? Step 3.1 may take t steps, if t is the length of tape 1. Step 3.2 takes a constant number of steps. Step 3.3 may take a steps if a is the number of bits required to store the longest address that is used on tape 1. Step 3.4 may also take a steps. Step 3.5 again may have to scan all of tape 1, so it may take t steps. The number of steps required to execute the instruction varies:

- Addition takes v steps if v is the length of the longer operand.
- Load takes v steps if v is the length of the value to be loaded.
- Store generally takes v steps if v is the length of the value to be stored. However, suppose that the value to be stored is longer than the value that is already stored at that location. Then *simcomputer* must shift the remainder of Tape 1 one square to the right in order to have room for the new value. So executing a Store instruction could take t steps (where t is the length of tape 1).

The remainder of the operations can be analyzed similarly. Notice that we have included no complex operations like multiply. (But this is not a limitation. Multiply can be implemented as a sequence of additions.) So it is straightforward to see that the number of steps required to perform any of the operations that we have defined is, in the worst case, a linear function of t , the length of tape 1.

So how long is tape 1? It starts out at some length k . Each instruction has the ability to increase the number of memory locations by 1 since a store instruction can store to an address that was not already represented on the tape. And each

instruction has the ability to increase by 1 the length of a machine “word”, since the add instruction can create a value that is one bit longer than either of its operands. So, after n simulated steps, t , the length of the tape, could be $k + n^2$ (if new words are created and each word gets longer). If we assume that n is large relative to k , we can say that the length of the tape, after n steps, is $\mathcal{O}(n^2)$. So the number of steps that *simcomputer* must execute to simulate each step of the original program is $\mathcal{O}(n^2)$. Since *simcomputer* must simulate n steps of the original program, the total number of steps executed by *simcomputer* is $\mathcal{O}(n^3)$.

The simulator *simcomputer* uses 7 tapes. We know, from Theorem 17.1, that a k -tape Turing machine that executes n steps can be simulated in $\mathcal{O}(n^2)$ steps by a one-tape, standard Turing Machine. So the total number of steps it would take a one-tape standard Turing Machine to simulate one of our programs executing n steps is $\mathcal{O}(n^6)$. While this represents a nontrivial increase in the number of steps, it is important to note that the increase is a polynomial function of n . It does not grow exponentially, the way the simulation of a nondeterministic Turing Machine did. ■

Any program that can be written in any modern programming language can be compiled into code for a machine such as the simple random access machine that we have just described. Since we have shown that any such machine can be simulated by a Turing machine, we will begin to use clear pseudocode to define Turing machines.

17.5 Alternative Turing Machine Definitions ♦

We have provided one definition for what a Turing machine is and how it operates. There are many equivalent alternatives. In this section we will explore two of them.

17.5.1 One-Way vs. Two-Way Infinite Tape

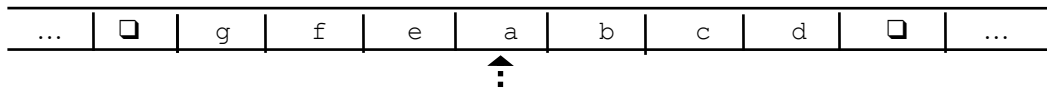
Many books define a Turing machine to have a tape that is infinite in only one direction. We use a two-way infinite tape. Does this difference matter? In other words, are there any problems that one kind of machine can solve that the other one cannot? The answer is no.

Theorem 17.5 A One-Way Infinite Tape is Equivalent to a Two-Way Infinite Tape

Theorem: Any computation by a Turing machine with a two-way infinite tape can be simulated by a Turing machine with a one-way infinite tape.

Proof: Let M be a Turing machine with a two-way infinite tape. We describe M' , an equivalent machine whose tapes are infinite in only one direction. M' will use three tapes. The first will hold that part of M 's tape that starts with the square under the read/write head and goes to the right. The second of M' 's tapes will hold that part of M 's tape to the left of the read/write head. The third tape will count, in unary, the number of moves that M has made so far. An example of this encoding is shown in Figure 17.5.

The two-way tape:



The simulation:

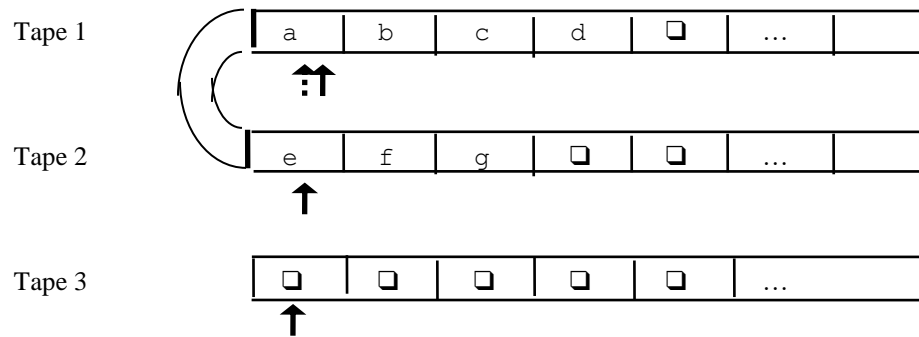


Figure 17.5 Simulating a two-way infinite tape on a one-way infinite tape

M 's read/write head is shown above as a dashed arrow. M' has three read/write heads (shown as dark arrows above), one for each tape. It will use its finite state controller to keep track of whether the simulated read/write head is on tape 1 or tape 2. If the simulated read/write head is on tape 1, square t , then M' 's tape 1 read/write head will be on square t and its tape 2 read/write head will be on the leftmost square. Similarly if the simulated read/write head is on tape 2.

Initially, M' 's tape 1 will be identical to M 's tape, M' 's tape 2 will be blank, and M' 's tape 3 will also be blank (since no moves have yet been made).

The simulation: M' simulates each step of M . If M attempts to move to the left, off the end of its tape, M' will begin writing at the left end of tape 2. If M continues to move left, M' will move right on tape 2. If M moves right and goes back onto its original tape, M' will begin moving right on tape 1. If M would halt, then M' halts the simulation.

But, if M' is computing a function, then M' must also make sure, when it halts, that its tape 1 contains exactly what M 's tape would have contained. Some of that may be on tape 2. If it is, then the contents of tape 1 must be shifted to the right far enough to allow the contents of tape 2 to be moved up. The maximum number of symbols that M' may have written on tape 2 is n , where n is the number of steps executed by M . Tape 3 contains n . So M' moves n squares to the right on tape 2. Then it moves leftward, one square at a time as long as it reads only blanks. Each time it moves to the left, it erases a 1 from tape 3. When it hits the first nonblank character, tape 3 will contain the unary representation of the number of times M' must shift tape 1 one square to the right and then copy one symbol from tape 2 to tape 1. M' executes this shifting process the required number of times and then halts. ■

17.5.2 Stacks vs. a Tape

When we switched from working with PDAs to working with Turing machines, we gave up the use of a stack. The Turing machine's infinite tape has given us more power than we had with the PDA's stack. But it makes sense to take one more look at the stack as a memory device and to ask two questions:

1. Did we lose anything by giving up the PDA's stack in favor of the Turing machine's tape?
2. Could we have gotten the power of a Turing machine's tape using just stacks?

Simulating a Stack by a Turing Machine Tape

Theorem 17.6 A PDA Can be Simulated by a Turing Machine

Theorem: The operation of any PDA P can be simulated by some Turing machine M .

Proof: The proof is by construction. Given some PDA P , we construct a (possibly) nondeterministic Turing machine M to simulate the operation of P . Since there is a finite number of states in P , M can keep track of the current state of P in its own finite state controller.

Each branch of M will use two tapes, one for the input and one for the stack, as shown in Figure 17.6. Tape 1 will function just like the read-only stream of input that is fed to the PDA. M will never write on tape 1 and will only move to the right, one square at a time. Tape 2 will mimic the behavior of M 's stack, with its read/write head moving back and forth as symbols are pushed onto and popped from the stack.

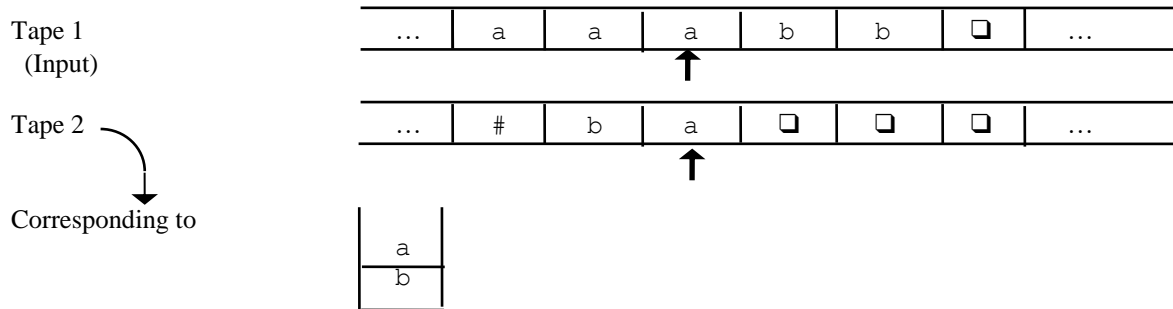


Figure 17.6 Simulating a PDA by a Turing machine

M will operate as follows:

1. Initialization: write #, indicating the bottom of the stack, under the read/write head of Tape 2. Tape 2's read/write head will always remain positioned on the top of the stack. Set the simulated state S_{sim} to s .
2. Simulation: let the character under the read/write head of Tape 1 be c . At each step of the operation of P do:
 - 2.1. If $c = \square$, halt and accept if S_{sim} is an accept state of P and reject otherwise.
 - 2.2. Nondeterministically choose from Δ a transition of the form $((S_{sim}, c, pop), (q_2, push))$ or $((S_{sim}, \epsilon, pop), (q_2, push))$. In other words, choose some transition from the current state that either reads the current input character or reads ϵ .
 - 2.3. Scan leftward on Tape 2 $|pop|$ squares, checking to see whether Tape 2 matches pop . If it does not, terminate this path. If it does, then move rightward on Tape 2 $|push|$ squares copying $push$ onto Tape 2.
 - 2.4. If we are not following an ϵ -transition, move the read/write head of Tape 1 one square to the right and set c to the character on that square.
 - 2.5. Set S_{sim} to q_2 and repeat.

■

So we gave up no power when we abandoned the PDA's stack in favor of the Turing machine's tape.

Simulating a Turing Machine Tape by Using Two Stacks

What about the other way around? Is there any way to use stacks to get the power of an infinite, writeable tape? The answer is yes. Any Turing machine M can be simulated by a PDA P with two stacks. Suppose that M 's tape is as shown in Figure 17.7 (a). Then P 's two stacks will be as shown in Figure 17.7 (b).

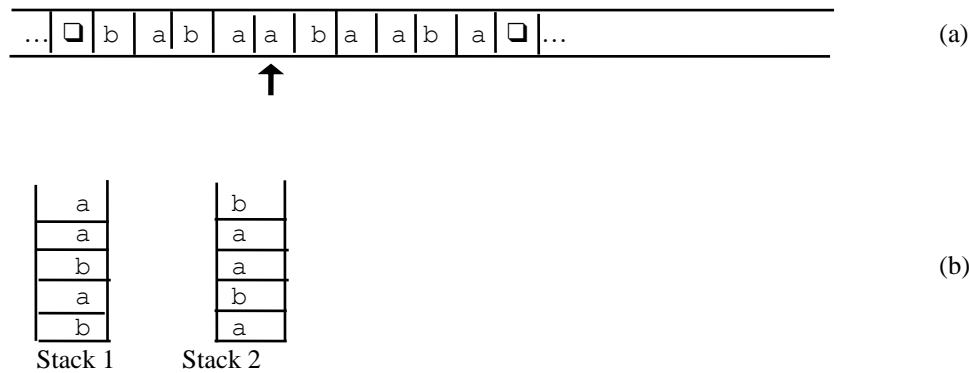


Figure 17.7 Simulating a Turing machine tape with two stacks

Stack 1 contains M 's active tape up to and including the square that is currently under the read/write head. Stack 2 contains the remainder of M 's active tape. If M moves to the left, the top character from stack 1 is popped and then pushed onto stack 2. If M moves onto the blank region to the left of its tape, then the character that it writes is simply pushed onto the top of stack 1. If M moves to the right, the top character from stack 2 is popped and then pushed onto stack 1. If M moves onto the blank region to the right of its tape, then the character that it writes is simply pushed onto the top of stack 1.

17.6 Encoding Turing Machines as Strings

So far, all of our Turing machines have been hardwired (just like early computers). Does it make sense, just as it did with real computers, to develop a programmable Turing machine: a single Turing machine that accepts as input a (M : Turing machine, s : input string) pair and outputs whatever M would output when started up on s ? The answer is yes. We will call such a device the *universal Turing machine* or simply U .

To define U we need to do two things:

1. Define an encoding scheme that can be used to describe to U a (Turing machine, input string) pair.
2. Describe the operation of U given such an encoded pair.

17.6.1 An Encoding Scheme for Turing Machines

We need to be able to describe an arbitrary Turing machine $M = (K, \Sigma, \Gamma, \delta, s, H)$ as a string that we will write as $\langle M \rangle$. When we define the universal Turing machine, we will have to assign it a fixed input alphabet. But the machines we wish to input to it may have an arbitrary number of states and they may exploit alphabets of arbitrary size. So we need to find a way to encode an arbitrary number of states and a tape alphabet of arbitrary size using some new alphabet of fixed size. The obvious solution is to encode both state sets and alphabets as binary strings.

We begin with K . We will determine i , the number of binary digits required to encode the numbers from 0 to $|K|-1$. Then we will number the states from 0 to $|K|-1$ and assign to each state the binary string of length i that corresponds to its assigned number. By convention, the start state s will be numbered 0. The others may be numbered in any order. Let t' be the binary string assigned to state t . Then we assign strings to states as follows:

- If t is the halting state y , assign it the string $y't'$.
- If t is the halting state n , assign it the string nt' .
- If t is any other state, assign it the string qt' .

Example 17.18 Encoding the States of a Turing Machine

Suppose that we are encoding a Turing machine M with 9 states. Then it will take four binary digits to encode the names of the 9 states. The start state s will be encoded as $q0000$. Assuming that y has been assigned the number 3 and n has been assigned the number 4, the remaining states will be encoded as $q0001$, $q0010$, $y0011$, $n0100$, $q0101$, $q0110$, $q0111$, and $q1000$.

Next we will encode the tape alphabet in a similar fashion. We will begin by determining j , the number of binary digits required to encode the numbers from 0 to $|\Gamma|-1$. Then we will number the characters (in any order) from 0 to $|\Gamma|-1$ and assign to each character the binary string of length j that corresponds to its assigned number. Finally, we will assign to each symbol y the string ay' , where y' is the binary string already assigned to y .

Example 17.19 Encoding the Tape Alphabet of a Turing Machine

Suppose that we are encoding a Turing machine M with $\Gamma = \{\square, a, b, c\}$. Then it will take two binary digits to encode the names of the four characters. The assignment of numbers to the characters is arbitrary. It just must be done consistently throughout the encoding. So, for example, we could let:

$\square =$	a00
a =	a01
b =	a10
c =	a11

Next we need a way to encode the transitions of δ , each of which is a 5-tuple: (state, input character, state, output character, move). We have just described how we will encode states and tape characters. There are only two allowable moves, \rightarrow and \leftarrow , so we can just use those two symbols to stand for their respective moves. We will encode each transition in δ as a string of exactly the form (state,character,state,character,move), using the state, character, and move encodings that we have just described. Then we can specify δ as a list of transitions separated by commas.

The only thing we have left to specify is H , which we will do as follows:

- States with no transitions out are in H (in other words, we don't need to say anything explicitly about this).
- If M decides a language, then $H = \{y, n\}$, and we will adopt the convention that y is the lexicographically smaller of the two halting states.
- If M has only one halting state (which is possible if it semidecides a language), we will take it to be y .

With these conventions, we can completely specify almost all Turing machines simply as a list of transitions. But we must also consider the special case of the simple Turing machine M_{none} , shown in Figure 17.8. M_{none} has no transitions but it is a legal Turing machine (that semidecides Σ^*). To enable us to represent machines like M_{none} , we add one more convention: When encoding a Turing machine M , for any state q in M that has no incoming transitions, add to M 's encoding the substring (q) . So M_{none} would be encoded as simply $(q0)$.



Figure 17.8 Encoding a Turing machine with no transitions

Example 17.20 Encoding a Complete Turing Machine Description

Consider $M = (\{s, q, h\}, \{a, b, c\}, \{\square, a, b, c\}, \delta, s, \{h\})$, where $\delta =$

state	symbol	δ
s	\square	$(q, \square, \square \rightarrow)$
s	a	(s, b, \rightarrow)
s	b	(q, a, \leftarrow)
s	c	(q, b, \leftarrow)
q	\square	$(s, a \square \rightarrow)$
q	a	(q, b, \rightarrow)
q	b	(q, b, \leftarrow)
q	c	(h, a, \leftarrow)

We start encoding M by determining encodings for each of its states and tape symbols:

state/symbol	representation
s	q00
q	q01
h	h10
\square	a00
a	a01
b	a10
c	a11

The complete encoding of M , which we will denote by $\langle M \rangle$, is then:

$(q00, a00, q01, a00, \rightarrow), (q00, a01, q00, a10, \rightarrow), (q00, a10, q01, a01, \leftarrow), (q00, a11, q01, a10, \leftarrow),$
 $(q01, a00, q00, a01, \rightarrow), (q01, a01, q01, a10, \rightarrow), (q01, a10, q01, a10, \leftarrow), (q01, a11, h10, a01, \leftarrow).$

17.6.2 Enumerating Turing Machines

Now that we have an encoding scheme for Turing machines, it is possible to create an enumeration of them.

Theorem 17.7 We Can Lexicographically Enumerate the Valid Turing Machines

Theorem: There exists an infinite lexicographic enumeration of:

- All syntactically valid Turing machines.
- All syntactically valid Turing machines whose input alphabet is some particular set Σ .
- All syntactically valid Turing machines whose input alphabet is some particular set Σ and whose tape alphabet is some particular set Γ .

Proof: Fix an alphabet $\Sigma = \{ (,), a, q, 0, 1, \text{comma}, \rightarrow, \leftarrow \}$, i.e., the set of characters that are used in the Turing machine encoding scheme that we just described. Let the symbols in Σ be ordered as shown in the list we just gave. The following procedure lexicographically enumerates all syntactically valid Turing machines:

- Lexicographically enumerate the strings in Σ^* .
- As each string s is generated, check to see whether it is a syntactically valid Turing machine description. If it is, output it.

To enumerate just those Turing machines whose input and/or tape alphabets are limited to the symbols in some particular sets Σ and Γ , add to step 2 a check to see that only alphabets of the appropriate sizes are allowed. ■

With this procedure in hand, we can now talk about the i^{th} Turing machine. It is the i^{th} element generated by the enumeration procedure.

17.6.3 Another Win of Encoding

Our motivation for defining what we mean by $\langle M \rangle$ was that we would like to be able to input a definition of M to the universal Turing machine U , which will then execute M . But it turns out that, now that we have a well-defined string encoding $\langle M \rangle$ for any Turing machine M , we can pass $\langle M \rangle$ as input to programs other than U and ask those programs to operate on M . So we can talk about some Turing machine T that takes the description of another Turing machine (say M_1) as input and transforms it into a description of a different machine (say M_2) that performs some different, but possibly related task. We show this schematically in Figure 17.9.

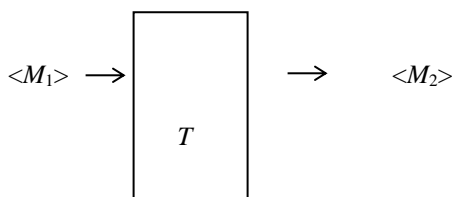


Figure 17.9 Turing machine T takes one Turing machine as input and creates another as its output

We will make extensive use of this idea of transforming one Turing machine into another when we discuss the use of reduction to show that various problems are undecidable.

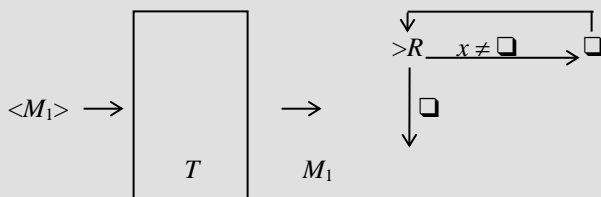
Example 17.21 One Turing Machine Operates on the Description of Another

Define a Turing machine T whose specifications are:

Input: $\langle M_1 \rangle$, where M_1 is a Turing machine that reads its input tape and performs some operation P on it.

Output: $\langle M_2 \rangle$, where M_2 is a Turing machine that performs P on an empty input tape.

The job of T is shown in the following diagram. We have, for convenience here, described $\langle M_2 \rangle$ using our macro language, but we could have written out the detailed string encoding of it.



T constructs the machine M_2 that starts by erasing its input tape. Then it passes control to M_1 . So we can define T as follows:

$T(\langle M_1 \rangle) =$
Output the machine shown on the right above.

17.6.4 Encoding Multiple Inputs to a Turing Machine

Every Turing machine takes a single string as its input. Sometimes, however, we wish to define a Turing machine that operates on more than one object. For example, we are going to define the universal Turing machine U to accept a machine M and a string w and to simulate the execution of M on w . To do this, we need to encode both arguments as a single string. We can easily do that by encoding each argument separately and then concatenating them together,

separated by some character that is not in any of the alphabets used in forming the individual strings. For example, we could encode the pair $\langle M \rangle, \langle aabb \rangle$ as $\langle M \rangle; \langle aabb \rangle$. We will use the notation $\langle x_1, x_2, \dots, x_n \rangle$ to mean a single string that encodes the sequence of individual values x_1, x_2, \dots, x_n .

17.7 The Universal Turing Machine

We are now in a position to return to the problem of building a universal Turing machine, which we'll call U . U is not truly "universal" in the sense that it can compute "everything". As we'll see in the next few chapters, there are things that cannot be computed by any Turing machine. U is, however, universal in the sense that, given an arbitrary Turing machine M and an input w , U will simulate the operation of M on w .

We can state U 's specification as follows: On input $\langle M, w \rangle$, U must:

- Halt iff M halts on w .
- If M is a deciding or a semideciding machine, then:
 - If M accepts, accept.
 - If M rejects, reject.
- If M computes a function, then $U(\langle M, w \rangle)$ must equal $M(w)$.

U will use three tapes to simulate the execution of M on w :

- Tape 1 will correspond to M 's tape.
- Tape 2 will contain $\langle M \rangle$, the "program" that U is running.
- Tape 3 will contain the encoding of the state that M is in at any point during the simulation. Think of tape 3 as holding the program counter.

When U begins, it will have $\langle M, w \rangle$ on tape 1. (Like all multitape machines, it starts with its input on tape 1 and all other tapes blank.) Figure 17.10 (a) illustrates U 's three tapes when it begins. It uses the multitrack encoding of three tapes that we described in Section 17.3.1.

U 's first job is to initialize its tapes. To do so, it must do the following:

1. Transfer $\langle M \rangle$ from tape 1 to tape 2 (erasing it from tape 1).
2. Examine $\langle M \rangle$ to determine the number of states in M and thus i , the number of binary digits required to encode M 's states. Write $q0^i$ (corresponding to the start state of M) on tape 3.

Assume that it takes three bits to encode the states of M . Then, after initialization, U 's tapes will be as shown in Figure 17.10 (b).



Figure 17.10 The tapes of the universal Turing machine U

U begins simulating M with the read/write heads of its three tapes as shown above. More generally, it will start each step of its simulation with the read/write heads placed as follows:

- Tape 1's read/write head will be over the a that is the first character of the encoding of the current character on M 's tape.
- Tape 2's read/write head will be at the beginning of $\langle M \rangle$.
- Tape 3's read/write head will be over the q of the program counter.

Following initialization as described above, U operates as follows:

1. Until M would halt do:
 - 1.1. Scan tape 2 for a quintuple that matches the current state, input pair.
 - 1.2. Perform the associated action, by changing tapes 1 and 3. If necessary, extend the simulated tape that is encoded on tape 1.
 - 1.3. If no matching quintuple found, halt.
2. Report the same result M would report:
 - If M is viewed as a deciding or semideciding machine for some language L : if the simulated state of M is y , then accept. If the simulated state is n , then reject.
 - If M is viewed as a machine that computes a function: reformat the tape so that the value of tape 1 is all that is left.

How long does it take U to simulate the computation of M ? If M would halt in k steps, then U must go through its loop k times. Each time through the loop, it must scan $\langle M \rangle$ to find out what to do. So U takes $\mathcal{O}(|M| \cdot k)$ steps.

Now we know that if we wanted to build real Turing machines we could build one physical machine and feed it descriptions of any other Turing machines that we wanted to run. So this is yet another way in which the Turing machine is a good general model of computation.

The existence of U enables us to prove the following theorem:

Theorem 17.8 One Turing Machine Can Simulate Another

Theorem: Given any Turing machine M and input string w , there exists a Turing machine M' that simulates the execution of M on w and:

- halts iff M halts on w , and
- if it halts, returns whatever result M returns.

Proof: Given a particular M and w , we construct a specific M' to operate as follows:

$M'(x) =$
 Invoke the universal Turing machine U on the string $\langle M, w \rangle$.

Notice that M' ignores its own input (which we've called x). It is a constant function. M' halts iff U halts and, if it halts, will return the result of executing M on w . ■

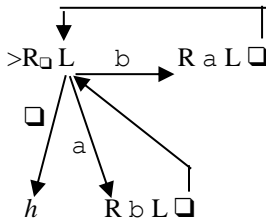
Theorem 17.8 enables us to write, in a Turing machine definition, the pseudocode, "Run M on w ," and then branch based on whether or not M halts (and, if it halts, what it returns).

If the universal Turing machine is a good idea, what about universal other things? Could we, for example, define a universal FSM? Such an FSM would accept the language $L = \{ \langle F, w \rangle : F \text{ is a finite state machine and } w \in L(F). \}$ The answer is no. Since any FSM has only a finite amount of memory, it has no way to remember and then execute a program of arbitrary length. We have waited until now to introduce the idea of a universal machine because we had to.

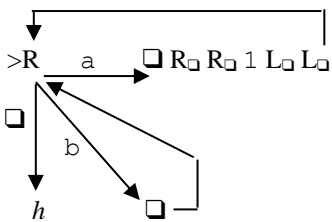
17.8 Exercises

1) Give a short English description of what each of these Turing machines does:

a) $\Sigma_M = \{a, b\}$. $M =$



b) $\Sigma_M = \{a, b\}$. $M =$



2) Construct a standard, deterministic, one-tape Turing machine M to decide each of the following languages L . You may find it useful to define subroutines. Describe M in the macro language defined in Section 17.1.5.

- a) $\{x * y = z : x, y, z \in 1^+ \text{ and, when } x, y, \text{ and } z \text{ are viewed as unary numbers, } xy = z\}$. For example, the string $1111 * 11 = 11111111 \in L$.
- b) $\{a^i b^j c^i d^j, i, j \geq 0\}$.
- c) $\{w \in \{a, b, c, d\}^* : \#_b(w) \geq \#_c(w) \geq \#_d(w) \geq 0\}$.

3) Construct a standard, deterministic, one-tape Turing machine M to compute each of the following functions:

a) The function sub_3 , which is defined as follows:

$$sub_3(n) = \begin{cases} n-3 & \text{if } n > 2 \\ 0 & \text{if } n \leq 2. \end{cases}$$

Specifically, compute sub_3 of a natural number represented in binary. For example, on input 10111 , M should output 10100 . On input 11101 , M should output 11010 . (Hint: you may want to define a subroutine.)

- b) Addition of two binary natural numbers (as described in Example 17.13). Specifically, given the input string $\langle x \rangle; \langle y \rangle$, where $\langle x \rangle$ is the binary encoding of a natural number x and $\langle y \rangle$ is the binary encoding of a natural number y , M should output $\langle z \rangle$, where z is the binary encoding of $x + y$. For example, on input $101; 11$, M should output 1000 .
- c) Multiplication of two unary numbers. Specifically, given the input string $\langle x \rangle; \langle y \rangle$, where $\langle x \rangle$ is the unary encoding of a natural number x and $\langle y \rangle$ is the unary encoding of a natural number y , M should output $\langle z \rangle$, where z is the unary encoding of xy . For example, on input $111; 1111$, M should output 111111111111 .
- d) The proper subtraction function *monus*, which is defined as follows:

$$\text{monus}(n, m) = \begin{cases} n - m & \text{if } n > m \\ 0 & \text{if } n \leq m. \end{cases}$$

Specifically, compute *monus* of two natural numbers represented in binary. For example, on input $101; 11$, M should output 1 . On input $11; 101$, M should output 0 .

- 4) Construct a Turing machine M that computes the function $f: \{a, b\}^* \rightarrow N$, where:

$$f(x) = \text{the unary encoding of } \max(\#_a(x), \#_b(x)).$$

For example, on input $aaabbb$, M should output 1111 . M may use more than one tape. It is not necessary to write the exact transition function for M . Describe it in clear English.

- 5) Construct a Turing machine M that converts binary numbers to their unary representations. So, specifically, on input $\langle w \rangle$, where w is the binary encoding of a natural number n , M will output 1^n . (Hint: use more than one tape.)
- 6) Let M be a three-tape Turing machine with $\Sigma = \{a, b, c\}$ and $\Gamma = \{a, b, c, \square, 1, 2\}$. We want to build an equivalent one-tape Turing machine M' using the technique described in Section 17.3.1. How many symbols must there be in Γ' ?
- 7) In Example 13.2, we showed that the language $L = \{a^{n^2}, n \geq 0\}$ is not context-free. Show that it is in D by describing, in clear English, a Turing machine that decides it. (Hint: use more than one tape.)
- 8) In Example 17.9, we showed a Turing machine that decides the language WcW . If we remove the middle marker c , we get the language WW . Construct a Turing machine M that decides WW . You may want to exploit nondeterminism. It is not necessary to write the exact transition function for M . Describe it in clear English.
- 9) In Example 4.9, we described the Boolean satisfiability problem and we sketched a nondeterministic program that solves it using the function *choose*. Now define the language $\text{SAT} = \{\langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ is satisfiable}\}$. Describe in clear English the operation of a nondeterministic (and possibly n -tape) Turing machine that decides SAT.
- 10) Prove Theorem 17.3.
- 11) Prove rigorously that the set of regular languages is a *proper* subset of D.
- 12) In this question, we explore the equivalence between function computation and language recognition as performed by Turing machines. For simplicity, we will consider only functions from the nonnegative integers to the nonnegative integers (both encoded in binary). But the ideas of these questions apply to any computable function. We'll start with the following definition:

- Define the *graph* of a function f to be the set of all strings of the form $[x, f(x)]$, where x is the binary encoding of a nonnegative integer, and $f(x)$ is the binary encoding of the result of applying f to x .

For example, the graph of the function *succ* is the set $\{[0, 1], [1, 10], [10, 11], \dots\}$.

- a) Describe in clear English an algorithm that, given a Turing machine M that computes f , constructs a Turing machine M' that decides the language L that contains exactly the graph of f .
- b) Describe in clear English an algorithm that, given a Turing machine M that decides the language L that contains the graph of some function f , constructs a Turing machine M' that computes f .
- c) A function is said to be partial if it may be undefined for some arguments. If we extend the ideas of this exercise to partial functions, then we do not require that the Turing machine that computes f halts if it is given some input x for which $f(x)$ is undefined. Then L (the graph language for f), will contain entries of the form $[x, f(x)]$ for only those values of x for which f is defined. In that case, it may not be possible to decide L , but it will be possible to semidecide it. Do your constructions for parts (a) and (b) work if the function f is partial? If not, explain how you could modify them so they will work correctly. By “work”, we mean:
- For part (a): given a Turing machine that computes $f(x)$ for all values on which f is defined, build a Turing machine that semidecides the language L that contains exactly the graph of f ;
 - For part (b): given a Turing machine that semidecides the graph language of f (and thus accepts all strings of the form $[x, f(x)]$ when $f(x)$ is defined), build a Turing machine that computes f .

13) What is the minimum number of tapes required to implement a universal Turing machine?

14) Encode the following Turing Machine as an input to the universal Turing machine that is described in Section 17.7:

$M = (K, \Sigma, \Gamma, \delta, q_0, \{h\})$, where:

$K = \{q_0, q_1, h\}$,

$\Sigma = \{a, b\}$,

$\Gamma = \{a, b, c, \square\}$, and

δ is given by the following table:

q	σ	$\delta(q, \sigma)$
q_0	a	(q_1, b, \rightarrow)
q_0	b	(q_1, a, \rightarrow)
q_0	\square	$(h, \square, \rightarrow)$
q_0	c	(q_0, c, \rightarrow)
q_1	a	(q_0, c, \rightarrow)
q_1	b	(q_0, b, \leftarrow)
q_1	\square	(q_0, c, \rightarrow)
q_1	c	(q_1, c, \rightarrow)

18 The Church-Turing Thesis

The Turing machine is the most powerful of the models of computation that we have so far considered. There are problems that can be solved by a Turing machine that cannot be solved by a PDA, just as there are problems that could be solved by a PDA but not by an FSM. Is this the end of the line, or should we expect a sequence of even more powerful models?

One way of looking at things suggests that we should expect to keep going. A simple counting argument shows that there are more languages than there are Turing machines:

- There is at most a countably infinite number of Turing machines since we can lexicographically enumerate all the strings that correspond to syntactically legal Turing machines.
- There is an uncountably infinite number of languages over any nonempty alphabet.
- So there are more languages than there are Turing machines.

So there are languages that cannot be recognized by any Turing machine. But can we do better by creating some new formalism? If any such new formalism shares with Turing machines the property that each instance of it has a finite description (for example a finite length Java program or a finite length grammar) then the same argument will apply to it and there will still be languages that it cannot describe.

But there might be some alternative model in which we could write finite length programs and for which no equivalent Turing machine exists. Is there? We showed in the last chapter that there are several features (e.g., multiple tapes, nondeterminism) that we could add to our definition of a Turing machine without increasing its power. But does that mean that there is nothing we could add that would make a difference? Or might there be some completely different model that has more power than the Turing machine?

18.1 The Thesis

Another way to ask the question about the existence of a more powerful model is this: recall that we have defined an *algorithm* to be a detailed procedure that accomplishes some clearly specified task. Note that this definition is general enough to include decision procedures (functions that return Boolean values), as well as functions that return values of other types. In fact, it is general enough to include recipes for beef Wellington. We will, however, focus just on tasks that involve computation. Now we can restate our question: “Is there any computational algorithm that cannot be implemented by some Turing machine? Then, if there is, can we find some more powerful model in which we could implement that algorithm?” Note that we are assuming here that both real-world inputs and real-world outputs can be appropriately encoded into symbols that can be written onto a device such as the Turing machine’s tape. We are not talking about whether an abstract Turing machine can actually chop mushrooms, take pictures, produce sound waves, or turn a steering wheel.

During the first third of the 20th century, a group of influential mathematicians was focused on developing a completely formal basis for mathematics. Out of this effort emerged, among other things, *Principia Mathematica* [Whitehead and Russell 1910, 1912, 1913], which is often described as the most influential work on logic ever written. Among its achievements was the introduction of a theory of types that offers a way out of Russell’s paradox⁸. The continuation and the ultimate success of this line of work depended on positive answers to two key questions:

1. Is it possible to axiomatize all of the mathematical structures of interest in such a way that every true statement becomes a theorem? We will allow the set of axioms to be infinite, but it must be decidable. In other words, there must exist an algorithm that can examine a string and determine whether or not it is an axiom.

⁸ Let M be “the set of all sets that are not members of themselves”. Is M a member of M ? The fact that either answer to this question leads to a contradiction was noticed by Bertrand Russell probably in about 1901. The question is called “Russell’s paradox”.

2. Does there exist an algorithm to decide, given a set of axioms, whether a given statement is a theorem? In other words, does there exist an algorithm that always halts and that returns *True* when given a theorem and *False* otherwise?

Principia Mathematica played a landmark role in the development of mathematical logic in the early part of the 20th century. 45 years later it played another landmark role, this time in a discipline that Whitehead and Russell could never have imagined. In 1956, the Logic Theorist, often regarded as the first artificial intelligence program, proved most of the theorems in Chapter 2 of *Principia Mathematica*. © 760.

It was widely believed that the answer to both of these questions was yes. Had it been, perhaps the goal of formalizing all of mathematics could have been attained. But the answer to both questions is no. Three papers that appeared within a few years of each other shattered that dream.

Kurt Gödel showed, in the proof of his Incompleteness Theorem [Gödel 1931], that the answer to question 1 is no. In particular, he showed that there exists no decidable axiomatization of Peano arithmetic (the natural numbers plus the operations *plus* and *times*) that is both consistent and complete. By complete we mean that all true statements in the language of the theory are theorems. Note that an infinite set of axioms is allowed, but it must be decidable. So an infinite number of true statements can be made theorems simply by adding new axioms. But Gödel showed that, no matter how often that is done, there must remain other true statements that are unprovable.

Question 2 had been clearly articulated a few years earlier in a paper by David Hilbert and Wilhelm Ackermann [Hilbert and Ackermann 1928]. They called it the *Entscheidungsproblem*. (“Entscheidungsproblem” is just the German word for “decision problem”.) There are three equivalent ways to state the problem:

- “Does there exist an algorithm to decide, given an arbitrary sentence w in first order logic, whether w is valid (i.e., true in all interpretations)?”
- “Given a set of axioms A and a sentence w , does there exist an algorithm to decide whether w is entailed by A ?” Note that this formulation is equivalent to the first one since the sentence $A \rightarrow w$ is valid iff w is entailed by A .
- “Given a set of axioms A and a sentence w , does there exist an algorithm to decide whether w can be proved from A ?” Note that this formulation is equivalent to the second one since Gödel’s Completeness Theorem tells us that there exists, for first-order logic, an inference procedure that is powerful enough to derive, from A , every sentence that is entailed by A .

Note that questions 1 and 2 (i.e., “Can the facts be axiomatized?” and “Can theoremhood be decided?”), while related, are different in an important way. The fact that the answer to question 1 is no does not obviously imply that the answer to the Entscheidungsproblem is no. While some true statements are not theorems, it might still have turned out to be possible to define an algorithm that distinguishes theorems from nontheorems.

The Entscheidungsproblem had captured the attention of several logicians of the time, including Alan Turing and Alonzo Church. Turing and Church, working independently, realized that, in order to solve the Entscheidungsproblem, it was necessary first to formalize what was meant by an algorithm. Turing’s formalization was what we now call a Turing machine. Church’s formalization was the *lambda calculus*, which we will discuss briefly below. The two formalizations look very different. But Turing showed that they are equivalent in power. Any problem that can be solved in one can be solved in the other. As it turns out ([Turing 1936] and [Church 1936]), the Entscheidungsproblem can be solved in neither. We’ll see why this is so in Chapter 19.

But out of the negative results that formed the core of the Church and Turing papers emerged an important new idea: Turing machines and the lambda calculus are equivalent. Perhaps that observation can be extended.

The *Church-Turing thesis*, or sometimes just *Church’s thesis*, states that all formalisms powerful enough to describe everything we think of as a computational algorithm are equivalent.

We should point out that this statement is stronger than anything that either Church or Turing actually said. This version is based on a substantial body of work that has occurred since Turing and Church’s seminal papers. Also note

that we have carefully used the word *thesis* here, rather than *theorem*. There exists no proof of the Church-Turing thesis because its statement depends on our informal definition of a computational algorithm. It is in principle possible that someone may come up with a more powerful model. Many very different models have been proposed over the years. We will examine a few of them below. All have been shown to be no more powerful than the Turing machine.

The Church-Turing thesis is significant. In the next several chapters, we are going to prove that there are important problems whose solutions cannot be computed by any Turing machine. The Church-Turing thesis tells us that we should not expect to find some other reasonable computational model in which those same problems can be solved. Moreover, the equivalence proofs that support the thesis tell us that it is certain that those problems cannot be solved in any of the computational models that have so far been considered and compared to the Turing machine.

18.2 Examples of Equivalent Formalisms ♦

All of the following models have been shown to be equivalent to our basic definition of a Turing machine:

- Modern computers, if we assume that there is an unbounded amount of memory available.
- Lambda calculus.
- Recursive functions (in which the class of computable functions is built from a small number of primitive functions and a small set of combining operations).
- Tag systems (in which we augment an FSM with a FIFO queue rather than a stack).
- Unrestricted grammars (in which we remove the constraint that the left-hand side of each production must consist of just a single nonterminal).
- Post production systems (in which we allow grammar-like rules with variables).
- Markov algorithms.
- Conway's Game of Life.
- One dimensional cellular automata.
- Various theoretical models of DNA-based computing.
- Lindenmayer systems.

We will describe recursive functions in Chapter 25, unrestricted grammars in Chapter 23, and Lindenmayer systems (also called L-systems) in Section 24.4. In the remainder of this chapter and we will briefly discuss the others.

18.2.1 Modern Computers

We showed in Section 17.4 that the functionality of modern “real” computers can be implemented with Turing machines. This observation suggests a slightly different way to define the decidable languages (i.e., those that are in D). A language L is decidable if there exists a decision procedure for it.

18.2.2 Lambda Calculus

Alonzo Church developed the lambda calculus λ as a way to formalize the notion of an algorithm. While Turing's solution to that same problem has the feel of a procedure, Church's solution feels more like a mathematical specification.

The lambda calculus is the basis for modern functional programming languages like Lisp, Scheme, ML, and Haskell. © 671.

The lambda calculus is an expression language. Each expression defines a function of a single argument, which is written as a variable bound by the operator λ . For example, the following simple lambda calculus expression describes the successor function:

$$(\lambda x. x + 1).$$

Functions can be applied to arguments by binding each argument to a formal parameter. So:

$$(\lambda x. x + 1) 3.$$

is evaluated by binding 3 to x and computing the result, 4.

Functions may be arguments to other functions and the value that is computed by a function may be another function. One of the most common uses of this feature is to define functions that we may think of as taking more than one argument. For example, we can define a function to add two numbers by writing:

$$(\lambda x. \lambda y. x + y)$$

Function application is left associative. So we can apply the addition function that we just described by writing, for example:

$$(\lambda x. \lambda y. x + y) 3 4$$

This expression is evaluated by binding 3 to x to create the new function $(\lambda y. 3 + y)$, which is then applied to 4 to return 7.

In the pure lambda calculus, there is no built-in data type number. All expressions are functions. But the natural numbers can be defined as lambda calculus functions. So the lambda calculus can effectively describe numeric functions just as we have done.

The lambda calculus can be shown to be equivalent in power to the Turing machine. In other words, the set of functions that can be defined in the lambda calculus is equal to the set of functions that can be computed by a Turing machine. Because of this equivalence, any problem that is undecidable for Turing machines is also undecidable for the lambda calculus. For example, we'll see in Chapter 21 that it is undecidable whether two Turing machines are equivalent. It is also undecidable whether two expressions in the lambda calculus are equivalent. In fact, Church's proof of that result was the first formal undecidability proof. (It appeared months before Turing's proof of the undecidability of questions involving Turing machines.)

18.2.3 Tag Systems

In the 1920's, a decade or so before the pioneering work of Gödel, Turing, and Church was published, the Polish logician Emil Post began working on the decidability of logical theories. Out of his work emerged two formalisms that are now known to be equivalent to the Turing machine. We'll mention the first, tag systems, here and the second, Post production systems, in the next section.

Post, and others, defined various versions (with differing restrictions on the alphabet and on the form of the operations that are allowed) of the basic tag system architecture. We describe the simplest here: A **tag system**, sometimes now called a **Post machine**, is a finite state machine that is augmented with a first-in, first out (FIFO) queue. In other words, it's a PDA with a FIFO queue rather than a stack.

It is easy to see that there are languages that are not context-free (and so cannot be accepted by any PDA) but that can be accepted by a tag system. Recall that while $\text{PalEven} = \{ww^R : w \in \{a, b\}^*\}$ is context-free, its cousin, $\text{WW} = \{ww : w \in \{a, b\}^*\}$, in which the second half of the string is not reversed, is not context-free. We could not build a PDA for WW because, using a stack, there was no way to compare the characters in the second half of a string to the characters in the first half except by reversing them. If we can use a FIFO queue instead of a stack, we no longer have this problem. So a simple tag system to accept WW writes the first half of its input string into its queue and then removes characters from the head of the queue, one at a time, and checks each of them against the characters in the second half of the input string.

But have we simply traded one set of languages for another? Or can we build a tag system to accept PalEven as well as WW? The answer is that, while there is not a simple tag system to accept PalEven, there is a tag system. In fact, any language that can be accepted by a Turing machine can also be accepted by a tag system. To see why, we'll sketch a technique for simulating a Turing machine with a tag system. Let the tag system's queue correspond to the Turing machine's active tape plus a blank on either side and let the head of the tag system's queue contain the square that is under the Turing machine's read/write head.

Now we just need a way to move both left and right in the queue, which would be easy if the tag system's queue were a loop (i.e., if its front and back were glued together). It isn't a loop, but we can treat it as though it were. To simulate a Turing machine that moves its head one square to the right, remove the symbol at the head of the queue and add it to the tail. To simulate a Turing machine that moves its head one square to the left, consider a queue that contains n symbols. One at a time, remove the first $n-1$ symbols from the head of the queue and add them to the tail. To simulate a Turing machine that moves onto the blank region of its tape, exploit the fact that a tag system is allowed to push more than one symbol onto the end of its queue. So push two, one of which corresponds to the newly nonblank square.

18.2.4 Post Production Systems

We next consider a second formalism that is derived from Post's early work. This one is based on the idea of a rewrite or production or rule-based system. A Post production system (or simply Post system), as such systems have come to be known (although Post never called them that), shares with the grammar formalisms that we have considered the property that computation is accomplished by applying a set of production rules whose left-hand sides are matched against a current working string and whose right-hand sides are used to rewrite the working string.

Post's early work inspired the development of many modern rule-based systems, including context-free grammars described in BNF, § 664, rule-based expert systems § 771, production rule-based cognitive architectures § 773, and rule-based specifications for the behavior of NPCs in interactive games, § 791.

Based on the ideas described in Post's work, we define a *Post system* P to be a quintuple (V, Σ, X, R, S) , where:

- V is the rule alphabet, which contains nonterminal and terminal symbols,
- Σ (the set of terminals) is a subset of V ,
- X is a set of variables whose values are drawn from V^* ,
- R (the set of rules) is a finite subset of $(V \cup X)^* \times (V \cup X)^*$, with the additional constraint that every variable that occurs on the right-hand side of a rule must also have occurred on the left-hand side, and
- S (the start symbol) can be any element of $V - \Sigma$.

There are three important differences between Post systems, as just defined, and both the regular and context-free grammar formalisms that we have already considered:

1. In a Post system, the left-hand side of a rule may contain two or more symbols.
2. In a Post system, rules may contain variables. When a variable occurs on the left-hand side of a rule, it may match any element of V^* . When a variable occurs on the right-hand side, it will generate whatever value it matched.
3. In a Post system, a rule may be applied only if its left-hand side matches the entire working string. When a rule is applied, the entire working string is replaced by the string that is specified by the rule's right-hand side. Note that this contrasts with the definition of rule application that we use in our other rule-based formalisms. In them, a rule may match any substring of the working string and just that substring is replaced as directed by the rule's right-hand side. So, suppose that we wanted to write a rule $A \rightarrow B$ that replaced an A anywhere in the string with a B . We would have to write instead the rule $XAY \rightarrow XBY$. The variables X and Y can match everything before the A and after it, respectively.

As with regular and context-free grammars, let $x \Rightarrow_P y$ mean that the string y can be derived from the string x by applying a single rule in R_P . Let $x \Rightarrow_{P^*} y$ mean that y can be derived from x by applying zero or more rules in R_P . The language generated by P , denoted $L(P)$ is $\{w \in \Sigma^* : S \Rightarrow_{P^*} w\}$.

Example 18.1 A Post System for WW

Recall the language $WW = \{ww : w \in \{a, b\}^*\}$, which is in D (i.e., it is decidable) but is not context-free. We can build a Post system P that generates WW . $P = (\{S, a, b\}, \{a, b\}, \{X\}, R, S)$, where $R =$

- (1) $XS \rightarrow XaS$ /* Generate $(a \cup b)^* S$.
- (2) $XS \rightarrow XbS$ /*
- (3) $XS \rightarrow XX$ /* Create a second copy of X .

This Post system can generate, for example, the string `abbabb`. It does so as follows:

```

S => (using rule (1) and letting X match ε)
aS => (using rule (2) and letting X match a)
abS => (using rule (2) and letting X match ab)
abbS => (using rule (3) and letting X match abb)
abbabb

```

Post systems, as we have just defined them, are equivalent in power to Turing machines. The set of languages that can be generated by a Post system is exactly *SD*, the set of semidecidable languages. The proof of this claim is by construction. For any Post system P , it is possible to build a Turing machine M that simulates P . And, for any Turing machine M , it is possible to build a Post system P that simulates M .

18.2.5 Unrestricted Grammars

While the availability of variables in Post systems is convenient, variables are not actually required to give Post systems their power. In Chapter 23, we will describe another formalism that we will call an unrestricted grammar. The rules in an unrestricted grammar may not contain variables, but their left-hand sides may contain any number of terminal and nonterminal symbols, subject to the sole constraint that there be at least one symbol. Unrestricted grammars have exactly the same power as do Post systems and Turing machines. They can generate exactly the semidecidable (*SD*) languages. In Example 23.3 we'll show an unrestricted grammar that generates *WW* (the language we considered above in Example 18.1).

18.2.6 Markov Algorithms

Next we consider yet another formalism based on rewrite rules. A Markov algorithm \equiv (named for its inventor, Andrey A. Markov, Jr., the son of the inventor of the stochastic Markov model that we described in Section 5.11.1), is simply an ordered list of rules, each of which has a left-hand side that is a single string and a right-hand side that is also a single string. Formally a Markov algorithm M is a triple (V, Σ, R) , where:

- V is the rule alphabet, which contains both working symbols and input symbols. Whenever the job of M is to semidecide or decide a language (as opposed to compute a function), V will contain two special working symbols, *Accept* and *Reject*.
- Σ (the set of input symbols) is a subset of V , and
- R (the rules) is an ordered list of rules, each of which is an element of $V^* \times V^*$. There are two kinds of rules, continuing and terminating. Whenever a terminating rule is applied, the algorithm halts. We will write continuing rules, as usual, as $X \rightarrow Y$. We will write terminating rules by adding a dot after the arrow. So we will have $X \rightarrow \bullet Y$.

Notice that there is no start symbol. Markov algorithms, like Turing machines, are given an input string. The job of the algorithm is to examine its input and return the appropriate result.

The rules are interpreted by the following algorithm:

Markovalgorithm(M : Markov algorithm, w : input string) =

1. Until no rules apply or the process has been terminated by executing a terminal rule do:
 - 1.1. Find the first rule in the list R that matches against w . If that rule matches w in more than one place, choose the leftmost match.
 - 1.2. If no rule matches then exit.
 - 1.3. Apply the matched rule to w by replacing the substring that matched the rule's left-hand side with the rule's right-hand side.
 - 1.4. If the matched rule is a terminating rule, exit.
2. If w contains the symbol *Accept* then accept.
3. If w contains the symbol *Reject* then reject.
4. Otherwise, return w .

Notice that a Markov algorithm (unlike a program in any of the other rule-based formalisms that we have considered so far) is completely deterministic. At any step, either no match exists, in which case the algorithm halts, or exactly one match can be selected.

The logic programming language Prolog executes programs (sets of rules) in very much the same way that the Markov algorithm interpreter does. Programs are deterministic and programmers control the order in which rules are applied by choosing the order in which to write them $\text{C } 762$.

The Markov algorithm formalism is equivalent in power to the Turing machine. This means that Markov algorithms can semidecide exactly the set of SD languages (in which case they may accept or reject) and they can compute exactly the set of computable functions (in which case they may return a value). The proof of this claim is by construction: It is possible to show that a Markov algorithm can simulate the universal Turing machine U , and vice versa.


Example 18.2 A Markov Algorithm for $A^nB^nC^n$

We show a Markov algorithm M to decide the language $A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$. Let $M = (\{a, b, c, \#, \%, ?, \text{Accept}, \text{Reject}\}, \{a, b, c\}, R)$, where $R =$

- | | | |
|------|---|--|
| (1) | $\#a \rightarrow \%$ | /* If the first character is an a, erase it and look for a b next. |
| (2) | $\#b \rightarrow \bullet \text{Reject}$ | /* If the first character is a b, reject. |
| (3) | $\#c \rightarrow \bullet \text{Reject}$ | /* If the first character is a c, reject. |
| (4) | $\%a \rightarrow a\%$ | /* Move the % past the a's until it finds a b. |
| (5) | $\%b \rightarrow ?$ | /* If it finds a b, erase it and look for a c next. |
| (6) | $\% \rightarrow \bullet \text{Reject}$ | /* No b found. Just c's or end of string. Reject. |
| (7) | $?b \rightarrow b?$ | /* Move the ? past the b's until it finds a c. |
| (8) | $?c \rightarrow \varepsilon$ | /* If it finds a c, erase it. Then only rule (11) can fire next. |
| (9) | $? \rightarrow \bullet \text{Reject}$ | /* No c found. Just a's or b's or end of string. Reject. |
| (10) | $\# \rightarrow \bullet \text{Accept}$ | /* A # was created but there are no input characters left. Accept. |
| (11) | $\varepsilon \rightarrow \#$ | /* This one goes first since none of the others can. |

When M begins, the only rule that can fire is (11), since all the others must match some working symbol. So rule (11) matches at the far left of the input string and adds a # to the left of the string. If the first input character is an a, it will be picked up by rule (1), then erased and replaced by a new working symbol %. The job of the % is to sweep past any other a's and find the first b. If there is no b or if a c comes first, M will reject. If there is a b, it will be picked up by rule (5), then erased and replaced by a third working symbol ?, whose job is to sweep past any remaining b's and find the first c. If there is no c, M will reject. If there is, it will be erased by rule (8). At that point, there are no remaining working symbols, so the only thing that can happen is that rule (11) fires and the process repeats until all matched sets of a's, b's, and c's have been erased. If that happens, the final # that rule (11) adds will be the only symbol left. Rule (10) will fire and accept.

18.2.7 Conway's Game of Life

The Game of Life  was first proposed by John Conway. In the game, the board (the world) starts out in some initial configuration in which each square is either alive (shown in black) or dead (shown in white). A simple example is shown in Figure 18.1.

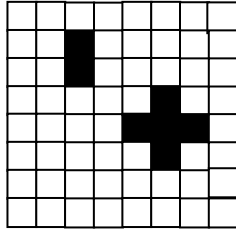



Figure 18.1 An example of the Game of Life

Life is not a game in the usual sense of having players. It is more like a movie that we can watch. It proceeds in discrete steps. At each step, the value for each cell is determined by computing the number of immediate neighbors (including the four on the diagonals, so up to a maximum of eight) it currently has, according to the following rules:

- A dead cell with exactly three live neighbors becomes a live cell (birth).
- A live cell with two or three live neighbors stays alive (survival).
- In all other cases, a cell dies or remains dead (overcrowding or loneliness).


Once values for all the cells at the next step have been determined, all of them change values simultaneously. Then the next step begins.

Life is fascinating to watch .

Life can be played on a board of any size and it can be given any desired starting configuration. Depending on the starting configuration, Life may end (all the cells die), it may reach some other stable configuration (it looks the same from one step to the next), or it may enter a cycle of configurations. We'll say that the game of Life halts iff it reaches some stable configuration.

We can imagine the Life simulator as a computing device that takes the initial board configuration as input, knows one operation (namely how to move from one configuration to the next), may or may not halt, and if it halts, produces some stable configuration as its result. Conway and others have shown that, with an appropriate encoding of Turing machines and input strings as board configurations, the operation of any Turing machine can be simulated by the game of Life. And a Life simulator can be written as a Turing machine. So Life is equivalent in power to a Turing machine.

18.2.8 One Dimensional Elementary Cellular Automata

The game of Life can be thought of as a two-dimensional cellular automaton. Each square looks at its neighboring cells in two dimensions to decide what should happen to it at the next step. But we don't need two dimensions to simulate a Turing machine. Wolfram [2002] describes one-dimensional cellular automata  that look like the one shown in Figure 18.2. As in the game of Life, each cell is either on or off (black or white), an initial configuration is specified, and the configuration of the automaton at each later step t is determined by independently computing the value for each cell, which in turn is a function solely of the values of itself and its neighbors (in this case two) at step $t - 1$.

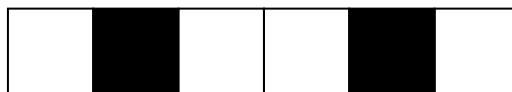


Figure 18.2 A one-dimensional cellular automaton

In the game of Life, Conway specified the rule that is to be used to compute the value of each cell at the next step. What rule shall we use for these one-dimensional automata? Since each cell can have one of the two values (black or white) and each cell's next configuration depends on the current configuration of three cells (itself and its two neighbors), there are 256 (2^8) rules that we could use. Each rule contains 8 (2^3) parts, specifying what should happen next for each of the 8 possible current situations. Figure 18.3 shows the rule that Wolfram numbers 110. Wolfram describes a proof that Rule 110, with an appropriate (and complex) encoding of Turing machines and strings as cellular automata, is equivalent in power to the Turing machine.

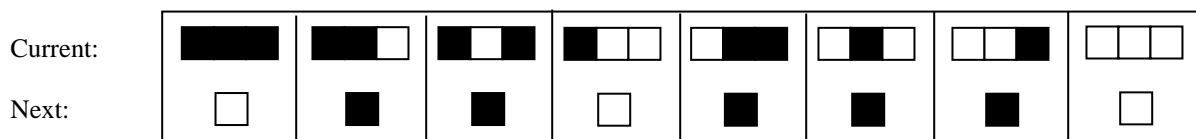


Figure 18.3 Rule 110

18.2.9 DNA Computing

See § 727 for a very short introduction to molecular biology and genetics.

In 1993, Len Adleman observed that DNA molecules and Turing machine tapes both do the same thing: they encode information as strings. Further, he observed that both nature and the Turing machine offer simple operations for manipulating those strings. So he wondered: can DNA compute? To begin to answer that question, he performed a fascinating experiment . In a laboratory, he solved an instance of the Hamiltonian path problem⁹ using DNA molecules. More precisely, what he did was the following:

1. He chose a particular graph G (with 7 vertices and 14 edges).
2. He encoded each vertex of G as a sequence of 8 nucleotides. For example, a vertex might be represented as ACCTGCAG.
3. He encoded each directed edge of G as a sequence of 8 nucleotides, namely the last four from the encoding of the start vertex and the first four from the encoding of the end vertex. So, for example, if there was an edge from ACTTGCAG to TCGGACTG, then it would be encoded as GCAGTCGG.
4. He synthesized many copies of each of the edge sequences, as well as many copies of the DNA complements¹⁰ of all the vertex encodings. So for example, since one of the vertices was encoded as ACTTGCAG, its complement, the sequence TGAACGTC, was synthesized.

⁹ The definition that Adleman uses for the Hamiltonian path problem is the following: let G be a directed graph, with one node s designated as the start node and another node d designated as the end node. A Hamiltonian path through G is a path that begins at s , ends at d , and visits each other node in G exactly once. A Hamiltonian path problem is then the following decision problem: given a directed graph G , with designated s and d , does there exist a Hamiltonian path through it? We will return to this problem in Part V. There we will use a slightly different definition that asks for any Hamiltonian path through G . It will not specify a particular start and end vertex.

¹⁰ Each DNA molecule is a double strand of nucleotide sequences. Each nucleotide contains one of the four bases: adenine (A), thymine (T), guanine (G) and cytosine (C). Each of these has a complement: C and G are complements and A and T are complements. When a double strand of DNA is examined as a sequence of base pairs (one from each

5. He combined the vertex-complement molecules and the edge molecules in a test tube, along with water, salt, some important enzymes, and a few other chemicals required to support the natural biological processes.
6. He allowed to happen the natural process by which complementary strands of DNA in solution will meet and stick together (anneal). So for example, consider again the edge GCAGTCGG. It begins at the vertex whose encoding is ACTTGCAG and it ends at a vertex whose encoding is TCGGACTG. The complements of those vertices are TGAACGTC and AGCCTGAC. So, in solution, the edge strands will anneal with the vertex-complement strands to produce the double strand:

path of length one (i.e., one edge): GCAGTCGG
 complement of sequence of two vertices: TGAACGTCAGCCTGAC

But then, suppose that there is an edge from the second vertex to some third one. Then that edge will anneal to the lower string that was produced above, generating:

path of length two: GCAGTCGGACTGGGCT
 complement of sequence of two vertices: TGAACGTCAGCCTGAC

Then a third vertex may anneal to the right end of the path sequence. And so forth. Eventually, if there is a path from the start vertex to the end one, there will be a sequence of fragments, like our top one, that corresponds to that path.

7. He allowed a second biological reaction to occur. The enzyme ligase that had been added to the mixture joins adjacent sequences of DNA. So instead of strands of fragments, as above, the following strands will be produced:

path of length two: GCAGTCGGACTGGGCT
 complement of sequence of three vertices: TGAACGTCAGCCTGACCCGATACA

8. He used the polymerase chain reaction (PCR) technique to make massive numbers of copies of exactly those sequences that started at the start vertex and ended at the end one. Other sequences were still present in the mix after this step, but in much lower numbers.
9. He used gel electrophoresis to select only those molecules whose length corresponded to a Hamiltonian path through the graph.
10. He checked that each of the vertices other than the source and the destination did in fact occur in the selected molecules. To do this required one pass through the following procedure for each intermediate vertex:
 - 10.1. Use a DNA “probe” that attracts molecules that contain a particular DNA sequence (i.e., the one for the vertex that is being checked).
 - 10.2. Use a magnet to attract the probes.
 - 10.3. Throw away the rest of the solution, thus losing those molecules that were not attached to the probe.
11. He checked that some DNA molecules remained at the end. Only molecules that corresponded to paths that started at the start vertex, ended at the end vertex, had the correct length for a path that visited each vertex exactly once, and contained each of the vertices could still be present. So if any DNA was left, a Hamiltonian path existed.

Since that early experiment, other scientists have tried other ways of encoding information in DNA molecules and using biological operations to compute with it [\[4\]](#). The question then arises: is DNA computing Turing-equivalent? The answer depends on exactly what we mean by DNA computing. In particular, what operations are allowed? For example, must the model be limited only to operations that can be performed only by naturally occurring enzymes?

strand), every base occurs across from its complement. So, whenever one strand has a C, the other has a G. And whenever one strand has an A, the other has a T.

It has been shown that, given some reasonable assumptions about allowed operations, DNA computing is Turing-equivalent.

18.3 Exercises

- 1) Church's Thesis makes the claim that all reasonable formal models of computation are equivalent. And we showed in, Section 17.4, a construction that proved that a simple accumulator/register machine can be implemented as a Turing machine. By extending that construction, we can show that any computer can be implemented as a Turing machine. So the existence of a decision procedure (stated in any notation that makes the algorithm clear) to answer a question means that the question is decidable by a Turing machine.

Now suppose that we take an arbitrary question for which a decision procedure exists. If the question can be reformulated as a language, then the language will be in D iff there exists a decision procedure to answer the question. For each of the following problems, your answers should be a precise description of an algorithm. It need not be the description of a Turing Machine:

- a) Let $L = \{ \langle M \rangle : M \text{ is a DFSM that doesn't accept any string containing an odd number of 1's} \}$. Show that L is in D .
 - b) Let $L = \{ \langle E \rangle : E \text{ is a regular expression that describes a language that contains at least one string } w \text{ that contains } 111 \text{ as a substring} \}$. Show that L is in D .
 - c) Consider the problem of testing whether a DFSM and a regular expression are equivalent. Express this problem as a language and show that it is in D .
- 2) Consider the language $L = \{ w = xy : x, y \in \{a, b\}^* \text{ and } y \text{ is identical to } x \text{ except that each character is duplicated} \}$. For example $ababaabbaabb \in L$.
 - a) Show that L is not context-free.
 - b) Show a Post system (as defined in Section 18.2.4) that generates L .
 - 3) Show a Post system that generates $A^n B^n C^n$.
 - 4) Show a Markov algorithm (as defined in Section 18.2.6) to subtract two unary numbers. For example, on input 111-1, it should halt with the string 11. On input 1-111, it should halt with the string -11.
 - 5) Show a Markov algorithm to decide WW .
 - 6) Consider Conway's Game of Life, as described in Section 18.2.7. Draw an example of a simple Life initial configuration that is an oscillator, meaning that it changes from step to step but it eventually repeats a previous configuration.

19 The Unsolvability of the Halting Problem

So far, we have focused on solvable problems and we have described an increasingly powerful sequence of formal models for computing devices that can implement solutions to those problems. Our last attempt is the Turing machine and we've shown how to use Turing machines to solve several of the problems that were not solvable with a PDA or an FSM. The Church-Turing thesis suggests that, although there are alternatives to Turing machines, none of them is any more powerful. So, are we done? Can we build a Turing machine to solve any problem we can formally describe?

Until a bit before the middle of the 20th century, western mathematicians believed that it would eventually be possible to prove any true mathematical statement and to define an algorithm to solve any clearly stated mathematical problem. Had they been right, our work would be done. But they were wrong. And, as a consequence, the answer to the question in the last paragraph is no. There are well-defined problems for which no Turing machine exists.

In this chapter we will prove our first result that shows the limits of what we can compute. In later chapters, we will discuss other unsolvable problems and we will see how to analyze new problems and then prove either that they are solvable or that they are not. We will do this by showing that there are languages that are not in decidable (i.e., they are not in D). So, recall the definitions of the sets D and SD that we presented in Chapter 17:

- A Turing machine M with input alphabet Σ **decides** a language $L \subseteq \Sigma^*$ (or, alternatively, implements a decision procedure for L) iff, for any string $w \in \Sigma^*$:

- If $w \in L$ then M accepts w , and
- If $w \notin L$ then M rejects w .

A language L is **decidable** (and thus an element of D) iff there is a Turing machine M that decides it.

- A Turing machine M with input alphabet Σ **semidecides** a language $L \subseteq \Sigma^*$ (or, alternatively, implements a semidecision procedure for L) iff for any string $w \in \Sigma^*$:

- If $w \in L$ then M accepts w , and
- If $w \notin L$ then M does not accept w . (Note that M may fail to accept either by rejecting or by failing to halt.)

A language L is **semidecidable** (and thus an element of SD) iff there is a Turing machine that semidecides it.

Many of the languages that we are about to consider are composed of strings that correspond, at least in part, to encodings of Turing machines. Some of them may also contain other fragments. So we will be considering languages such as:

- $L_1 = \{ \langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w \}$.
- $L_2 = \{ \langle M \rangle : \text{there exists no string on which Turing machine } M \text{ halts} \}$.
- $L_3 = \{ \langle M_a, M_b \rangle : M_a \text{ and } M_b \text{ are Turing machines that halt on the same strings} \}$.

Recall that $\langle M \rangle$ is the notation that we use for the encoding of a Turing machine M using the scheme described in Section 17.6. $\langle M, w \rangle$ means the encoding of a pair of inputs: a Turing machine M and an input string w . $\langle M_a, M_b \rangle$ means the encoding of a pair of inputs, both of which are Turing machines.

Consider L_1 above. It consists of the set of strings that encode a (Turing machine, string) pair with the property that the Turing machine M , when started with w on its tape, halts. So, in order for some string s to be in language L_1 , it must possess two properties:

- It must be syntactically well-formed, and
- It must encode a machine M and a string w such that M would halt if started on w .

We will be attempting to find Turing machines that can decide (or semidecide) languages like L_1 , L_2 , and L_3 . Building a Turing machine to check for syntactic validity is easy. We would like to focus on the other part. So, in our discussion of languages such as these, we will define the universe from which we are drawing strings to be the set that contains only those strings that meet the syntactic requirements of the language definition. For example, that could be the set that contains descriptions of Turing machines (strings of the form $\langle M \rangle$), or the set that contains descriptions of a Turing machine and a string (strings of the form $\langle M, w \rangle$). This contrasts with the convention we have been using up until now, in which the universe was Σ^* , where Σ is the alphabet over which L is defined. This change in convention will be important whenever we talk about the complement of a language such as L_1 , L_2 , or L_3 . So, for example, we have:

$$\neg L_1 = \{ \langle M, w \rangle : \text{Turing machine } M \text{ does not halt on input string } w \}.$$

Note that this convention has no impact on the decidability of any of these languages since the set of syntactically valid strings is in D . So it is straightforward to build a precondition checker that accepts exactly the syntactically well-formed strings and rejects all others.

19.1 The Language H is Semidecidable but Not Decidable

We begin by considering the language we called L_1 in the last section. We're now going to call it H , the halting problem language. So, define:

- $H = \{ \langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w \}.$

H is:

- easy to state and to understand.
- of great practical importance since a program to decide H could be a very useful part of a program-correctness checker. You don't want to go online to pay a bill and have the system go into an infinite loop after it has debited your bank account and before it credits the payment to your electric bill.
- semidecidable.
- not decidable.

We need to prove these last two claims. Before we attempt to do that, let's consider them. H would be decidable if there existed an algorithm that could take as input a program M and an input w and decide whether M will halt on w . It is easy to define such an algorithm that works some of the time. For example, it would be easy to design an algorithm that could discover that the following program (and many others like it that contain no loops) halts on all inputs:

1. Concatenate 0 to the end of the input string.
2. Halt.

It would also be easy to design an algorithm that could discover that the following program (and many others like it) halts on no inputs:

1. Concatenate 0 to the end of the input string.
2. Move right one square.
3. Go to step 1.

But, for H to be decidable, we would need an algorithm that decides the question in all cases. Consider the following program:

```
times3(x: positive integer) =
  While  $x \neq 1$  do:
    If  $x$  is even then  $x = x/2$ .
    Else  $x = 3x + 1$ .
```

It is easy to prove that *times3* halts on any positive integer that is a power of 2. In that case, x decreases each time through the loop and must eventually hit 1. But what about other inputs? Will it halt, for example on 23,478? It is

conjectured that, for any positive integer input, the answer to this question is yes. But, so far, no one has been able either to prove that conjecture or to find a counterexample. The problem of determining whether *times3* must always halt is called the **3x+1 problem** ■.

So there appear to be programs whose halting behavior is difficult to determine. We now prove that the problem of deciding halting behavior for an arbitrary (machine, input) pair is semidecidable but not decidable.

Theorem 19.1 Semidecidability of the Halting Problem

Theorem: The language $H = \{ \langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w \}$ is semidecidable.

Proof: The proof is by construction of a semideciding Turing machine M_{SH} . The design of M_{SH} is simple. All it has to do is to run M on w and accept if M halts. So:

$M_{SH}(\langle M, w \rangle) =$

1. Run M on w .
2. Accept.

M_{SH} accepts iff M halts on w . Thus M_{SH} semidecides H . ■

But H is not decidable. This single fact is going to turn out to be the cornerstone of the entire theory of undecidability that we will discuss in the next several chapters.

Compilers check for various kinds of errors in programs. But, because H is undecidable, no compiler can offer a guarantee that a program is free of infinite loops. © 669.

Theorem 19.2 Undecidability of the Halting Problem

Theorem: The language $H = \{ \langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w \}$ is not decidable.

Proof: If H were decidable, then there would be some Turing machine M_H that decided it. M_H would implement the following specification:

$halts(\langle M: \text{string}, w: \text{string} \rangle) =$
 If $\langle M \rangle$ is the description of a Turing machine that halts on input w , then accept; else reject.

Note that we have said nothing about how M_H would work. It might use simulation. It might examine M for loops. It might use a crystal ball. The only claim we are making about M_H is that it can implement *halts*. In other words, it can decide somehow whether M halts on w and report *True* if it does and *False* if it does not.

Now suppose that we write the specification for a second Turing machine, which we'll call *Trouble*:

$Trouble(x: \text{string}) =$
 If *halts* accepts $\langle x, x \rangle$, then loop forever; else halt.

If there exists some M_H that computes the function *halts*, then the Turing machine *Trouble* also exists. We can easily write the code for it as follows: Assume that $C_{\#}$ is a Turing machine (similar to the copy machine that we showed in Example 17.11) that writes onto its tape a second copy of its input, separated from the first by a comma. Also assume that M_H exploits the variable r , into which it puts 1 if it is about to halt and accept and 0 if it is about to halt and reject. Then, using the notation defined in Section 17.1.5, *Trouble* is shown in Figure 19.1.

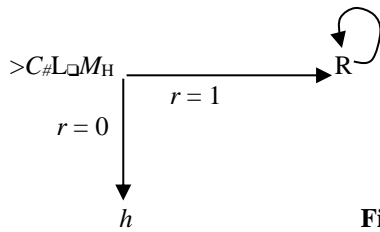


Figure 19.1 A Turing machine that implements the function *Trouble*

Trouble takes a single string x as its input. It makes a copy of that string, moves its read/write head all the way back to the left, and then invokes M_H on x . M_H will treat the first copy as a Turing machine and the second one as the input to that Turing machine. When M_H halts (which it must, since we've assumed that it is a deciding machine), *Trouble* will either halt immediately or loop forever, depending on whether M_H stored a 0 or a 1 in r .

What happens if we now invoke *Trouble*($\langle \textit{Trouble} \rangle$)? In other words, we invoke *Trouble* on the string that corresponds to its own description, as shown above. Then *Trouble* will invoke $M_H(\langle \textit{Trouble}, \textit{Trouble} \rangle)$. Since the second argument of M_H can be any string, this is a valid invocation of the function. What should M_H say?

- If M_H reports that *Trouble*($\langle \textit{Trouble} \rangle$) halts (by putting a 1 in the variable r), then what *Trouble* actually does is to loop.
- But if M_H reports that *Trouble*($\langle \textit{Trouble} \rangle$) does not halt (by putting a 0 in the variable r), then what *Trouble* actually does is to halt.

Thus there is no response that M_H can make that accurately predicts the behavior of *Trouble*($\langle \textit{Trouble} \rangle$). So we have found at least one input on which any implementation of *halts* must fail to report the correct answer. Thus there exists no correct implementation of *halts*. This means that M_H does not exist. So H is not decidable. ■

There is another way to state this proof that makes it clearer that what we have just done is to use diagonalization. Consider Table 19.1. To form column 0, we lexicographically enumerate all Turing machines, using the procedure that was defined in Section 17.6.2. To form row 0, we lexicographically enumerate all possible input strings over the alphabet Σ that we used to encode inputs to the universal Turing machine. The cell $[i, j]$ of the table contains the value 1 if TM_i halts on the j^{th} input string and is blank otherwise.

	i_1	i_2	i_3	...	$\langle \textit{Trouble} \rangle$...
<i>machine</i> ₁	1					
<i>machine</i> ₂		1				
<i>machine</i> ₃					1	
...				1		
<i>Trouble</i>			1			1
...	1	1	1			
...				1		

Table 19.1 Using diagonalization to construct *Trouble*

This table is infinite in both directions, so it will never be explicitly constructed. But, if we claim that the Turing machine M_H exists, we are claiming that it can compute the correct value for any cell in this table on demand. *Trouble* must correspond to some row in the table and so, in particular, M_H must be able to compute the values for that row. The string $\langle \textit{Trouble} \rangle$ must correspond to some column in the table. What value should occur in the grey cell of the picture? There is no value that correctly describes the behavior of *Trouble*, since we explicitly constructed it to look at the grey cell and then do exactly the opposite of what that cell says.

So we have just proven (twice) a very important result that can be stated in any one of three ways:

- The language H is not decidable.
- The halting problem is unsolvable (i.e., there can exist no implementation of the specification we have given for the *halts* function).
- The membership problem for the SD languages (i.e., those that can be accepted by some Turing machine) is not solvable.

Recall that we have seen many times that any decision problem that we can state formally can be restated as a language recognition task. So it comes as no surprise that this one can. In the rest of this book, we will use whichever version of this result is clearer at each point.

19.2 Some Implications of the Undecidability of H

We now have our first example, H , of a language that is semidecidable (i.e., it is in SD) but that is not decidable (i.e., it is not in D). What we will see in the rest of this section is that H is far more than an anomaly. It is the key to the fundamental distinction between the classes D and SD.

Theorem 19.3 H is the Key to the Difference between D and SD

Theorem: If H were in D then every SD language would be in D.

Proof: Let L be any SD language. Since L is in SD, there exists a Turing machine M_L that semidecides it. Suppose H were also in D. Then it would be decided by some Turing machine that we can call O (for oracle). To decide whether some string w is in L , we can appeal to O and ask it whether M_L will halt on the input w . If the answer is yes, we can (without risk of getting into an infinite loop) run M_L on w and see whether or not it accepts. So, given M_L (the machine that semidecides L), we can build a new Turing machine M' that decides L by appeal to O :

$M'(w: \text{string}) =$

1. Run O on $\langle M_L, w \rangle$.
2. If O accepts (which it will iff M_L halts on w), then:
 - 2.1. Run M_L on w .
 - 2.2. If it accepts, accept. Else reject.
3. Else reject.

Since O is a deciding machine for H , it always halts. If it reports that M would halt on w , then M' can run M on w to see whether it accepts or rejects. If, on the other hand, O reports that M would not halt then it certainly cannot accept, so M' rejects. So M' always halts and returns the correct answer. Thus, if H were in D, all SD languages would be. ■

But H is not in D. And as we are about to see, it is not alone.

19.3 Back to Turing, Church, and the Entscheidungsproblem

At the beginning of Chapter 18, we mentioned that Turing invented the Turing machine because he was attempting to answer the question, “Given a set of axioms A and a sentence s , does there exist an algorithm to decide whether s is entailed by A ?” To do that, he needed a formal definition of an algorithm, which the Turing machine provided. As an historical aside, we point out here that in Turing’s model, machines (with the exception of a universal machine that could simulate other machines) were always started on a blank tape. So, while in H we ask whether a Turing machine M halts on some particular input w , Turing would ask simply whether it halts. But note that this is not a significant change. In our model, all inputs are of finite length. So it is possible to encode any particular input in the states of a machine that is to operate on it. That machine can start out with a blank tape, write the desired input on its tape, and then continue as though the tape had contained the input.

Having defined the Turing machine (which he called simply a “computing machine”), Turing went on to show the unsolvability of the halting problem. He then used that result to show that no solution to the Entscheidungsproblem can exist. An outline of Turing’s proof is the following:

1. If we could solve the problem of determining whether a given Turing machine ever prints the symbol 0, then we could solve the problem of determining whether a given Turing machine halts. Turing presented the technique by which this could be done.
2. But we can’t solve the problem of determining whether a given Turing machine halts, so neither can we solve the problem of determining whether it ever prints 0.
3. Given a Turing machine M , we can construct a logical formula F that is a theorem, given the axioms of Peano arithmetic, iff M ever prints the symbol 0. Turing also presented the technique by which this could be done.
4. If there were a solution to the Entscheidungsproblem, then we would be able to determine the theoremhood of any logical sentence and so, in particular, we could use it to determine whether F is a theorem. We would thus be able to decide whether M ever prints the symbol 0.
5. But we know that there is no procedure for determining whether M ever prints 0.
6. So there is no solution to the Entscheidungsproblem.

This proof is an example of the technique that we will use extensively in Chapter 21 to show that problems are not decidable. We reduce a problem that is already known not to be decidable to a new problem whose decidability is in question. In other words, we show that if the new problem were decidable by some Turing machine M , then we could use M as the basis for a procedure to decide the old problem. But, since we already know that no solution to the old problem can exist, no solution for the new one can exist either. The proof we just sketched uses this technique twice: once in steps 1 and 2 to show that we cannot solve the problem of determining whether a Turing machine ever prints the symbol 0, and a second time, in steps 3-6, to show that we cannot solve the Entscheidungsproblem.

19.4 Exercises

- 1) Consider the language $L = \{ \langle M \rangle : \text{Turing machine } M \text{ accepts at least two strings} \}$.
 - a) Describe in clear English a Turing machine M that semidecides L .
 - b) Now change the definition of L just a bit. Consider:

$$L' = \{ \langle M \rangle : \text{Turing machine } M \text{ accepts exactly 2 strings} \}.$$

Can you tweak the Turing machine you described in part a to semidecide L' ?

- 2) Consider the language $L = \{ \langle M \rangle : \text{Turing machine } M \text{ accepts the binary encodings of the first three prime numbers} \}$.
 - a) Describe in clear English a Turing machine M that semidecides L .
 - b) Suppose (contrary to fact, as established by Theorem 19.2) that there were a Turing machine *Oracle* that decided H . Using it, describe in clear English a Turing machine M that decides L .

20 Decidable and Semidecidable Languages

Now that we have shown that the halting problem is undecidable, it should be clear why we introduced the notion of a semidecision procedure. For some problems, it is the best we will be able to come up with. In this chapter we explore the relationship between the classes D and SD, given what we now know about the limits of computation.

20.1 D: The Big Picture

First, we observe that the class D includes the regular and the context-free languages. More precisely:

Theorem 20.1 All Context-Free Languages, Plus Others, are in D

Theorem: The set of context-free languages is a proper subset of D.

Proof: By Theorem 14.1, the membership problem for the context-free languages is decidable. So the context-free languages are a subset of D. And there is at least one language, $A^nB^nC^n$, that is decidable but not context-free. So the context-free languages are a *proper* subset of D. ■

20.2 SD: The Big Picture

Now what can we say about the relationship between D and the larger class SD? Almost every language you can think of that is in SD is also in D. Examples include:

- $A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$.
- $WCW = \{w c w : w \in \{a, b\}^*\}$.
- $WW = \{w w : w \in \{a, b\}^*\}$.
- $\{w \text{ of the form } x*y=z, \text{ where } x,y,z \in \{0, 1\}^* \text{ and, when } x, y, \text{ and } z \text{ are viewed as binary numbers, } x \cdot y = z\}$.

But there are languages that are in SD but not in D. We already know one:

- $H = \{\langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w\}$.

What about others? It isn't possible to come up with any physical examples since there are only finitely many molecules in the causal universe. So every physical set is finite and thus regular. But unless we want to model all our real world problems using only the power of a finite state machine, we generally ignore the fact that the true language is finite and model it as a more complex set that is unbounded and thus, for all practical purposes, infinite. If we do that, then here's a language that is effectively in SD and has the look and feel of many SD languages:

- $L = \{w : w \text{ is the email address of someone who will respond to a message you just posted to your newsgroup}\}$.

If someone responds, you know that their email address is in L . But if your best friend hasn't responded yet, you don't know that she isn't going to. All you can do is wait.

In Chapter 21 we will see that any question that asks about the result of running a Turing machine is undecidable (and so its corresponding language formulation is not in D). In a nutshell, if you can't think of a way to answer the question by simulating the Turing machine, it is very likely that there is no other way to do it and the question is undecidable. But keep in mind that we said that the question must ask about the *result of running* the Turing machine. Questions that ask simply about the Turing machine itself (e.g., how many states does it have) or about its behavior partway through its computation (e.g., what does it do after exactly 100 steps) are generally decidable.

In Chapter 18 we will see some examples of undecidable problems that do not ask questions about Turing machines. If you'd like to be convinced that this theory applies to more than the analysis of Turing machines (or of programs in general), skip ahead briefly to Chapter 18.

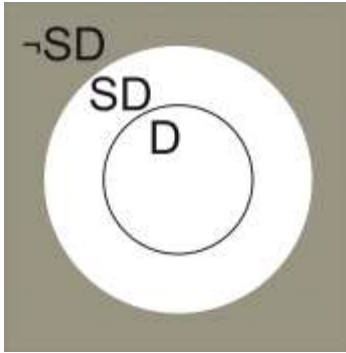


Figure 20.1 The relationships between D and SD

In this chapter we will look at properties of four classes of languages and see how they relate to each other. The classes we will consider are shown in Figure 20.1. They are:

- D , corresponding to the inner circle of the figure, the set of decidable languages.
- SD , corresponding to the outer circle of the figure, the set of semidecidable languages.
- SD/D , corresponding to the donut in the figure, the set of languages that are in SD but not D .
- $\neg SD$, corresponding to the hatched area in the figure, the set of languages that are not even semidecidable.

20.3 Subset Relationships between D and SD

The picture that we just considered implicitly makes three claims about the relationship between the classes D and SD . From the inside out they are:

1. D is a subset of SD . In other words, every decidable language is also semidecidable.
2. There exists at least one language that is in SD but not D and so the donut in the picture is not empty.
3. There exist languages that are not in SD . In other words, the gray area of the figure is not empty.

We have already proven the second of these claims: In Chapter 19 we described $H = \{ \langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w \}$ and showed that H is not in D but is in SD . We now consider each of the other two claims.

Theorem 20.2 D is a Subset of SD

Theorem: Every decidable language is also semidecidable.

Proof: The proof follows directly from the definitions of deciding and semideciding Turing machines. If L is in D , then it is decided by some Turing machine M . M therefore accepts all and only the strings in L . So M is also a semideciding machine for L . Since there is a Turing machine that semidecides L , it is in SD . ■

Next we consider whether the class SD includes all languages or whether there are languages that are not even semidecidable. As the picture we drew at the beginning of the chapter suggests (by the existence of the gray region), the answer is that there are languages that are not in SD .

Theorem 20.3 Not all Languages are in SD

Theorem: There exist languages that are not in SD .

Proof: We will use a counting argument. Assume any nonempty alphabet Σ . First we prove the following lemma:

Lemma: There is a countably infinite number of SD languages over Σ .

Proof of Lemma: Every semidecidable language is semidecided by some Turing machine. We can lexicographically enumerate all the syntactically legal Turing machines with input alphabet Σ . That enumeration is infinite, so, by Theorem 32.1, there is a countably infinite number of semideciding Turing machines. There cannot be more SD

languages than there are semideciding Turing machines, so there are at most a countably infinite number of SD languages. There is not a one-to-one correspondence between SD languages and semideciding Turing machines since there is an infinite number of machines that semidecide any given language. But the number of SD languages must be infinite because it includes (by Theorem 20.1 and Theorem 20.2) all the context-free languages and, by Theorem 13.2, there are an infinite number of them. So there is a countably infinite number of SD languages.

Proof of Theorem: There is an uncountably infinite number of languages over Σ (by Theorem 2.3). So there are more languages over Σ than there are in SD. Thus there must exist at least one language that is in \neg SD. ■

We will see our first example of a language that is in \neg SD in the next section.

20.4 The Classes D and SD Under Complement

The regular languages are closed under complement. The context free languages are not. What about the decidable (D) languages and the semidecidable (SD) languages?

Theorem 20.4 The Decidable Languages are Closed under Complement

Theorem: The class D is closed under complement.

Proof: The proof is by a construction that is analogous to the one we used to show that the regular languages are closed under complement. Let L be any decidable language. Since L is in D, there is some deterministic Turing machine M that decides it. Recall that a deterministic Turing machine must be completely specified (i.e., there must be a transition from every nonhalting state on every character in the tape alphabet), so there is no need to worry about a dead state. From M we construct M' to decide $\neg L$. Initially, let $M' = M$. Now swap the y and n states. M' halts and accepts whenever M would halt and reject; M' halts and rejects whenever M would halt and accept. Since M always halts, so does M' . And M' accepts exactly those strings that M would reject, i.e., $\neg L$. Since there is a deciding machine for $\neg L$, it is in D. ■

Theorem 20.5 The Semidecidable Languages are not Closed under Complement

Theorem: The class SD is not closed under complement.

Proof: The proof is by contradiction. Suppose the class SD were closed under complement. Then, given any language L in SD, $\neg L$ would also be in SD. So there would be a Turing machine M that semidecides L and another Turing machine M' that semidecides $\neg L$. From those two we could construct a new Turing machine $M\#$ that decides L . On input w , $M\#$ will simulate M and M' , in parallel, running on w . Since w must be an element of either L or $\neg L$, one of M or M' must eventually accept. If M accepts, then $M\#$ halts and accepts. If M' accepts, then $M\#$ halts and rejects. So, if the SD languages were closed under complement, then all SD languages would also be in D. But we know from Chapter 19 that $H = \{ \langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w \}$ is in SD but not D. ■

These last two theorems give us a new way to prove that a language L is in D (or, in fact, a way to prove that a language is not in SD):

Theorem 20.6 L and $\neg L$ Both in SD is Equivalent to L is in D

Theorem: A language L is in D iff both it and its complement $\neg L$ are in SD.

Proof: We prove each direction of the implication:

Proof that L in D implies L and $\neg L$ are in SD: Because L is in D, it must also be in SD by Theorem 20.2. But what about $\neg L$? By Theorem 20.4, the class D is closed under complement, so $\neg L$ is also in D. And so, using Theorem 20.2 again, it is also in SD.

Proof that L and $\neg L$ are in SD implies L is in D: The proof is by construction and uses the same construction that we used to prove Theorem 20.5: Since L and $\neg L$ are in SD, they each have a semideciding Turing machine. Suppose L is semidecided by M and $\neg L$ is semidecided by M' . From those two we construct a new Turing machine $M\#$ that decides L . On input w , $M\#$ will simulate M and M' in parallel, running on w . Since w must be an element of either L or $\neg L$, one of M_1 or M_2 must eventually accept. If M_1 accepts, then $M\#$ halts and accepts. If M_2 accepts, then $M\#$ halts and rejects. Since $M\#$ decides L , L is in D. ■

We can use Theorem 20.6 to prove our first example of a language that is not in SD:

Theorem 20.7 $\neg H$ is not in SD

Theorem: The language $\neg H$, (the complement of H) = $\{ \langle M, w \rangle : \text{Turing machine } M \text{ does not halt on input string } w \}$ is not in SD.

Proof: Recall that we are defining the complement of languages involving Turing machine descriptions with respect to the universe of syntactically well-formed strings. From Theorem 19.1, we know that H is in SD (since we showed a semideciding Turing machine for it). By Theorem 20.6 we know that if $\neg H$ were also in SD then H would be in D. But, by Theorem 19.2, we know that H is not in D. So $\neg H$ is not in SD. ■

20.5 Enumerating a Language

In most of our discussion so far, we have defined a language by specifying either a grammar that can generate it or a machine that can accept it. But it is also possible to specify a machine that is a generator. Its job is to enumerate (in some order) the strings of the language. We will now explore how to use a Turing machine to do that.

20.5.1 Enumerating in Some Undefined Order

To generate a language L , we need a Turing machine M whose job is to start with a blank tape, compute for a while, place some string in L on the tape, signal that we should snapshot the tape to record its contents, and then go back and do it all that again. If L is finite, we can construct M so that it will eventually halt. If L is infinite, M must continue generating forever. If a Turing machine M behaves in this way and outputs all and only the strings in L , then we say that M *enumerates* L . Any enumerating Turing machine M must have a special state that we will call p (for print). Whenever M enters p , the shortest string that contains all the nonblank characters on M 's tape will be considered to have been enumerated by M . Note that p is not a halting state. It merely signals that the current contents of the tape should be viewed as a member of L . M may also have a halting state if L is finite. Formally, we say that a Turing machine M enumerates L iff, for some fixed state p of M ,

$$L = \{ w : (s, \sqcup) \vdash_{-M}^* (p, w) \}$$

A language L is **Turing-enumerable** iff there is a Turing machine that enumerates it. Note that we are making no claim here about the order in which the strings in L are generated.

To make it easy to describe enumerating Turing machines in our macro language, we'll define the simple subroutine P , shown in Figure 20.2. It simply enters the state p and halts.



Figure 20.2 A subroutine that takes a snapshot of the tape

Example 20.1 Enumerating in Lexicographic and in Random Order

Consider the language a^* . Here are two different Turing machines that enumerate it:

M_1 :

\downarrow
 \overline{PaR}

M_2 :

\downarrow
 $\overline{PaPQRaRaRaPQP}$

M_1 enumerates a^* in lexicographic order. M_2 enumerates it in a less straightforward order. It will produce the sequence $\epsilon, a, aaa, aa, aaaaa, aaaa, \dots$

So now we have one mechanism for using a Turing machine to generate a language and a separate mechanism for using a Turing machine to accept one. Is there any relationship between the class of Turing-enumerable languages and either the class of decidable languages (D) or the class of semidecidable languages (SD)? The answer is yes. The class of languages that can be enumerated by a Turing machine is identical to SD.

Theorem 20.8 Turing Enumerable is Equivalent to Semidecidable

Theorem: A language is in SD (i.e., it can be semidecided by some Turing machine) iff it is Turing-enumerable.

Proof: We must do two proofs, one that shows that if a language is Turing enumerable then it is in SD and another that shows that if a language is in SD then it is Turing enumerable.

Proof that if a language is Turing enumerable then it is in SD: if a language L is Turing enumerable then there is some Turing machine M that enumerates it. We convert M to a machine M' that semidecides L :

$M'(w: \text{string}) =$

1. Save input w on a second tape.
2. Invoke M , which will enumerate L . Each time an element of L is enumerated, compare it to w . If they match, halt and accept. Otherwise, continue with the enumeration.

Because there is a Turing machine that semidecides L , it is in SD. Figure 20.3 illustrates how M' works.

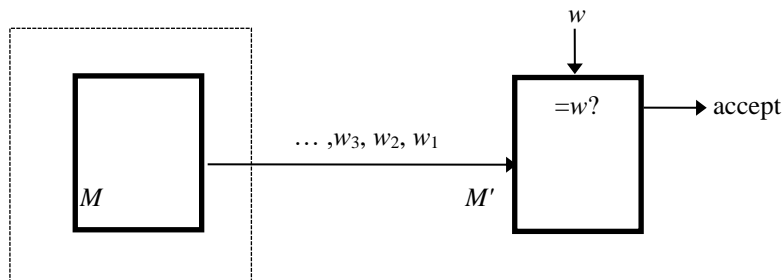


Figure 20.3 Using an enumerating machine in a semidecider

Proof that if a language is in SD then it is Turing enumerable: if $L \subseteq \Sigma^*$ (for some Σ) is in SD, then there is a Turing machine M that semidecides it. We will use M to construct a new machine M' that enumerates L . The idea behind M' is that it will lexicographically enumerate Σ^* . It will consider each of the strings it enumerates as a candidate for membership in L . So it will pass each such string to M . Whenever M accepts some string w , M' will output it. The problem is that M is not guaranteed to halt. So what happens if M' invokes M on a string that is not in L and M loops? If we are not careful, M' will wait forever and never give other strings a chance.

To solve this problem, M' will not just invoke M and sit back and wait to see what happens. It will carefully control the execution of M . In particular, it will invoke M on $string_1$ and let it compute one step. Then it will consider $string_2$. It will allow M to compute one step on $string_2$ and also one more step on $string_1$. Then it will consider $string_3$, this time trying the new $string_3$ for one step and applying one more step to the computations on $string_2$ and on $string_1$. Anytime M accepts some string in this sequence, M' will output that string. If there is some string s that is not in L , then the computation corresponding to s will either halt and reject or fail to halt. In either case, M' will never output s .

This pattern is shown in Figure 20.4. Each column corresponds to a candidate string and each row corresponds to one stage of the process. At each stage, a new string is added and one more step is executed for each string that is already being considered but on which M has not yet halted. The number of steps that have been executed on each string so far is shown in brackets. If M does halt on some string (as, for example, b , in the chart below), that column will simply be skipped at future stages.

ϵ [1]						
ϵ [2]	a [1]					
ϵ [3]	a [2]	b [1]				
ϵ [4]	a [3]	b [2]	aa [1]			
ϵ [5]	a [4]	<u>b</u> [3]	aa [2]	ab [1]		
ϵ [6]	a [5]		aa [3]	ab [2]	ba [1]	

Figure 20.4 Using dovetailing to control simulation

We will call the technique that we just described *dovetailing*. It will turn out to be useful for other similar kinds of proofs later.

So a description of M' is:

$M'() =$

1. Enumerate all $w \in \Sigma^*$ lexicographically. As each string w_j is enumerated:
 - 1.1. Start up a copy of M with w_j as its input.
 - 1.2. Execute one step of each M_i initiated so far, excluding only those that have previously halted.
2. Whenever an M_i accepts, output w_i .

■

20.5.2 Enumerating in Lexicographic Order

So far, we have said nothing about the order in which the strings in L are enumerated by M . But now suppose we do. We say that M *lexicographically enumerates* L iff M enumerates the elements of L in lexicographic order. A language L is *lexicographically Turing-enumerable* iff there is a Turing machine that lexicographically enumerates it.

Now we can ask whether there is any relationship between the class of lexicographically Turing-enumerable languages and any of the other classes we have already defined. Just as we found in the last section, in the case of unordered enumeration, we discover that the answer is yes. The class of languages that can be lexicographically enumerated by a Turing machine is identical to D .

Theorem 20.9 Lexicographically Turing Enumerable is Equivalent to Decidable

Theorem: A language is in D iff it is lexicographically Turing-enumerable.

Proof: Again we must do two proofs, one for each direction of the implication.

Proof that if a language is in D then it is lexicographically Turing enumerable: First consider what happens if L is finite. Then it is both in D and lexicographically TE. It is in D because every finite language is regular (Theorem 8.2), every regular language is CF (Theorem 13.1), and every CF language is in D (Theorem 20.1). And it is lexicographically TE by a hardwired TM that writes the elements of L one at a time, in lexicographic order, and then halts.

Now consider the case in which L is infinite: If a language $L \subseteq \Sigma^*$ (for some Σ) is in D, then there is some Turing machine M that decides it. Using M , we can build M' , which lexicographically generates the strings in Σ^* and tests them, one at a time by passing them to M . Since M is a deciding machine, it halts on all inputs, so dovetailing is not required here. If, on string w , M halts and accepts, then M' outputs w . If M halts and rejects, then M' just skips w and goes on to the next string in the lexicographic enumeration. Thus M' lexicographically enumerates L . The relationship between M and M' can be seen in Figure 20.5.

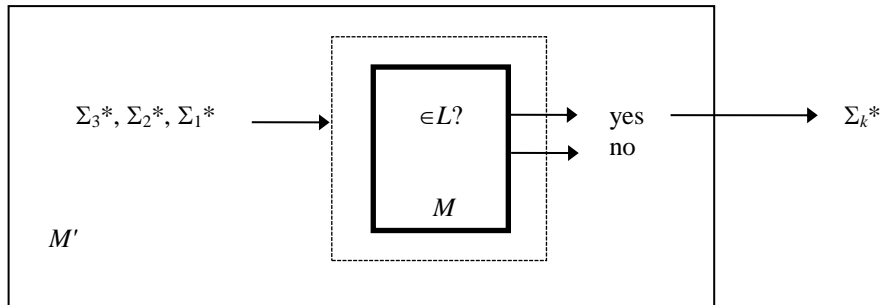


Figure 20.5 Using a decider in a lexicographic enumerator

Proof that if a language is lexicographically Turing enumerable then it is in D: If a language L is lexicographically Turing enumerable, then there is some Turing machine M that lexicographically enumerates it. Using M , we can build M' , which, on input w , starts up M and waits until either M generates w (in which case M' accepts w), M generates a string that comes after w in the enumeration (in which case M' rejects because it is clear that M will never go back and generate w), or M halts (in which case M' rejects because M failed to generate w). Thus M' decides L . The relationship between M and M' can be seen in Figure 20.6.

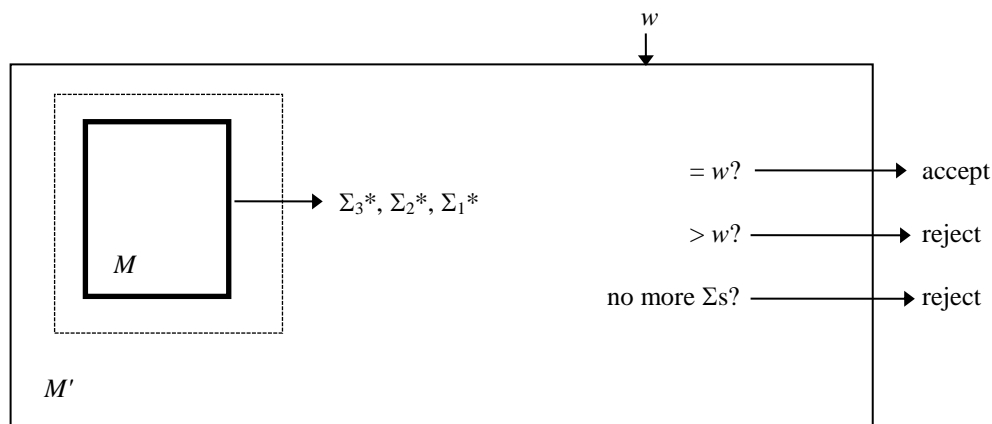


Figure 20.6 Using a lexicographic enumerator in a decider



20.6 Summary

In this chapter we have considered several ways in which the classes D and SD are related and we have developed theorems that give us ways to prove that a specific language L is in D and/or SD. Figure 20.7 attempts to summarize these results. The column labeled IN lists our techniques for proving that a language is in the corresponding language class. The column labeled OUT lists our techniques for proving that a language is not in the corresponding language class. We have listed reduction here for completeness. We will present reduction as a proof technique in Chapter 21. And we have mentioned unrestricted grammars, which we will discuss in Chapter 23. You'll also note, in the figure, one example language in each class.

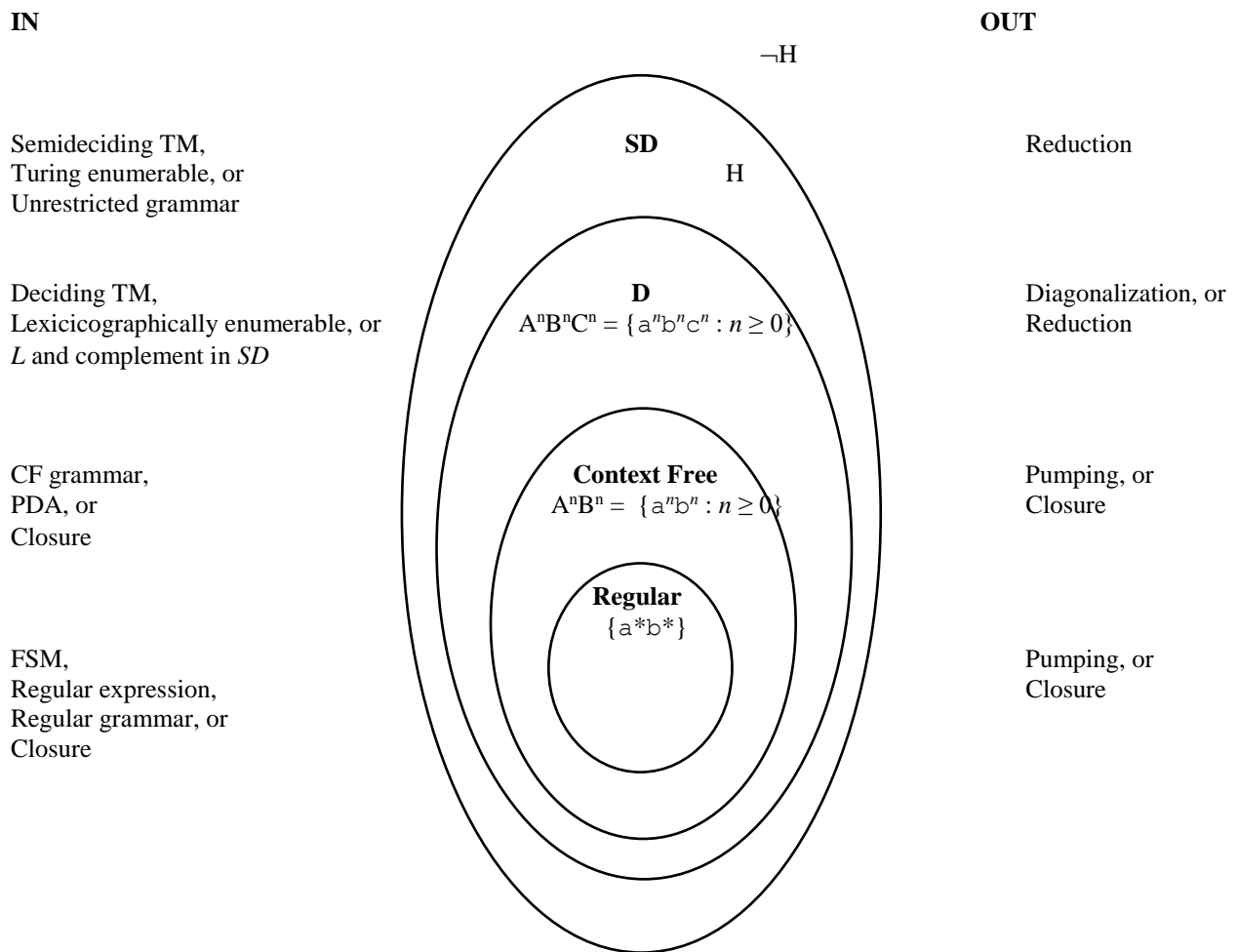


Figure 20.7 Relating four language classes

20.7 Exercises

- 1) Show that the set D (the decidable languages) is closed under:
 - a) Union
 - b) Concatenation
 - c) Kleene star

- d) Reverse
e) Intersection
- 2) Show that the set SD (the semidecidable languages) is closed under:
- Union
 - Concatenation
 - Kleene star
 - Reverse
 - Intersection
- 3) Let L_1, L_2, \dots, L_k be a collection of languages over some alphabet Σ such that:
- For all $i \neq j$, $L_i \cap L_j = \emptyset$.
 - $L_1 \cup L_2 \cup \dots \cup L_k = \Sigma^*$.
 - $\forall i$ (L_i is in SD).
- Prove that each of the languages L_1 through L_k is in D.
- 4) If L_1 and L_3 are in D and $L_1 \subseteq L_2 \subseteq L_3$, what can we say about whether L_2 is in D?
- 5) Let L_1 and L_2 be any two decidable languages. State and prove your answer to each of the following questions:
- Is it necessarily true that $L_1 - L_2$ is decidable?
 - Is it possible that $L_1 \cup L_2$ is regular?
- 6) Let L_1 and L_2 be any two undecidable languages. State and prove your answer to each of the following questions:
- Is it possible that $L_1 - L_2$ is regular?
 - Is it possible that $L_1 \cup L_2$ is in D?
- 7) Let M be a Turing machine that lexicographically enumerates the language L . Prove that there exists a Turing machine M' that decides L^R .
- 8) Construct a standard one-tape Turing machine M to enumerate the language:

$\{w : w \text{ is the binary encoding of a positive integer that is divisible by } 3\}$.

Assume that M starts with its tape equal to \square . Also assume the existence of the printing subroutine P , defined in Section 20.5.1. As an example of how to use P , consider the following machine, which enumerates L' , where $L' = \{w : w \text{ is the unary encoding of an even number}\}$:



You may find it useful to define other subroutines as well.

- 9) Construct a standard one-tape Turing machine M to enumerate the language $A^n B^n$. Assume that M starts with its tape equal to \square . Also assume the existence of the printing subroutine P , defined in Section 20.5.1.
- 10) If w is an element of $\{0, 1\}^*$, let $\neg w$ be the string that is derived from w by replacing every 0 by 1 and every 1 by 0. So, for example, $\neg 011 = 100$. Consider an infinite sequence S defined as follows:

$$\begin{aligned} S_0 &= 0. \\ S_{n+1} &= S_n \neg S_n. \end{aligned}$$

The first several elements of S are 0, 01, 0110, 01101001, 0110100110010110. Describe a Turing machine M to output S . Assume that M starts with its tape equal to \square . Also assume the existence of the printing subroutine P , defined in Section 20.5.1, but now with one small change: if M is a multitape machine, P will output the value of tape 1. (Hint: use two tapes.)

- 11) Recall the function *mix*, defined in Example 8.23. Neither the regular languages nor the context-free languages are closed under *mix*. Are the decidable languages closed under *mix*? Prove your answer.
- 12) Let $\Sigma = \{a, b\}$. Consider the set of all languages over Σ that contain only even length strings.
 - a) How many such languages are there?
 - b) How many of them are semidecidable?
- 13) Show that every infinite semidecidable language has a subset that is not decidable.

21 Decidability and Undecidability Proofs

We now know two languages that are not in D :

- $H = \{ \langle M, w \rangle : \text{Turing machine } M \text{ halts on input } w \}$
- $\neg H = \{ \langle M, w \rangle : \text{Turing machine } M \text{ does not halt on input } w \}$ (which also isn't in SD)

In this chapter we will see that they are not alone. Recall that we have two equivalent ways to describe a question: as a language (in which case we ask whether it is in D), and as a problem (in which case we ask whether it is decidable or whether it can be solved). Although all of our proofs will be based on the language formulation, it is sometimes easier, particularly for programmers, to imagine the question in its problem formulation. Table 21.1 presents a list, stated both ways, of some of the undecidable questions that we will consider in this and succeeding chapters.

The Problem View	The Language View
Given a Turing machine M and a string w , does M halt on w ?	$H = \{ \langle M, w \rangle : \text{TM } M \text{ halts on input } w \}$
Given a Turing machine M and a string w , does M not halt on w ?	$\neg H = \{ \langle M, w \rangle : \text{TM } M \text{ does not halt on input } w \}$
Given a Turing machine M , does M halt on the empty tape?	$H_\varepsilon = \{ \langle M \rangle : \text{TM } M \text{ halts on } \varepsilon \}$
Given a Turing machine M , is there any string on which M halts?	$H_{\text{ANY}} = \{ \langle M \rangle : \text{there exists at least one string on which TM } M \text{ halts} \}$
Given a Turing machine M , does M accept all strings?	$A_{\text{ALL}} = \{ \langle M \rangle : L(M) = \Sigma^* \}$
Given two Turing machines M_a and M_b , do they accept the same languages?	$\text{EqTMs} = \{ \langle M_a, M_b \rangle : L(M_a) = L(M_b) \}$
Given a Turing machine M , is the language that M accepts regular?	$\text{TM}_{\text{REG}} = \{ \langle M \rangle : L(M) \text{ is regular} \}$

Table 21.1 The problem and the language view

Some of these languages are also not in SD . We will return to them in Section 21.6, where we will see how to prove that languages are not in SD .

The primary technique that we will use here to show that a language L is not in D is reduction. We will show that if L were in D , we could use its deciding machine to decide some other language that we already know is not decidable. Thus we can conclude that L is not decidable either.

21.1 Reduction

We *reduce* a problem to one or more other problems when we describe a solution to the first problem in terms of solutions to the others. We generally choose to reduce to simpler problems, although sometimes it makes sense to pick problems just because we already have solutions for them. Reduction is ubiquitous in everyday life, puzzle solving, mathematics, and computing.

21.1.1 Everyday Examples of Reduction

Example 21.1 Calling Jen

We want to call our friend Jen but don't have her number. But we know that Jim has it. So we reduce the problem of finding Jen's number to the problem of getting hold of Jim.

The most important property of a reduction is clear even in the very simple example of finding Jen's number:

The reduction exists \wedge There is a procedure that works for getting hold of Jim \rightarrow We will have Jen's number.

But what happens if there is no way to get hold of Jim? Does that mean that we cannot find Jen's number? No. There may be some other way to get it.

If, on the other hand, we knew (via some sort of oracle) that there is no way we could ever end up with Jen's number, and if we still believed in the reduction (i.e., we believed that Jim knows Jen's number and would be willing to give it to us), we would be forced to conclude that there exists no effective procedure for getting hold of Jim.

Example 21.2 Crisis Detection

Suppose that we want to know whether there is some sort of crisis brewing in (pick one): the world, our city, the company we work for. We'd like to ask: the Pentagon, the city council, or top management, but they probably won't tell us. But perhaps we can reduce this question to one we can answer: Has there been a spike this week in orders for middle-of-the-night pizza delivery to: the Pentagon, the town hall, corporate headquarters? This reduction will work provided all of the following are true:

- There will be all-nighters at the specified locations if and only if there is a crisis.
- There will be a spike in middle-of-the-night pizza orders if and only if there are all-nighters there.
- It is possible to find out about pizza orders.

The crisis-detection example illustrates a common use of reduction: we wish to solve a problem but have no direct way of doing so. So we look for a way to transform the problem we care about into some other problem that we can solve. The transformation must have the property that the answer to this new problem provides the answer to the original one.

Example 21.3 Fixing Dinner

We can reduce the problem of fixing dinner to a set of simpler problems: fix the entrée, fix the salad, and fix the dessert.

Example 21.4 Theorem Proving

Suppose that we want to establish $Q(A)$ and that we have, as a theorem:

$$\forall x (R(x) \wedge S(x) \wedge T(x) \rightarrow Q(x)).$$

Then we can reduce the problem of proving $Q(A)$ to three new ones: proving $R(A)$, $S(A)$, and $T(A)$.

Backward chaining solves problems by reducing complex goals to simpler ones until direct solutions can be found. It is used in theorem provers and in a variety of kinds of automatic reasoning and intelligent systems. © 762.

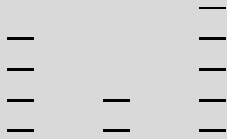
These last two examples illustrate an important kind of reduction, often called *divide-and-conquer*. One problem is reduced to two or more problems, all of which must be solved in order to produce a solution to the original problem. But each of the new problems is assumed to be easier to solve than the original one was.

Example 21.5 Nim

Nim¹¹ starts with one or more piles of sticks. Two players take turns removing sticks from the piles. At each turn, a player chooses a pile and may remove some or all of the sticks from that pile. The player who is left with no sticks

¹¹ This description is taken from [Misra 2004].

to remove loses. For example, an initial configuration of a Nim game could be the following, in which the sticks are arranged in three piles:



Consider the problem of determining whether there is any move that we can make that will guarantee that we can win. The obvious way to solve this problem is to search the space of legal moves until we find a move that makes it impossible for the other player to win. If we find such a move, we know that we can force a win. If we don't, then we know that we cannot. But the search tree can be very large and keeping track of it is nearly impossible for people. So how can we answer the question?

We can reduce the problem of searching a Nim game tree to a simple problem in Boolean arithmetic. We represent the number of sticks in each pile as a binary number, arrange the numbers in a column, lining up their low-order digits, and then apply the exclusive-or (XOR) operator to each column. So, in the example above, we'd have:

```

100 (4)
010 (2)
101 (5)
011

```

If the resulting string is in 0^+ , then the current board position is a guaranteed loss for the current player. If the resulting string is not in 0^+ , then there is a move by which the current player can assure that the next position will be a guaranteed loss for the opponent. So, given a Nim configuration, we can decide whether we can guarantee a win by transforming it into the XOR problem we just described and then checking to see that the result of the XOR is not in 0^+ .

In addition, we can easily extend this approach so that it tells us what move we should make. All that is required is to choose one number (i.e., one pile of sticks) and subtract from it some number such that the result of XORing together the new counts will yield some string in 0^+ . There may be more than one such move, but it suffices just to find the first one. So we try the rows one at a time. In our example, we quickly discover that if we remove one stick from the second pile (the one currently containing two sticks), then we get:

```

100 (4)
001 (1)
101 (5)
000

```

So we remove one stick from the second pile. No search of follow-on moves is required.

Some combinatorial problems can be solved easily by reducing them to graph problems. \mathfrak{B} 648.

Example 21.6 Computing a Function

Suppose that we have access only to a very simple calculator that can perform integer addition but not multiplication. We can reduce the problem of computing $x \cdot y$ to the problem of computing $a + b$ as follows:

```

multiply(x: integer, y: integer) =
  answer = 0.
  For i = 1 to |y| do:
    answer = answer + x.
  Return answer.

```

21.2 Using Reduction to Show that a Language is Not Decidable

So far, we have used reduction to show that problem₁ is solvable if problem₂ is. Now we will turn the idea around and use it to show that problem₂ is not solvable given that we already know that problem₁ isn't. Reduction, as we are about to use it, is a *proof by contradiction* technique. We will say, "Suppose that problem₂ were decidable. Then we could use its decider as a subroutine that would enable us to solve problem₁. But we already know that there is no way to solve problem₁. So there isn't any way to solve problem₂ either."

Example 21.7 Dividing an Angle

Given an arbitrary angle, divide it into sixths, using only a straightedge and a compass. We show that there exists no general procedure to solve this problem. Suppose that there were such a procedure, which we'll call *sixth*. Then we could define the following procedure to trisect an arbitrary angle:

trisect(*a*: angle) =

1. Divide *a* into six equal parts by invoking *sixth*(*a*).
2. Ignore every other line, thus dividing *a* into thirds.

So we have reduced the problem of trisecting an angle to the problem of dividing it into sixths. But we know that there exists no procedure for trisecting an arbitrary angle using only a straightedge and compass. The proof of that claim relies on a branch of mathematics known as Galois theory \square , after the French mathematician Evariste Galois, who was working on the problem of discovering solutions for polynomials of arbitrary degree. An interesting tidbit from the history of mathematics: Galois's work in this area was done while he was still a teenager, but was not published during his lifetime, which ended when he was killed in a duel in 1832 at age 20.

If *sixth* existed, then *trisect* would exist. But we know that *trisect* cannot exist. So neither can *sixth*.

In the rest of this chapter, we are going to construct arguments of exactly this form to show that various languages are not in D because $H = \{ \langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w \}$ isn't. We'll then extend the technique to show that some languages are not in SD either (because $\neg H$ isn't). But, before we do that, we should note one very important thing about arguments of this sort: solvability (and decidability) results can hinge on the details of the specifications of the problems involved. For example, let's reconsider the angle trisection problem. This time, instead of the requirement, "using only a straightedge and a compass", we'll change the rules to "in origami". Now, it turns out that it is possible to trisect an angle using the paper folding and marking operations that origami provides \square . We will have to be very careful to state exactly what we mean in specifying the languages that we are about to consider.

In the rest of this chapter, we are going to use reduction in a very specific way. The goal of a reduction is to enable us to describe a decision procedure for a language L_1 by using a decision procedure (which we will call *Oracle*) that we hypothesize exists for some other language L_2 . Furthermore, since our goal is to develop a decision procedure (i.e., design a Turing machine), we are interested only in reductions that are themselves computable (i.e., can be implemented as a Turing machine that is guaranteed to halt). So the precise meaning of reduction that we will use in the rest of this book is the following:

A *reduction* R from L_1 to L_2 consists of one or more Turing machines with the following property: if there exists a Turing machine *Oracle* that decides (or semidecides) L_2 , then the Turing machines in R can be composed with *Oracle* to build a deciding (or a semideciding) Turing machine for L_1 . The idea is that the machines in R perform the straightforward parts of the task, while we assume that *Oracle* can do a good deal of the work.¹²

We will focus on the existence of deciding Turing machines now. Then, in Section 21.6, we will use this same idea when we explore the existence of semideciding machines. We will use the notation $P \leq P'$ to mean that P is reducible to P' . While we require that a reduction be one or more Turing machines, we will allow the use of clear pseudocode

¹² It is common to define a reduction as a function, rather than as a Turing machine. But, when that is done, we require that the function be computable. Since the computable functions are exactly the functions that can be computed by some Turing machine, these two definitions are equivalent.

as a way to specify the machines. Because the key property of a reduction, as we have just defined it, is that it be computable by Turing machines, reducibility in this sense is sometimes called **Turing reducibility**.

Since our focus in the rest of Part IV is on answering the question, “Does there exist a Turing machine to decide (or semidecide) some language L ?” we will accept as a reduction any collection of Turing machines that meets the definition that we just gave. If, in addition, we cared about the efficiency of our (semi)deciding procedure, we would also have to care about the efficiency of the reduction. We will discuss that issue in Part V.

Having defined reduction precisely in terms of Turing machines, we can now return to the main topic of the rest of this chapter: How can we use reduction to show that some language L_2 is not decidable? When we reduce L_1 to L_2 via a reduction R , we show that if L_2 is in D then so is L_1 (because we can decide it with a composition of the machines in R with the *Oracle* that decides L_2). So what if we already know that L_1 is not in D ? Then we have just shown that L_2 isn’t either.

To see why this is so, recall that the definition of reduction tells us that:

$$(R \text{ is a reduction from } L_1 \text{ to } L_2) \wedge (L_2 \text{ is in } D) \rightarrow (L_1 \text{ is in } D).$$

If $(L_1 \text{ is in } D)$ is false, then at least one of the two antecedents of that implication must be false. So if we know that $(R \text{ is a reduction from } L_1 \text{ to } L_2)$ is true, then $(L_2 \text{ is in } D)$ must be false.

We now have a way to show that some new language L_2 is not in D : we find a language that is reducible to L_2 and that is known not to be in D . We already have one language, H , that is not in D . So we can use it to prove that other languages aren’t either.

Figure 21.1 shows the form of this argument graphically. The solid arrow indicates the reduction R from L_1 to L_2 . The other two arrows correspond to implication. So the diagram says that if L_1 is reducible to L_2 then we know (as shown in the upward implication) that if L_2 is in D , so is L_1 . But L_1 is known not to be in D . So (as shown in the downward implication) we know that L_2 is not in D either.

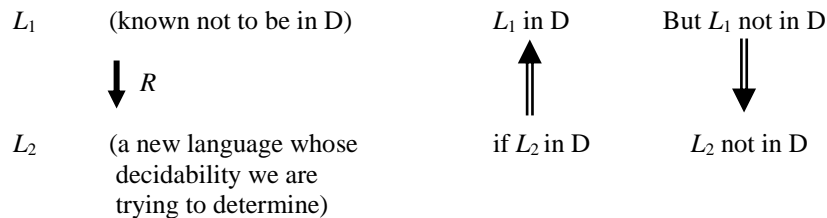


Figure 21.1 Using reduction for undecidability

The important thing about this diagram is the direction of the arrows. We reduce L_1 to L_2 to show that the undecidability of L_1 guarantees that L_2 is also undecidable. As we do our reduction proofs, we must be careful always to reduce a known undecidable language to the unknown one. The most common mistake in doing reduction proofs is to do them backwards.

Summarizing what we have said: To use reduction to show that a language L_2 is not in D , we need to do three things:

1. Choose a language L_1 to reduce from. We must choose an L_1 :
 - that is already known not to be in D , and
 - that can be reduced to L_2 (i.e., there would be a deciding machine for it if there existed a deciding machine for L_2).

2. Define the reduction R and describe the composition C of R with *Oracle*, the machine that we hypothesize decides L_2 .
3. Show that C does correctly decide L_1 if *Oracle* exists. We do this by showing:
 - that R can be implemented as one or more Turing machines.
 - that C is correct, meaning that it correctly decides whether its input x is an element of L_1 . To do this, we must show that:
 - If $x \in L_1$, then $C(x)$ accepts, and
 - If $x \notin L_1$, then $C(x)$ rejects.

21.2.2 Mapping Reducibility

The most straightforward way to reduce one problem, which we'll call A , to another, which we'll call B , is to find a way to transform instances of A into instances of B . Then we simply hand the transformed input to the program that solves B and return the result. In Example 21.5, we illustrated this idea in our solution to the problem of determining whether or not we could force a win in the game of Nim. We transformed a problem involving a pile of sticks into a Boolean XOR problem. And we did it in such a way that a procedure that determined whether the result of the XOR was nonzero would also tell us whether we could force a win. So our reduction consisted of a single procedure *transform*. Then we argued that, if *XORsolve* solved the Boolean XOR problem, then *XORsolve(transform(x))* correctly decided whether x was a position from which we could guarantee a win.

In the specific context of attempting to solve decision procedures, we can formalize this idea as follows: given an alphabet Σ , we will say that L_1 is **mapping reducible** to L_2 , which we will write as $L_1 \leq_M L_2$, iff there exists some computable function f such that:

$$\forall x \in \Sigma^* (x \in L_1 \text{ iff } f(x) \in L_2).$$

In general, the function f gives us a way to transform any value x into a new value x' so that we can answer the question, "Is x in L_1 ?" by asking instead the question, "Is x' in L_2 ?" If f can be computed by some Turing machine R , then R is a **mapping reduction** from L_1 to L_2 . So, if $L_1 \leq_M L_2$ and there exists a Turing machine *Oracle* that decides L_2 , then the following Turing machine C , which is simply the composition of *Oracle* with R , will decide L_1 :

$$C(x) = \text{Oracle}(R(x)).$$

The first several reduction proofs that we will do use mapping reducibility. In the first few, we show that a new language L_2 is not in D because H can be reduced to it. Once we have done several of those proofs, we'll have a collection of languages, all of which have been shown not to be in D . Then, for a new proof that some language L_2 is not in D , it will suffice to show that any one of the others can be reduced to it.

Theorem 21.1 "Does M Halt on ε ?" is Undecidable

Theorem: The language $H_\varepsilon = \{ \langle M \rangle : \text{Turing machine } M \text{ halts on } \varepsilon \}$ is in SD/D .

Proof: We will first show that H_ε is in SD . Then we will show that it is not in D .

We show that H_ε is in SD by exhibiting a Turing machine T that semidecides it. T operates as follows:

- $T(\langle M \rangle) =$
1. Run M on ε .
 2. Accept.

T accepts $\langle M \rangle$ iff M halts on ε , so T semidecides H_ε .

Next we show that $H \leq_M H_\varepsilon$ and so H_ε is not in D . We will define a mapping reduction R whose job will be to map instances of H to instances of H_ε in such a way that, if there exists a Turing machine (which we will call *Oracle*) that decides H_ε , then *Oracle*($R(\langle M, w \rangle)$) will decide H .

R will transform any input of the form $\langle M, w \rangle$ into a new string, of the form $\langle M \# \rangle$, suitable as input to *Oracle*. Specifically, what R does is to build a new Turing machine, which we will call $M \#$, that halts on ϵ iff M halts on w . One way to do that is to build $M \#$ so that it completely ignores its own input. That means that it will halt on everything (including ϵ) or nothing. And we need for it to halt on everything precisely in case M would halt on w . That's easy. Let $M \#$ simply run M on w . It will halt on everything iff M halts on w . Note that $M \#$, like every Turing machine has an input, namely whatever is on its tape when it begins to execute. So we'll define a machine $M \#(x)$, where x is the name we'll give to $M \#'s$ input tape. We must do that even though, in this and some other cases we'll consider, it happens that the behavior of $M \#$ doesn't depend on what its input tape contains.

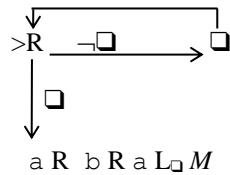
So let R be a mapping reduction from H to H_ϵ defined as follows:

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M \# \rangle$ of a new Turing machine $M \#(x)$ that, on input x , operates as follows:
 - 1.1. Erase the tape.
 - 1.2. Write w on the tape.
 - 1.3. Run M on w .
2. Return $\langle M \# \rangle$.

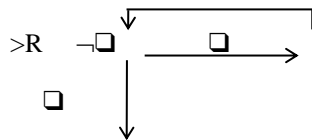
We claim that if *Oracle* exists and decides H_ϵ , then $C = Oracle(R(\langle M, w \rangle))$ decides H . To complete the proof, we need to show that R corresponds to a computable function (i.e., that it can be implemented by a Turing machine) and that C does in fact decide H :

- R can be implemented as a Turing machine: R must construct $\langle M \# \rangle$ from $\langle M, w \rangle$. To see what $M \#$ looks like, suppose that $w = aba$. Then $M \#$ will sweep along its input tape, blanking it out. Then it will write the string aba , move its read/write head back to the left, and, finally, pass control to M . So, in our macro language, $M \#$ will be:



The procedure for constructing $M \#$, given an arbitrary M and w , is:

1. Write the following code, which erases the tape:



2. For each character c in w do:
 - 2.1. Write c .
 - 2.2. If c is not the last character in w , write R .
3. Write $L \sqcup M$.

- C is correct: $M \#$ ignores its own input. It halts on everything or nothing. Think of its step 1.3 as a gate. The computation only makes it through the gate if M halts on w . If that happens then $M \#$ halts, no matter what its own input was. Otherwise, it loops in step 1.3. So:
 - If $\langle M, w \rangle \in H$: M halts on w , so $M \#$ halts on everything. In particular, it halts on ϵ . $Oracle(\langle M \# \rangle)$ accepts.
 - If $\langle M, w \rangle \notin H$: M does not halt on w , so $M \#$ halts on nothing and thus not on ϵ . $Oracle(\langle M \# \rangle)$ rejects.

But no machine to decide H can exist, so neither does *Oracle*. ■

This result may seem surprising. It says that if we could decide whether some Turing machine M halts on the specific string ϵ , then we could solve the more general problem of deciding whether a machine M halts on an arbitrary input. Clearly, the other way around is true: if we could decide H (which we cannot), then we could decide whether M halts on any one particular string. But doing a reduction in that direction would tell us nothing about whether H_ϵ is decidable. The significant thing that we just saw in this proof is that there also exists a reduction in the direction that does tell us that H_ϵ is not decidable.

To understand the reduction proof that we just did (and all the others that we are about to do), keep in mind that it involves two different kinds of languages:

- H and H_ϵ : The strings in H_ϵ are encodings of Turing machines, so they look like

$(q000,a000,q001,a010,\leftarrow), (q000,a000,q001,a010,\rightarrow), \dots$

The strings in H are similar, except that they also include a particular w , so they look like

$(q000,a000,q001,a010,\leftarrow), (q000,a000,q001,a010,\rightarrow), \dots; aabb$

- The language on which some particular Turing machine M , whose membership in either H or H_ϵ we are trying to determine, halts: since M can be any Turing machine, the set of strings on which M halts can be anything. It might, for example, be A^nB^n , in which case it would contain strings like $aaabbbb$. It could also, of course, be a language of Turing machine descriptions, but it will help to keep from getting confused if you think of M 's whose job is to recognize languages like A^nB^n that are very different from H .

The proof also referred to five different Turing machines:

- *Oracle* (the hypothesized, but provably nonexistent, machine to decide H_ϵ).
- R (the machine that builds $M\#$). This one actually exists.
- C (the composition of R with *Oracle*).
- $M\#$ (the machine whose description we will pass as input to *Oracle*). Note that $M\#$ will never actually be executed.
- M (the machine whose behavior on the input string w we are interested in determining). Its description is input to R .

Figure 21.2 shows a block diagram of C . It illustrates the relationship among the five machines.

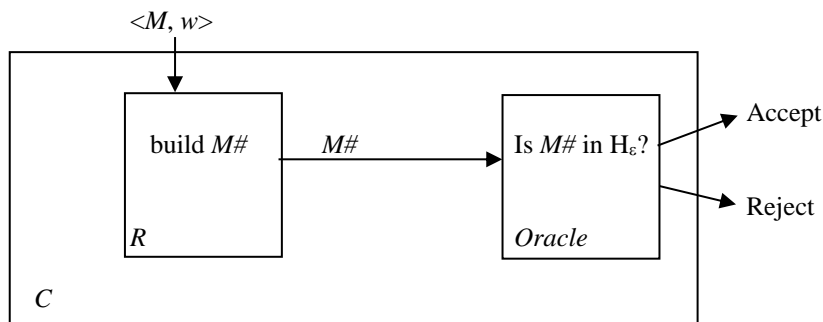


Figure 21.2 The relationships among C , R , and *Oracle*

Theorem 21.2 “Does M Halt on Anything?” is Undecidable

Theorem: The language $H_{ANY} = \{\langle M \rangle : \text{there exists at least one string on which Turing machine } M \text{ halts}\}$ is in SD/D.

Proof: Again, we will first show that H_{ANY} is in SD. Then we will show that it is not in D.

We show that H_{ANY} is in SD by exhibiting a Turing machine T that semidecides it. We could try building T so that it simply runs M on all strings in Σ^* in lexicographic order. If it finds one that halts, it halts. But, of course, the problem with this approach is that if M fails to halt on the first string T tries, T will get stuck and never try any others. So we need to try the strings in Σ^* in a way that prevents T from getting stuck. We build T so that it operates as follows:

$T(\langle M \rangle) =$

1. Use the dovetailing technique described in the proof of Theorem 20.8 to try M on all of the elements of Σ^* until there is one string on which M halts. Recall that, in dovetailing, we run M on one step of the first string, then another step on that string plus one step on the next, and so forth, as shown here (assuming $\Sigma = \{a, b\}$):

ε	[1]										
ε	[2]	a	[1]								
ε	[3]	a	[2]	b	[1]						
ε	[4]	a	[3]	b	[2]	aa	[1]				
ε	[5]	a	[4]	b	[3]	aa	[2]	ab	[1]		
ε	[6]	a	[5]	b	[4]	aa	[3]	ab	[2]	ba	[1]

2. If any instance of M halts, halt.

T will halt iff M halts on at least one string. So T semidecides H_{ANY} .

Next we show that $H \leq_M H_{ANY}$ and so H_{ANY} is not in D. Let R be a mapping reduction from H to H_{ANY} defined as follows:

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. Examine x .
 - 1.2. If $x = w$, run M on x , else loop.
2. Return $\langle M\# \rangle$.

If $Oracle$ exists and decides H_{ANY} , then $C = Oracle(R(\langle M, w \rangle))$ decides H :

- R can be implemented as a Turing machine: The proof is similar to that for Theorem 21.1. We will omit it in this and future proofs unless it is substantially different from the one we have already done.
- C is correct: $M\#$'s behavior depends on its input. The only string on which $M\#$ has a chance of halting is w . So:
 - If $\langle M, w \rangle \in H$: M halts on w , so $M\#$ halts on w . So there exists at least one string on which $M\#$ halts. $Oracle(\langle M\# \rangle)$ accepts.
 - If $\langle M, w \rangle \notin H$: M does not halt on w , so neither does $M\#$. So there exists no string on which $M\#$ halts. $Oracle(\langle M\# \rangle)$ rejects.

But no machine to decide H can exist, so neither does $Oracle$. ■

Sometimes there is more than one straightforward reduction that works. For example, here is an alternative proof that H_{ANY} is not in D:

Proof: We show that H_{ANY} is not in D by reduction from H . Let R be a mapping reduction from H to H_{ANY} defined as follows:

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. Erase the tape.
 - 1.2. Write w on the tape.
 - 1.3. Run M on w .
2. Return $\langle M\# \rangle$.

If *Oracle* exists and decides H_{ANY} , then $C = Oracle(R(\langle M, w \rangle))$ decides H . R can be implemented as a Turing machine. And C is correct. $M\#$ ignores its own input. It halts on everything or nothing. So:

- If $\langle M, w \rangle \in H$: M halts on w , so $M\#$ halts on everything. So it halts on at least one string. $Oracle(\langle M\# \rangle)$ accepts.
- If $\langle M, w \rangle \notin H$: M does not halt on w , so $M\#$ halts on nothing. So it does not halt on at least one string. $Oracle(\langle M\# \rangle)$ rejects.

But no machine to decide H can exist, so neither does *Oracle*. ■

Notice that we used the same reduction in this last proof that we used for Theorem 21.1. This is not uncommon. The fact that a single construction may be the basis for several reduction proofs is important. It derives from the fact that several quite different looking problems may in fact be distinguishing between the same two cases.

Recall the steps in doing a reduction proof of undecidability:

1. Choose an undecidable language L_1 to reduce from.
2. Define the reduction R .
3. Show that the composition of R with *Oracle* correctly decides L_1 .

We make choices at steps 1 and 2. Our last example showed that there may be more than one reasonable choice for step 2. There may also be more than one reasonable choice for step 1. So far, we have chosen to reduce from H . But now that we know other languages that are not in D , we could choose to use one of them. We want to pick one that makes step 2, constructing R , as straightforward as possible.

Theorem 21.3 “Does M Halt on Everything?” is Undecidable

Theorem: The language $H_{ALL} = \{\langle M \rangle : \text{Turing machine } M \text{ halts on } \Sigma^*\}$ is not in D . (Note: H_{ALL} is also not in SD , which we will show in Section 21.6.)

Proof: We show that $H_\epsilon \leq_M H_{ALL}$ and so H_{ALL} is not in D . We have chosen to use H_ϵ rather than H because H_ϵ looks more like H_{ALL} than H does. Both of them contain strings composed of a single Turing machine description, without reference to a particular string w . It is possible to do this proof by reduction from H instead. We leave that as an exercise.

Let R be a mapping reduction from H_ϵ to H_{ALL} defined as follows:

$R(\langle M \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. Erase the tape.
 - 1.2. Run M .
2. Return $\langle M\# \rangle$.

If *Oracle* exists and decides H_{ALL} , then $C = Oracle(R(\langle M \rangle))$ decides H_ϵ . R can be implemented as a Turing machine. And C is correct. $M\#$ runs M on ϵ . It halts on everything or nothing, depending on whether M halts on ϵ . So:

- If $\langle M \rangle \in H_\varepsilon$: M halts on ε , so $M\#$ halts on all inputs. $Oracle(\langle M\# \rangle)$ accepts.
- If $\langle M \rangle \notin H_\varepsilon$: M does not halt on ε , so $M\#$ halts on nothing. $Oracle(\langle M\# \rangle)$ rejects.

But no machine to decide H_ε can exist, so neither does $Oracle$. ■

Are safety and security properties of complex systems decidable? $\text{C } 718$.

We next define a new language that corresponds to the membership question for Turing machines:

- $A = \{ \langle M, w \rangle : \text{Turing machine } M \text{ accepts } w \}$.

Note that A is different from H , since it is possible that M halts but does not accept. Accepting is a stronger condition than halting. An alternative definition of A is then:

- $A = \{ \langle M, w \rangle : M \text{ is a Turing machine and } w \in L(M) \}$.

Recall that, for finite state machines and pushdown automata, the membership question was decidable. In other words, there exists an algorithm that, given M (an FSM or a PDA) and a string w , answers the question, “Does M accept w ?” We’re about to show that the membership question for Turing machines is undecidable.

Theorem 21.4 “Does M Accept w ?” is Undecidable

Theorem: The language $A = \{ \langle M, w \rangle : M \text{ is a Turing machine and } w \in L(M) \}$ is not in D .

Proof: We show that $H \leq_M A$ and so A is not in D . Since H and A are so similar, it may be tempting to define a mapping reduction R simply as the identity function:

$R(\langle M, w \rangle) =$
 1. Return $\langle M, w \rangle$.

But this won’t work, as we see immediately when we try to prove that $C = Oracle(R(\langle M, w \rangle))$ decides A :

- If $\langle M, w \rangle \in H$: M halts on w . It may either accept or reject. If M accepts w , then $Oracle(\langle M, w \rangle)$ accepts. But if M rejects w , $Oracle(\langle M, w \rangle)$ will reject.

So we cannot guarantee that $Oracle$ will accept whenever M halts on w . We need to construct R so that it passes to $Oracle$ a machine $M\#$ that is guaranteed both to halt and to accept whenever M would halt on w .

We can make that happen by defining R , a mapping reduction from H to A , as follows:

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. Erase the tape.
 - 1.2. Write w on the tape.
 - 1.3. Run M on w .
 - 1.4. Accept. /* This step is new. It is important since the hypothesized $Oracle$ will decide whether $M\#$ accepts w , not just whether it halts on w .
2. Return $\langle M\#, w \rangle$. /* Note that R returns not just a description of $M\#$. It returns a string that encodes both $M\#$ and an input string. This is important since any decider for A will accept only strings of that form. We chose w somewhat arbitrarily. We could have chosen any other string, for example ε .

If *Oracle* exists and decides A , then $C = Oracle(R(\langle M, w \rangle))$ decides H . R can be implemented as a Turing machine. And C is correct. $M\#$ ignores its own input. It accepts everything or nothing, depending on whether it makes it through the gate at step 1.3, and thus to step 1.4. So:

- If $\langle M, w \rangle \in H$: M halts on w , so $M\#$ accepts everything. In particular, it accepts w . $Oracle(\langle M\#, w \rangle)$ accepts.
- If $\langle M, w \rangle \notin H$: M does not halt on w . $M\#$ gets stuck in step 1.3 and so accepts nothing. In particular, it does not accept w . $Oracle(\langle M\#, w \rangle)$ rejects.

But no machine to decide H can exist, so neither does *Oracle*. ■

We can also define A_ϵ , A_{ANY} , and A_{ALL} , in a similar fashion and show that they too are not in D :

Theorem 21.5 “Does M Accept ϵ ?” is Undecidable

Theorem: The language $A_\epsilon = \{\langle M \rangle : \text{Turing machine } M \text{ accepts } \epsilon\}$ is not in D .

Proof: Analogous to that for H_ϵ . It is left as an exercise. ■

Theorem 21.6 “Does M Accept Anything?” is Undecidable

Theorem: The language $A_{ANY} = \{\langle M \rangle : \text{there exists at least one string that Turing machine } M \text{ accepts}\}$ is not in D .

Proof: Analogous to that for H_{ANY} . It is left as an exercise. ■

The fact that A_{ANY} is not in D means that there exists no decision procedure for the emptiness question for the SD languages. Note that, in this respect, they are different from both the regular and the context-free languages, for which such a procedure does exist.

Theorem 21.7 “Does M Accept Everything?” is Undecidable

Theorem: The language $A_{ALL} = \{\langle M \rangle : M \text{ is a Turing machine and } L(M) = \Sigma^*\}$ is not in D .

Proof: Analogous to that for H_{ALL} . It is left as an exercise. ■

So far, we have discovered that many straightforward questions that we might like to ask about the behavior of Turing machines are undecidable. It should come as no surprise then to discover that the equivalence question for Turing machines is also undecidable. Consider the language:

- $\text{EqTMs} = \{\langle M_a, M_b \rangle : M_a \text{ and } M_b \text{ are Turing machines and } L(M_a) = L(M_b)\}$.

We will show that EqTMs is not in D . How can we use reduction to do that? So far, all the languages that we know are not in D involve the description of a single Turing machine. So suppose that EqTMs were decidable by some Turing machine *Oracle*. How could we use *Oracle*, which compares two Turing machines, to answer questions about a single machine, as we must do to solve H or A or any of the other languages we have been considering? The answer is that our reduction R must create a second machine $M\#$ whose behavior it knows completely. Then it can answer questions about some other machine by comparing it to $M\#$. We illustrate this idea in our proof of the next theorem.

Consider the problem of virus detection. Suppose that a new virus V is discovered and its code is determined to be $\langle V \rangle$. Is it sufficient for antivirus software to check solely for occurrences of $\langle V \rangle$? © 725.

Theorem 21.8 “Are Two Turing Machines Equivalent?” is Undecidable

Theorem: The language $\text{EqTMs} = \{\langle M_a, M_b \rangle : M_a \text{ and } M_b \text{ are Turing machines and } L(M_a) = L(M_b)\}$ is not in D.

Proof: We show that $A_{\text{ALL}} \leq_M \text{EqTMs}$ and so EqTMs is not in D. Let R be a mapping reduction from A_{ALL} to EqTMs defined as shown below. Since R must invoke *Oracle* on a pair of Turing machines, it will create one new one, $M\#$, which can be compared to M . The idea is that $M\#$ will be designed so that it simply halts and accepts, whatever its input is. By comparing M to $M\#$, we can determine M 's behavior:

$R(\langle M \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. Accept.
2. Return $\langle M, M\# \rangle$.

If *Oracle* exists and decides EqTMs , then $C = \text{Oracle}(R(\langle M \rangle))$ decides A_{ALL} . R can be implemented as a Turing machine. And C is correct. $M\#$ accepts everything. So if $L(M) = L(M\#)$, M must also accept everything. So:

- If $\langle M \rangle \in A_{\text{ALL}}$: $L(M) = L(M\#)$. $\text{Oracle}(\langle M, M\# \rangle)$ accepts.
- If $\langle M \rangle \notin A_{\text{ALL}}$: $L(M) \neq L(M\#)$. $\text{Oracle}(\langle M, M\# \rangle)$ rejects.

But no machine to decide A_{ALL} can exist, so neither does *Oracle*. ■

Consider the problem of grading programs that are written as exercises in programming classes. We would like to compare each student program to a “correct” program written by the instructor and accept those programs that behave identically to the one written by the instructor. Theorem 21.8 says that a perfect grading program cannot exist.

We should point out here that EqTMs is not only not in D, it is also not in SD. We leave the proof of that as an exercise.

21.2.3 Reductions That Are Not Mapping Reductions

The general definition of reducibility that we provided at the beginning of Section 21.2 is strictly more powerful than the more restricted notion of mapping reducibility that we have used in the examples that we have considered so far. We'll next consider a case where no mapping reduction exists, but a more general one does. Recall that the more general definition of a reduction from L_1 to L_2 may consist of two or more functions that can be composed to decide L_1 if *Oracle* exists and decides L_2 . We'll see that one particularly useful thing to do is to exploit a second function that applies to the output of *Oracle* and flips it (i.e., it turns an *Accept* into a *Reject* and vice versa).

Theorem 21.9 “Does M Accept No Even Length Strings?” is Undecidable

Theorem: The language $L_2 = \{\langle M \rangle : \text{Turing machine } M \text{ accepts no even length strings}\}$ is not in D.

Proof: We show that $H \leq L_2$ and so L_2 is not in D. As in the other examples we have considered so far, we need to define a reduction from H to L_2 . But this time we are going to run into a glitch. We can try to implement a straightforward mapping reduction R between H and L_2 , just as we have done for our other examples. But, when we do that and then pass its result to *Oracle*, we'll see that *Oracle* will return the opposite of the answer we need to decide H . But that is an easy problem to solve. Since *Oracle* is (claimed to be) a deciding machine, it always halts. So we can add to the reduction a second Turing machine that runs after *Oracle* and just inverts *Oracle*'s response. We'll call that second machine simply \neg .

Define:

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. Erase the tape.
 - 1.2. Write w on the tape.
 - 1.3. Run M on w .
 - 1.4. Accept.
2. Return $\langle M\# \rangle$.

$\{R, \neg\}$ is a reduction from H to L_2 . If *Oracle* exists and decides L_2 , then $C = \neg Oracle(R(\langle M, w \rangle))$ decides H . R and \neg can be implemented as a Turing machines. And C is correct. $M\#$ ignores its own input. It accepts everything or nothing, depending on whether it makes it to step 1.4. So:

- If $\langle M, w \rangle \in H$: M halts on w , so $M\#$ accepts everything, including some even length strings. $Oracle(\langle M\# \rangle)$ rejects so C accepts.
- If $\langle M, w \rangle \notin H$: M does not halt on w . $M\#$ gets stuck in step 1.3 and accepts nothing, and so, in particular, no even length strings. $Oracle(\langle M\# \rangle)$ accepts. So C rejects.

But no machine to decide H can exist, so neither does *Oracle*. So L_2 is not in D . It is also not in SD . We leave the proof of that as an exercise. ■

We have just shown that there exists a reduction from H to $L_2 = \{\langle M \rangle : \text{Turing machine } M \text{ accepts no even length strings}\}$. It is possible to prove that this is a case where it was necessary to exploit the greater power offered by the more general definition of reducibility. We leave as an exercise the proof that no mapping reduction from H to L_2 exists. To see why this might be so, it is important to keep in mind the definition of mapping reducibility: $L_1 \leq_M L_2$ iff there exists some computable function f such that:

$$\forall x (x \in L_1 \text{ iff } f(x) \in L_2).$$

Note that, if such an f exists, it is also a mapping reduction from $\neg L_1$ to $\neg L_2$.

21.3 Are All Questions About Turing Machines Undecidable?

By now it should be clear that many interesting properties of the behavior of Turing machines are undecidable. Is it true that any question that asks about a Turing machine or its behavior is undecidable? No.

First, we observe that questions that ask just about a Turing machine's physical structure, rather than about its behavior, are likely to be decidable.

Example 21.8 The Number of States of M is Decidable

Let $L_A = \{\langle M \rangle : \text{Turing machine } M \text{ contains an even number of states}\}$. L_A is decidable by the following procedure:

1. Make a pass through $\langle M \rangle$, counting the number of states in M .
2. If even, accept; else reject.

Next we'll consider two questions that do ask about a Turing machine's behavior but are, nevertheless, decidable.

Example 21.9 Whether M Halts in Some Fixed Time is Decidable

Let $L_B = \{\langle M, w \rangle : \text{Turing machine } M \text{ halts on } w \text{ within 3 steps}\}$. L_B is decidable by the following procedure:

1. Simulate M for 3 steps.
2. If it halted, accept; else reject.

Example 21.10 Exactly How M Works May be Decidable

Let $L_C = \{ \langle M, w \rangle : \text{Turing machine } M \text{ moves right exactly twice while running on } w \}$.

Notice that M must move either to the right or the left on each move. We make the usual assumption that M 's read/write head is positioned immediately to the left of the leftmost input character when M starts. If M cannot move right more than twice, it can read no more than two characters of its input. But it may loop forever moving left. As it moves left, it can write on the tape, but it cannot go back more than two squares to read what it has written. So the only part of the tape that can affect M 's future behavior is the current square, two squares to the right and two squares to the left (since all other squares to the left still contain \square). Let K be the set of states of M and let Γ be M 's tape alphabet. Then the number of effectively distinct configurations of M is $\text{maxconfigs} = |K_M| \cdot |\Gamma_M|^5$. If we simulate M running for maxconfigs moves, it will have entered, at least once, each configuration that it is ever going to reach. If it has not halted, then it is in an infinite loop. Each time through the loop it will do the same thing it did the last time.

If, in simulating maxconfigs moves, M moved right more than twice, we can reject. If it did not move right at all, or if it moved right once, we can reject. If it moved right twice, we need to find out whether either of those moves occurred during some loop. We can do that by running M for up to maxconfigs more moves. In the extreme case of a maximally long loop, it will move right once more. If there is a shorter loop, M may move right several times more. So the following procedure decides L_C :

1. Run M on w for $|K_M| \cdot |\Gamma_M|^5$ moves or until M halts or moves right three times:
 - 1.1. If M moved right exactly twice, then:
 - Run M on w for another $|K_M| \cdot |\Gamma_M|^5$ moves or until it moves right.
 - If M moved right any additional times, reject; otherwise accept.
 - 1.2. If M moved right some other number of times, reject.

What is different about languages such as L_A , L_B , and L_C (in contrast to H , H_ε , H_{ANY} , H_{ALL} , and the other languages we have proven are not in D)? The key is that, in the case of L_A , the question is not about M 's behavior at all. It involves just its structure. In the case of L_B and L_C , the question we must answer is not about the language that the Turing machine M halts on or accepts. It is about a detail of M 's behavior as it is computing. In the case of L_B , it has to do with the exact number of steps in which M might halt. In the case of L_C , it is about the way that M goes about solving the problem (specifically how often it moves right). It turns out that questions like those can be decided. We'll see, though, in Section 21.5, that we must be careful about this. Some questions that appear to be about the details of how M operates can be recast as questions about M 's output and so are not decidable.

Rice's Theorem, which we present next, articulates the difference between languages like H and languages like L_A , L_B , and L_C .

21.4 Rice's Theorem

Consider the set SD of semidecidable languages. Suppose that we want to ask any of the following questions about some language L in that set:

- Does L contain some particular string w ?
- Does L contain ε ?
- Does L contain any strings at all?
- Does L contain all strings over some alphabet Σ ?

In order to consider building a program to answer any of those questions, we first need a way to specify formally what L is. Since the SD languages are, by definition, exactly the languages that can be semidecided by some Turing machine, one way to specify a language L is to give a semideciding Turing machine for it. If we do that, then we can restate each of those questions as:

- Given a semideciding Turing machine M , does M accept some particular string w ?
- Given a semideciding Turing machine M , does M accept ε ?
- Given a semideciding Turing machine M , does M accept anything?

- Given a semideciding Turing machine M , does M accept all strings in Σ^* ?

We can encode each of those decision problems as a language to be decided, yielding:

- $A = \{ \langle M, w \rangle : \text{Turing machine } M \text{ accepts } w \}$.
- $A_\epsilon = \{ \langle M \rangle : \text{Turing machine } M \text{ accepts } \epsilon \}$.
- $A_{\text{ANY}} = \{ \langle M \rangle : \text{there exists at least one string that Turing machine } M \text{ accepts} \}$.
- $A_{\text{ALL}} = \{ \langle M \rangle : \text{Turing machine } M \text{ accepts all inputs} \}$.

We have already seen that none of these languages is in D , so none of the corresponding questions is decidable. Rice's Theorem, which we are about to state and prove, tells us that not only these languages, but any language that can be described as $\{ \langle M \rangle : P(L(M)) = \text{True} \}$, for any nontrivial property P , is not in D . By a *nontrivial property* we mean a property that is not simply *True* for all languages or *False* for all languages.

But we can state Rice's Theorem even more generally than that. The questions we have just considered are questions we can ask of any semidecidable language, independently of how we describe it. We have used semideciding Turing machines as our descriptions. But we could use some other descriptive form. (For example, in Chapter 23, we will consider a grammar formalism that describes exactly the SD languages.) The key is that the property we are evaluating is a property of the language itself and not a property of some particular Turing machine that happens to semidecide it.

So an alternative way to state Rice's Theorem is:

No nontrivial property of the SD languages is decidable.

Just as languages that are defined in terms of the behavior of Turing machines are generally not decidable, functions that describe the way that Turing machines behave are likely not to be computable. See, for example, the *busy beaver functions* described in Section 25.1.4.

To use Rice's Theorem to show that a language L of the form $\{ \langle M \rangle : P(L(M)) = \text{True} \}$ is not in D we must:

- Specify property P .
- Show that the domain of P is the set of SD languages.
- Show that P is nontrivial:
 - P is true of at least one language.
 - P is false of at least one language.

Let M , M_a , and M_b be Turing machines. We'll consider each of the following languages and see whether Rice's Theorem applies to it:

1. $\{ \langle M \rangle : M \text{ is a Turing machine and } L(M) \text{ contains only even length strings} \}$.
2. $\{ \langle M \rangle : M \text{ is a Turing machine and } L(M) \text{ contains an odd number of strings} \}$.
3. $\{ \langle M \rangle : M \text{ is a Turing machine and } L(M) \text{ contains all strings that start with } a \}$.
4. $\{ \langle M \rangle : M \text{ is a Turing machine and } L(M) \text{ is infinite} \}$.
5. $\{ \langle M \rangle : M \text{ is a Turing machine and } L(M) \text{ is regular} \}$.
6. $\{ \langle M \rangle : \text{Turing machine } M \text{ contains an even number of states} \}$.
7. $\{ \langle M \rangle : \text{Turing machine } M \text{ has an odd number of symbols in its tape alphabet} \}$.
8. $\{ \langle M \rangle : \text{Turing machine } M \text{ accepts } \epsilon \text{ within 100 steps} \}$.
9. $\{ \langle M \rangle : \text{Turing machine } M \text{ accepts } \epsilon \}$.
10. $\{ \langle M_a, M_b \rangle : M_a \text{ and } M_b \text{ are Turing machines and } L(M_a) = L(M_b) \}$.

In cases 1-5, we can easily state P . For example, in case 1, P is, "*True* if L contains only even length strings and *False* otherwise". In all five cases, the domain of P is the set of SD languages and P is nontrivial. For example, in case 1, P is *True* of $\{aa, bb\}$ and *False* of $\{a, aa\}$.

But now consider cases 6-8. In case 6, P is, “*True* if M has an even number of states and *False* otherwise”. P is no longer a property of a language. It is a property of some specific machine, independent of the language that the machine accepts. The same is true in cases 7 and 8. So Rice’s Theorem tells us nothing about whether those languages are in D . They may or may not be. As it turns out, all three of these examples are in D and languages that look like them usually are. But Rice’s Theorem does not tell us that. It simply tells us nothing.

Next consider case 9. In form, it looks something like case 8. But it is in fact like 1-5. An alternative way to state P is, “ $\varepsilon \in L(M)$ ”. It is not the wording of the description that matters. It is the property P itself that counts.

Finally consider case 10. We have already shown that this language, which we have named EqTMs, is not in D . But Rice’s Theorem does not tell us that. Again, it says nothing since now we are asking about a property whose domain is $SD \times SD$, rather than simply SD . So when Rice’s Theorem doesn’t apply, it is possible that we are dealing with a language in D . It is also possible we are dealing with one not in D . Without additional investigation, we just don’t know.

Rice’s Theorem is not going to give us a way to prove anything we couldn’t have proven with reduction. Although it is an alternative proof strategy, its main value is its insight. We know immediately, when confronted with a question about the SD languages, that it will not be decidable.

The proof of Rice’s Theorem is by reduction from H . It is a bit more complex than any of the reductions we have done so far, but the principle is exactly the same. What we are going to do is to show that if it were possible to decide *any* property P (without regard to what P is except that it is nontrivial), then it would be possible to decide H . It may seem surprising that we can show this without appeal to any information about what P tells us. But we can.

Theorem 21.10 Rice’s Theorem

Theorem: For any nontrivial P , the language $L = \{ \langle M \rangle : P(L(M)) = \textit{True} \}$ is not in D .

Proof: We prove Rice’s Theorem by showing that $H \leq_M L$. Let P be any nontrivial property of the SD languages. We do not know what P is. But, whatever it is, either $P(\emptyset) = \textit{True}$ or $P(\emptyset) = \textit{False}$. Assume it is *False*. We leave the proof that the theorem holds if $P(\emptyset) = \textit{True}$ as an exercise. Since P is nontrivial, there is some SD language L_T such that $P(L_T)$ is *True*. Since L_T is in SD , there exists some Turing machine K that semidecides it.

We need to define a mapping reduction R from H to L . The main idea in this reduction is that R will build a machine $M\#$ that first runs M on w as a sort of filter. If it makes it by that step, then it considers its own input and either accepts it or not. If we can design $M\#$ ’s action at that second step so that we can tell whether it makes it there, we will know whether M halted on w .

Let R be a reduction from H to L defined as follows:

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. Copy its input x to a second tape.
 - 1.2. Erase the tape.
 - 1.3. Write w on the tape.
 - 1.4. Run M on w .
 - 1.5. Put x back on the first tape and run K (the Turing machine that semidecides L_T , a language of which P is *True*) on x .
2. Return $\langle M\# \rangle$.

If *Oracle* exists and decides L , then $C = \textit{Oracle}(R(\langle M, w \rangle))$ decides H . R can be implemented as a Turing machine. And C is correct:

- If $\langle M, w \rangle \in H$: M halts on w , so $M\#$ makes it to step 1.5. So $M\#$ does whatever K would do. So $L(M\#) = L(K)$ and $P(L(M\#)) = P(L(K))$. We chose K precisely to assure that $P(L(K))$ is *True*, so $P(L(M\#))$ must also be *True*. *Oracle* decides P . *Oracle*($\langle M\# \rangle$) accepts.

- If $\langle M, w \rangle \notin H$: M does not halt on w . $M\#$ gets stuck in step 1.4 and so accepts nothing. $L(M\#) = \emptyset$. By assumption, $P(\emptyset) = \text{False}$. Oracle decides P . Oracle($\langle M\# \rangle$) rejects.

But no machine to decide H can exist, so neither does Oracle. ■

Now that we have proven the theorem, we can use it as an alternative to reduction in proving that a language L is not in D .

Theorem 21.11 “Is $L(M)$ Regular?” is Undecidable

Theorem: Given a Turing machine M , the question, “Is $L(M)$ regular?” is not decidable. Alternatively, the language $\text{TM}_{\text{REG}} = \{\langle M \rangle : M \text{ is a Turing machine and } L(M) \text{ is regular}\}$ is not in D .

Proof: By Rice’s Theorem. We define P as follows:

- Let P be defined on the set of languages accepted by some Turing machine M . Let it be *True* if $L(M)$ is regular and *False* otherwise.
- The domain of P is the set of SD languages since it is the set of languages accepted by some Turing machine.
- P is nontrivial:
 - $P(a^*) = \text{True}$.
 - $P(A^n B^n) = \text{False}$.

Thus we can conclude that $\{\langle M \rangle : M \text{ is a Turing machine and } L(M) \text{ is regular}\}$ is not in D .

We can also prove this by reduction from H . The reduction we will use exploits two functions, R and \neg . R will map instances of H to instances of TM_{REG} . It will use a strategy that is very similar to the one we used in proving Rice’s Theorem. As in the proof of Theorem 21.9, \neg will simply invert Oracle’s response (turning an *Accept* into a *Reject* and vice versa). So define:

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. Copy its input x to a second tape.
 - 1.2. Erase the tape.
 - 1.3. Write w on the tape.
 - 1.4. Run M on w .
 - 1.5. Put x back on the first tape.
 - 1.6. If $x \in A^n B^n$ then accept, else reject.
2. Return $\langle M\# \rangle$.

$\{R, \neg\}$ is a reduction from H to TM_{REG} . If Oracle exists and decides TM_{REG} , then $C = \neg \text{Oracle}(R(\langle M, w \rangle))$ decides H . R and \neg can be implemented as Turing machines. In particular, it is straightforward to build a Turing machine that decides whether a string x is in $A^n B^n$. And C is correct:

- If $\langle M, w \rangle \in H$: M halts on w , so $M\#$ makes it to step 1.5. Then it accepts x iff $x \in A^n B^n$. So $M\#$ accepts $A^n B^n$, which is not regular. Oracle($\langle M\# \rangle$) rejects. C accepts.
- If $\langle M, w \rangle \notin H$: M does not halt on w . $M\#$ gets stuck in step 1.4 and so accepts nothing. $L(M\#) = \emptyset$, which is regular. Oracle($\langle M\# \rangle$) accepts. C rejects.

But no machine to decide H can exist, so neither does Oracle. ■

It turns out that we can also make a stronger statement about TM_{REG} . It is not only not in D , it is not in SD . We leave the proof of that as an exercise.

21.5 Undecidable Questions About Real Programs

The real practical impact of the undecidability results that we have just presented is the following: The programming environments that we actually use every day are equal in computational power to the Turing machine. So questions that are undecidable when asked about Turing machines are equally undecidable when asked about Java programs or C++ programs, or whatever. The undecidability of a question about real programs can be proved by reduction from the corresponding question about Turing machines. We'll show one example of doing this.

Theorem 21.12 “Are Two Programs Equivalent?” is Undecidable

Theorem: The language $\text{EqPrograms} = \{ \langle P_a, P_b \rangle : P_a, P_b \text{ are programs in any standard programming language } PL \text{ and } L(P_a) = L(P_b) \}$ is not in D.

Proof: Recall that $\text{EqTMs} = \{ \langle M_a, M_b \rangle : M_a \text{ and } M_b \text{ are Turing machines and } L(M_a) = L(M_b) \}$. We show that $\text{EqTMs} \leq_M \text{EqPrograms}$ and so EqPrograms is not in D (since EqTMs isn't). It is straightforward to build, in any standard programming language, an implementation of the universal Turing machine U . Call that program SimUM . Now let R be a mapping reduction from EqTMs to EqPrograms defined as follows:

$R(\langle M_a, M_b \rangle) =$

1. Let P_1 be a PL program that, on input w , invokes $\text{SimUM}(M_a, w)$ and returns its result.
2. Let P_2 be a PL program that, on input w , invokes $\text{SimUM}(M_b, w)$ and returns its result.
3. Return $\langle P_1, P_2 \rangle$.

If Oracle exists and decides EqPrograms , then $C = \text{Oracle}(R(\langle M_a, M_b \rangle))$ decides EqTMs . R can be implemented as a Turing machine. And C is correct. $L(P_1) = L(M_a)$ and $L(P_2) = L(M_b)$. So:

- If $\langle M_a, M_b \rangle \in \text{EqTMs}$: $L(M_a) = L(M_b)$. So $L(P_1) = L(P_2)$. $\text{Oracle}(\langle P_1, P_2 \rangle)$ accepts.
- If $\langle M_a, M_b \rangle \notin \text{EqTMs}$: $L(M_a) \neq L(M_b)$. So $L(P_1) \neq L(P_2)$. $\text{Oracle}(\langle P_1, P_2 \rangle)$ rejects.

But no machine to decide EqTMs can exist, so neither does Oracle . ■

The United States Patent Office issues patents on software. But, before the Patent Office can issue any patent, it must check for prior art. The theorem we have just proved suggests that there can exist no general purpose program that can do that checking automatically.

Because the undecidability of questions about real programs follows from the undecidability of those questions for Turing machines, we can show, for example, that all of the following questions are undecidable:

1. Given a program P and input x , does P , when running on x , halt?
2. Given a program P , might P get into an infinite loop on some input?
3. Given a program P and input x , does P , when running on x , ever output a 0? Or anything at all?
4. Given two programs, P_1 and P_2 , are they equivalent?
5. Given a program P , input x , and a variable n , does P , when running on x , ever assign a value to n ? We need to be able to answer this question if we want to be able to guarantee that every variable is initialized before it is used.
6. Given a program P and code segment S in P , does P ever reach S on any input (in other words, can we chop S out)?
7. Given a program P and code segment S in P , does P reach S on every input (in other words, can we guarantee that S happens)?

We've already proved that questions 1, 2, and 4 are undecidable for Turing machines. Question 3 (about printing 0) is one that Turing himself asked and showed to be undecidable. We leave that proof as an exercise.

Is it possible to build a program verification system that can determine, given an arbitrary specification S and program P whether or not P correctly implements S ? $\text{C } 678$.

But what about questions 5, 6, and 7? They appear to be about details of how a program operates, rather than about the result of running the program (i.e., the language it accepts or the function it computes). We know that many questions of that sort are decidable, either by inspecting the program or by running it for some bounded number of steps. So why are these questions undecidable? Because they cannot be answered either by inspection or by bounded simulation. We can prove that each of them is undecidable by showing that some language that we already know is not in D can be reduced to it. To do this, we'll return to the Turing machine representation for programs. We'll show that question 6 is undecidable and leave the others as exercises.

Can a compiler check for dead code and eliminate it? $\text{C } 669$.

Theorem 21.13 “Does M Ever Reach Some Particular State?” is Undecidable

Theorem: The language $L = \{ \langle M, q \rangle : \text{Turing machine } M \text{ reaches state } q \text{ on some input} \}$ is not in D .

Proof: We show that $H_{\text{ANY}} \leq_M L$ and so L is not in D . Let R be a mapping reduction from H_{ANY} to L defined as follows:

$R(\langle M \rangle) =$

1. From $\langle M \rangle$, construct the description $\langle M\# \rangle$ of a new Turing machine $M\#$ that will be identical to M except that, if M has a transition $((q_1, c_1), (q_2, c_2, a))$ and q_2 is a halting state other than h , replace that transition with $((q_1, c_1), (h, c_2, a))$.
2. Return $\langle M\#, h \rangle$.

If *Oracle* exists and decides L , then $C = \text{Oracle}(R(\langle M \rangle))$ decides H_{ANY} . R can be implemented as a Turing machine. And C is correct: $M\#$ will reach the halting state h iff M would reach some halting state. So:

- If $\langle M \rangle \in H_{\text{ANY}}$: There is some string on which M halts. So there is some string on which $M\#$ reaches state h . $\text{Oracle}(\langle M\#, h \rangle)$ accepts.
- If $\langle M \rangle \notin H_{\text{ANY}}$: There is no string on which M halts. So there is no string on which $M\#$ reaches state h . $\text{Oracle}(\langle M\#, h \rangle)$ rejects.

But no machine to decide H_{ANY} can exist, so neither does *Oracle*. ■

21.6 Showing That a Language is Not Semidecidable

We know, from Theorem 20.3, that there exist languages that are not in SD . In fact, we know that there are uncountably many of them. And we have seen one specific example, $\neg H$. In this section we will see how to show that other languages are also not in SD . Although we will first discuss a couple of other methods of proving that a language is not in SD (which we will also write as in $\neg SD$), we will again make extensive use of reduction. This time, the basis for our reduction proofs will be $\neg H$. We will show that if some new language L were in SD , $\neg H$ would also be. But it is not.

Before we try to prove that a language L is not in SD (or that it is), we need an intuition that tells us what to prove. A good way to develop such an intuition is to think about trying to write a program to solve the problem. Languages that are not in SD generally involve either infinite search, or knowing that a Turing machine will infinite loop, or both. For example, the following languages are not in SD :

- $\neg H = \{ \langle M, w \rangle : \text{Turing machine } M \text{ does not halt on } w \}$. To solve this one by simulation, we would have to run M forever.
- $\{ \langle M \rangle : L(M) = \Sigma^* \}$. To solve this one by simulation, we would have to try all strings in Σ^* . But there are infinitely many of them.

- $\{ \langle M \rangle : \text{there does not exist a string on which Turing machine } M \text{ halts} \}$. To solve this one by simulation, we would have to try an infinite number of strings and show that all of them fail to halt. Even to show that one fails to halt would require an infinite number of steps.

In the rest of this section, we present a collection of techniques that can be used to prove that a language is not in SD.

21.6.1 Techniques Other Than Reduction ★

Sometimes we can show that a language L is not in SD by giving a proof by contradiction that does not exploit reduction. We will show one such example here. For this example, we need to make use of a theorem that we will prove in Section 25.3: the recursion theorem tells us that there exists a subroutine, *obtainSelf*, available to any Turing machine M , that constructs $\langle M \rangle$, the description of M .

We have not so far said anything about minimizing Turing machines. The reason is that no algorithm to do so exists. In fact, given a Turing machine M , it is undecidable whether M is minimal. Alternatively, the language of descriptions of minimal Turing machines is not in SD. More precisely, define a Turing machine M to be *minimal* iff there exists no other Turing machine M' such that $|\langle M' \rangle| < |\langle M \rangle|$ and M' is equivalent to M .

Theorem 21.14 “Is M Minimal?” is Not Semidecidable

Theorem: The language $\text{TM}_{\text{MIN}} = \{ \langle M \rangle : \text{Turing machine } M \text{ is minimal} \}$ is not in SD.

Proof: If TM_{MIN} were in SD, then (by Theorem 20.8) there would exist some Turing machine $ENUM$ that enumerates its elements. Define the following Turing machine:

$M\#(x) =$

1. Invoke *obtainSelf* to produce $\langle M\# \rangle$.
2. Run $ENUM$ until it generates the description of some Turing machine M' whose description is longer than $|\langle M\# \rangle|$.
3. Invoke the universal Turing machine U on the string $\langle M', x \rangle$.

Since TM_{MIN} is infinite, $ENUM$ must eventually generate a string that is longer than $|\langle M\# \rangle|$. So $M\#$ makes it to step 3 and so is equivalent to M' since it simulates M' . But, since $|\langle M\# \rangle| < |\langle M' \rangle|$, M' cannot be minimal. Yet it was generated by $ENUM$. Contradiction. ■

Another way to prove that a language is not in SD is to exploit Theorem 20.6, which tells us that a language L is in D iff both it and its complement, $\neg L$, are in SD. This is true because, if we could semidecide both L and $\neg L$, we could run the two semideciders in parallel, wait until one of them halts, and then either accept (if the semidecider for L accepted) or reject (if the semidecider for $\neg L$ accepted).

So suppose that we are considering some language L . We want to know whether L is in SD and we already know:

- $\neg L$ is in SD, and
- At least one of L or $\neg L$ is not in D.

Then we can conclude that L is not in SD, because, if it were, it would force both itself and its complement into D, and we know that cannot be true. This is the technique that we used to prove that $\neg H$ is not in SD. We can use it for some other languages as well, which we will do in our proof of the next theorem.

Theorem 21.15 “Does There Exist No String on Which M Halts?” is Not Semidecidable

Theorem: $H_{\text{-ANY}} = \{ \langle M \rangle : \text{there does not exist any string on which Turing machine } M \text{ halts} \}$ is not in SD.

Proof: Recall that we said, at the beginning of Chapter 19, that we would define the complement of a language of Turing machine descriptions with respect to the universe of syntactically valid Turing machine descriptions. So the

complement of $H_{\neg ANY}$ is $H_{ANY} = \{ \langle M \rangle : \text{there exists at least one string on which Turing machine } M \text{ halts} \}$. From Theorem 21.2, we know:

- $\neg H_{\neg ANY}$ (namely, H_{ANY}) is in SD.
- $\neg H_{\neg ANY}$ (namely, H_{ANY}) is not in D.

So $H_{\neg ANY}$ is not in SD because, if it were, then H_{ANY} would be in D but it isn't. ■

21.6.2 Reduction

The most general technique that we can use for showing that a language is not in SD is reduction. Our argument will be analogous to the one we used to show that a language is not in D. It is:

To prove that a language L_2 is not in SD, find a reduction R :

- from some language L_1 that is already known not to be in SD,
- to L_2 .

If L_2 were in SD, then there would exist some Turing machine *Oracle* that semidecides it. Then the composition of R and *Oracle* would semidecide L_1 . But there can exist no Turing machine that semidecides L_1 . So *Oracle* cannot exist. So L_2 is not in SD.

There are two differences between reductions that show that a language is not in SD and those that show that a language not in D:

1. We must choose to reduce from a language that is already known not to be in SD (as opposed to choosing one where all we can prove is that it is not in D). We already have one example of such a language: $\neg H$. So we have a place to start.
2. We hypothesize the existence of a *semideciding* machine *Oracle*, rather than a deciding one.

The second of these will sometimes turn out to be critical. In particular, the function \neg (which inverts the output of *Oracle*), can no longer be implemented as a Turing machine. Since *Oracle* is claimed only to be a semideciding machine, there is no guarantee that it halts. Since *Oracle* may loop, there is no way to write a procedure that accepts iff *Oracle* doesn't. So we won't be able to include \neg in any of our reductions.

We will need to find a way around this problem when it arises. But let's first do a very simple example where there is no need to do the inversion. We begin with a reduction proof of Theorem 21.15. We show that $\neg H \leq_M H_{\neg ANY}$. Let R be a mapping reduction from $\neg H$ to $H_{\neg ANY}$ defined as follows.

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. Erase the tape.
 - 1.2. Write w on the tape.
 - 1.3. Run M on w .
2. Return $\langle M\# \rangle$.

If *Oracle* exists and semidecides $H_{\neg ANY}$, then $C = Oracle(R(\langle M, w \rangle))$ semidecides $\neg H$. R can be implemented as a Turing machine. And C is correct: $M\#$ ignores its input. It halts on everything or nothing, depending on whether M halts on w . So:

- If $\langle M, w \rangle \in \neg H$: M does not halt on w , so $M\#$ halts on nothing. $Oracle(\langle M\# \rangle)$ accepts.
- If $\langle M, w \rangle \notin \neg H$: M halts on w , so $M\#$ halts on everything. $Oracle(\langle M\# \rangle)$ does not accept.

But no machine to semidecide $\neg H$ can exist, so neither does *Oracle*.

Straightforward reductions of this sort can be used to show that many other languages, particularly those that are defined by the failure of a Turing machine to halt, are also not in SD.

Theorem 21.16 “Does M Fail to Halt on ϵ ?” is Not Semidecidable

Theorem: $\neg H_\epsilon = \{ \langle M \rangle : \text{Turing machine } M \text{ does not halt on } \epsilon \}$ is not in SD.

Proof: The proof is by reduction from $\neg H$. We leave it as an exercise. ■

Sometimes, however, finding a reduction that works is a bit more difficult. We next consider the language:

- $A_{\text{anbn}} = \{ \langle M \rangle : M \text{ is a Turing machine and } L(M) = A^n B^n = \{ a^n b^n : n \geq 0 \} \}$.

Note that A_{anbn} contains strings that look like:

$(q_{00}, a_{00}, q_{01}, a_{00}, \rightarrow), (q_{00}, a_{01}, q_{00}, a_{10}, \rightarrow), (q_{00}, a_{10}, q_{01}, a_{01}, \leftarrow), (q_{00}, a_{11}, q_{01}, a_{10}, \leftarrow),$
 $(q_{01}, a_{00}, q_{00}, a_{01}, \rightarrow), (q_{01}, a_{01}, q_{01}, a_{10}, \rightarrow), (q_{01}, a_{10}, q_{01}, a_{11}, \leftarrow), (q_{01}, a_{11}, q_{11}, a_{01}, \leftarrow)$

It does not contain strings like $aaabbbb$. But $A^n B^n$ does.

We are going to have to try a couple of times to find a correct reduction that can be used to prove that A_{anbn} is not in SD.

Theorem 21.17 “Is $L(M) = A^n B^n$?” is Not Semidecidable

Theorem: The language $A_{\text{anbn}} = \{ \langle M \rangle : M \text{ is a Turing machine and } L(M) = A^n B^n \}$ is not in SD.

Proof: We show that $\neg H \leq_M A_{\text{anbn}}$ and so A_{anbn} is not in SD. We will build a mapping reduction R from $\neg H$ to A_{anbn} . R needs to construct the description of a new Turing machine $M\#$ so that $M\#$ is an acceptor for $A^n B^n$ if M does not halt on w and it is something else if M does halt on w . We can try the simple $M\#$ that first runs M on w as a gate that controls access to the rest of the program:

Reduction Attempt 1: Define:

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. Copy the input x to a second tape.
 - 1.2. Erase the tape.
 - 1.3. Write w on the tape.
 - 1.4. Run M on w .
 - 1.5. Put x back on the first tape.
 - 1.6. If $x \in A^n B^n$ then accept; else loop.
2. Return $\langle M\# \rangle$.

Now we must show that, if some Turing machine $Oracle$ semidecides A_{anbn} , then $C = Oracle(R(\langle M, w \rangle))$ semidecides $\neg H$. But we encounter a problem when we try to show that C is correct. If M halts on w , then $M\#$ makes it to step 1.5 and becomes an $A^n B^n$ acceptor, so $Oracle(\langle M\# \rangle)$ accepts. If M does not halt on w , then $M\#$ accepts nothing. It is therefore not an $A^n B^n$ acceptor, so $Oracle(\langle M\# \rangle)$ does not accept. The reduction R has succeeded in capturing the correct distinction: $Oracle$ returns one answer when $\langle M, w \rangle \in \neg H$ and another answer when $\langle M, w \rangle \notin \neg H$. But the answer is backwards. And this time we can't simply add the function \neg to the reduction and define C to return $\neg Oracle(R(\langle M, w \rangle))$. $Oracle$ is only hypothesized to be a semideciding machine so there is no way for \neg to accept if $Oracle$ fails to accept (since it may loop).

There is an easy way to fix this. We build $M\#$ so that it either accepts just A^nB^n (if M does not halt on w) or everything (if M does halt on w). We make that happen by putting the gate *after* the code that accepts A^nB^n instead of before.

Reduction Attempt 2: Define:

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. If $x \in A^nB^n$ then accept. Else:
 - 1.2. Erase the tape.
 - 1.3. Write w on the tape.
 - 1.4. Run M on w .
 - 1.5. Accept.
2. Return $\langle M\# \rangle$.

If *Oracle* exists and semidecides A_{anbn} , then $C = \text{Oracle}(R(\langle M, w \rangle))$ semidecides $\neg H$. R can be implemented as a Turing machine. And C is correct: $M\#$ immediately accepts all strings in A^nB^n . If M does not halt on w , those are the only strings that $M\#$ accepts. If M does halt on w , $M\#$ accepts everything. So:

- If $\langle M, w \rangle \in \neg H$: M does not halt on w , so $M\#$ accepts A^nB^n in step 1.1. Then it gets stuck in step 1.4, so it accepts nothing else. It is an A^nB^n acceptor. $\text{Oracle}(\langle M\# \rangle)$ accepts.
- If $\langle M, w \rangle \notin \neg H$: M halts on w , so $M\#$ accepts everything. $L(M\#) \neq A^nB^n$. $\text{Oracle}(\langle M\# \rangle)$ does not accept.

But no machine to semidecide $\neg H$ can exist, so neither does *Oracle*. ■

Sometimes, however, the simple gate technique doesn't work, as we will see in the next example.

Theorem 21.18 “Does M Halt on Everything?” is Not Semidecidable

Theorem: $H_{\text{ALL}} = \{\langle M \rangle : \text{Turing machine } M \text{ halts on } \Sigma^*\}$ is not in SD.

Proof: We show that H_{ALL} is not in SD by reduction from $\neg H$.

Reduction Attempt 1: Define:

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. Erase the tape.
 - 1.2. Write w on the tape.
 - 1.3. Run M on w .
2. Return $\langle M\# \rangle$.

We can attempt to show that, if *Oracle* exists and semidecides H_{ALL} , then $C = \text{Oracle}(R(\langle M, w \rangle))$ correctly semidecides $\neg H$. The problem is that it doesn't:

- If $\langle M, w \rangle \in \neg H$: M does not halt on w , so $M\#$ gets stuck in step 1.3 and halts on nothing. $\text{Oracle}(\langle M\# \rangle)$ does not accept.
- If $\langle M, w \rangle \notin \neg H$: M halts on w , so $M\#$ halts on everything. $\text{Oracle}(\langle M\# \rangle)$ accepts.

This is backwards. We could try halting on something before running M on w , the way we did on the previous example. But the only way to make $M\#$ into a machine that halts on everything would be to have it halt immediately, before running M . But then its behavior would not depend on whether M halts on w . We need a new technique.

Reduction Attempt 2: Define:

$R(\langle M, w \rangle) =$

1. Construct the description $\langle M\# \rangle$ of a new Turing machine $M\#(x)$ that, on input x , operates as follows:
 - 1.1. Copy the input x to a second tape.
 - 1.2. Erase the tape.
 - 1.3. Write w on the tape.
 - 1.4. Run M on w for $|x|$ steps or until it halts.
 - 1.5. If M would have halted, then loop.
 - 1.6. Else halt.
2. Return $\langle M\# \rangle$.

We build $M\#$ so that it runs the simulation of M on w for some finite number of steps and observes whether M would have halted in that time. If M would have halted, $M\#$ loops. If M would *not* have halted, $M\#$ promptly halts. This is where we flip from halting to looping and vice versa.. It works because the simulation always halts, so $M\#$ never gets stuck running it. But for how many steps should we run the simulation? If M is going to halt, we don't know how long it will take for it to do so. We need to guarantee that we don't quit too soon and think that M isn't going to halt when it actually will. Here's the insight: The language Σ^* is infinite. So if $M\#$ is going to halt on every string in Σ^* , it will have to halt on an infinite number of strings. It's okay if $M\#$ gets fooled into thinking that M will halt some of the time as long as it does not do so for all possible inputs. So $M\#$ will run the simulation of M on w for a number of steps equal to the length of its ($M\#'s$) own input. It may be fooled into thinking M is going to halt on w when it is invoked on short strings. But, if M does eventually halt, it does so in some number of steps n . When started on any strings of length n or more, $M\#$ will try long enough and will discover that M on w would have halted. Then it will loop. So it will not halt on all strings in Σ^* .

If *Oracle* exists and semidecides H_{ALL} , then $C = Oracle(R(\langle M, w \rangle))$ semidecides $\neg H$. R can be implemented as a Turing machine. And C is correct:

- If $\langle M, w \rangle \in \neg H$: M does not halt on w . So, no matter how long x is, M will not halt in $|x|$ steps. So, for all inputs x , $M\#$ makes it to step 1.6. So it halts on everything. $Oracle(\langle M\# \rangle)$ accepts.
- If $\langle M, w \rangle \notin \neg H$: M halts on w . It does so in some number of steps n . On inputs of length less than n , $M\#$ will make it to step 1.6 and halt. But on all inputs of length n or greater, $M\#$ will loop in step 1.5. So it fails to halt on everything. $Oracle(\langle M\# \rangle)$ does not accept.

But no machine to semidecide $\neg H$ can exist, so neither does *Oracle*. ■

21.6.3 Is L in D , SD/D , or $\neg SD$?

Throughout this chapter, we have seen examples of languages that are decidable (i.e., they are in D), are semidecidable but not decidable (i.e., they are in SD/D), and, most recently, are not even semidecidable (i.e., they are in $\neg SD$). We have seen some heuristics that are useful for analyzing a language and determining what class it is in. In applying those heuristics, it is critical that we look closely at the language definition. Small changes can make a big difference to the decidability of the language. For example, consider the following four languages (where, in each case, M is a Turing machine):

1. $\{ \langle M \rangle : M \text{ has an even number of states} \}$.
2. $\{ \langle M \rangle : |\langle M \rangle| \text{ is even} \}$.
3. $\{ \langle M \rangle : |L(M)| \text{ is even} \}$ (i.e., $L(M)$ contains an even number of strings).
4. $\{ \langle M \rangle : M \text{ accepts all even length strings} \}$.

Language 1 is in D . A simple examination of $\langle M \rangle$ will tell us how many states M has. Language 2 is also in D . To decide it, all we need to do is to examine $\langle M \rangle$, the string description of M , and determine whether that string is of even length. Rice's Theorem does not apply in either of those cases since the property we care about involves the physical Turing machine M , not the language it accepts. But languages 3 and 4 are different. To decide either of them requires evaluating a property of the language that a Turing machine M accepts. Rice's Theorem tells us, therefore,

that neither of them is in D . In fact, neither of them is in SD either. The intuition here is that to semidecide them by simulation would require trying an infinite number of strings. We leave the proof of this claim as an exercise.

Now consider another set:

1. $\{ \langle M, w \rangle : \text{Turing machine } M \text{ does not halt on input string } w \}$. (This is just $\neg H$.)
2. $\{ \langle M, w \rangle : \text{Turing machine } M \text{ rejects } w \}$.
3. $\{ \langle M, w \rangle : \text{Turing machine } M \text{ is a deciding Turing machine and } M \text{ rejects } w \}$.

We know that $\neg H$ is in $\neg SD$. What about language 2? It seems similar. But it is different in a crucial way and is therefore in SD/D . The following Turing machine semidecides it: run M on w ; if it halts and rejects, accept. The key difference is that now, instead of needing to detect that M loops, we need to detect that M halts and rejects. We can detect halting (but not looping) by simulation. Now consider language 3. If it were somehow possible to know that M were a deciding machine, then there would be a decision procedure to determine whether or not it rejects w : run M on w . It must halt (since it's a deciding machine). If it rejects, accept, else reject. That would mean language 3 would be in D . But language 3 is, in fact, in $\neg SD$. It is harder than language 2. The problem is that there is not even a semideciding procedure for the question, "Is M a deciding machine?" That question is equivalent to asking whether M halts on all inputs, which we have shown is not semidecidable.

21.7 Summary of D , SD/D and $\neg SD$ Languages that Include Turing Machine Descriptions

At the beginning of this chapter, we presented a table with a set of questions that we might like to ask about Turing machines and we showed the language formulation of each question. We have now proven where most of those languages fall in the D , SD/D , $\neg SD$ hierarchy that we have defined. (The rest are given as exercises.) So we know whether there exists a decision procedure, a semidecision procedure, or neither, to answer the corresponding question. Because many of these questions are very important as we try to understand the power of the Turing machine formalism, we summarize in Table 21.2 the status of those initial questions, along with some of the others that we have considered in this chapter.

<i>The Problem View</i>	<i>The Language View</i>	<i>Status</i>
Does TM M have an even number of states?	$\{ \langle M \rangle : \text{TM } M \text{ has an even number of states} \}$	D
Does TM M halt on w ?	$H = \{ \langle M, w \rangle : \text{TM } M \text{ halts on } w \}$	SD/D
Does TM M halt on the empty tape?	$H_\varepsilon = \{ \langle M \rangle : \text{TM } M \text{ halts on } \varepsilon \}$	SD/D
Is there any string on which TM M halts?	$H_{\text{ANY}} = \{ \langle M \rangle : \text{there exists at least one string on which TM } M \text{ halts} \}$	SD/D
Does TM M halt on all strings?	$H_{\text{ALL}} = \{ \langle M \rangle : \text{TM } M \text{ halts on } \Sigma^* \}$	\neg SD
Does TM M accept w ?	$A = \{ \langle M, w \rangle : \text{TM } M \text{ accepts } w \}$	SD/D
Does TM M accept ε ?	$A_\varepsilon = \{ \langle M \rangle : \text{TM } M \text{ accepts } \varepsilon \}$	SD/D
Is there any string that TM M accepts?	$A_{\text{ANY}} = \{ \langle M \rangle : \text{there exists at least one string that TM } M \text{ accepts} \}$	SD/D
Does TM M fail to halt on w ?	$\neg H = \{ \langle M, w \rangle : \text{TM } M \text{ does not halt on } w \}$	\neg SD
Does TM M accept all strings?	$A_{\text{ALL}} = \{ \langle M \rangle : L(M) = \Sigma^* \}$	\neg SD
Do TMs M_a and M_b accept the same languages?	$\text{EqTMs} = \{ \langle M_a, M_b \rangle : L(M_a) = L(M_b) \}$	\neg SD
Is it the case that TM M does not halt on any string?	$H_{\neg \text{ANY}} = \{ \langle M \rangle : \text{there does not exist any string on which TM } M \text{ halts} \}$	\neg SD
Does TM M fail to halt on its own description?	$\{ \langle M \rangle : \text{TM } M \text{ does not halt on input } \langle M \rangle \}$	\neg SD
Is TM M minimal?	$\text{TM}_{\text{MIN}} = \{ \langle M \rangle : \text{TM } M \text{ is minimal} \}$	\neg SD
Is the language that TM M accepts regular?	$\text{TM}_{\text{REG}} = \{ \langle M \rangle : L(M) \text{ is regular} \}$	\neg SD
Is $L(M) = A^n B^n$?	$A_{\text{anbn}} = \{ \langle M \rangle : L(M) = A^n B^n \}$	\neg SD

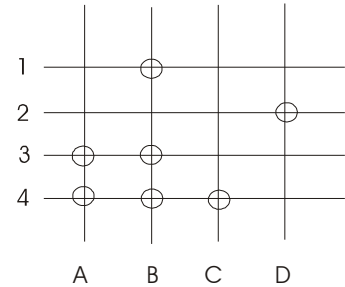
Table 21.2 The problem and the language views

21.8 Exercises

- 1) For each of the following languages L , state whether L is in D, SD/D, or \neg SD. Prove your claim. Assume that any input of the form $\langle M \rangle$ is a description of a Turing machine.
 - a) $\{ a \}$.
 - b) $\{ \langle M \rangle : a \in L(M) \}$.
 - c) $\{ \langle M \rangle : L(M) = \{ a \} \}$.
 - d) $\{ \langle M_a, M_b \rangle : M_a \text{ and } M_b \text{ are Turing machines and } \varepsilon \in L(M_a) - L(M_b) \}$.
 - e) $\{ \langle M_a, M_b \rangle : M_a \text{ and } M_b \text{ are Turing machines and } L(M_a) = L(M_b) - \{ \varepsilon \} \}$.
 - f) $\{ \langle M_a, M_b \rangle : M_a \text{ and } M_b \text{ are Turing machines and } L(M_a) \neq L(M_b) \}$.
 - g) $\{ \langle M, w \rangle : M, \text{ when operating on input } w, \text{ never moves to the right on two consecutive moves} \}$.
 - h) $\{ \langle M \rangle : M \text{ is the only Turing machine that accepts } L(M) \}$.
 - i) $\{ \langle M \rangle : L(M) \text{ contains at least two strings} \}$.
 - j) $\{ \langle M \rangle : M \text{ rejects at least two even length strings} \}$.
 - k) $\{ \langle M \rangle : M \text{ halts on all palindromes} \}$.
 - l) $\{ \langle M \rangle : L(M) \text{ is context-free} \}$.
 - m) $\{ \langle M \rangle : L(M) \text{ is not context-free} \}$.
 - n) $\{ \langle M \rangle : A_\#(L(M)) > 0 \}$, where $A_\#(L) = |L \cap a^*|$.
 - o) $\{ \langle M \rangle : |L(M)| > 0 \text{ and is prime} \}$.
 - p) $\{ \langle M \rangle : \text{there exists a string } w \text{ such that } |w| < |\langle M \rangle| \text{ and that } M \text{ accepts } w \}$.
 - q) $\{ \langle M \rangle : M \text{ does not accept any string that ends with } 0 \}$.
 - r) $\{ \langle M \rangle : \text{there are at least two strings } w \text{ and } x \text{ such that } M \text{ halts on both } w \text{ and } x \text{ within some number of steps } s, \text{ and } s < 1000 \text{ and } s \text{ is prime} \}$.
 - s) $\{ \langle M \rangle : \text{there exists an input on which } M \text{ halts in fewer than } |\langle M \rangle| \text{ steps} \}$.
 - t) $\{ \langle M \rangle : L(M) \text{ is infinite} \}$.
 - u) $\{ \langle M \rangle : L(M) \text{ is uncountably infinite} \}$.

- v) $\{ \langle M \rangle : M \text{ accepts the string } \langle M, M \rangle \text{ and does not accept the string } \langle M \rangle \}$.
- w) $\{ \langle M \rangle : M \text{ accepts at least two strings of different lengths} \}$.
- x) $\{ \langle M \rangle : M \text{ accepts exactly two strings and they are of different lengths} \}$.
- y) $\{ \langle M, w \rangle : M \text{ accepts } w \text{ and rejects } w^R \}$.
- z) $\{ \langle M, x, y \rangle : M \text{ accepts } xy \}$.
- aa) $\{ \langle D \rangle : \langle D \rangle \text{ is the string encoding of a deterministic FSM } D \text{ and } L(D) = \emptyset \}$.

2) In \mathfrak{B} 648, we describe a straightforward use of reduction that solves a grid coloring problem by reducing it to a graph problem. Given the grid G shown here:



- a) Show the graph that corresponds to G .
- b) Use the graph algorithm we describe to find a coloring of G .

3) In this problem, we consider the relationship between H and a very simple language $\{a\}$.

- a) Show that $\{a\}$ is *mapping* reducible to H.
- b) Is it possible to reduce H to $\{a\}$? Prove your answer.

4) Show that $H_{ALL} = \{ \langle M \rangle : \text{Turing machine } M \text{ halts on } \Sigma^* \}$ is not in D by reduction from H.

5) Show that each of the following languages is not in D:

- a) A_ϵ .
- b) A_{ANY} .
- c) A_{ALL} .
- d) $\{ \langle M, w \rangle : \text{Turing machine } M \text{ rejects } w \}$.
- e) $\{ \langle M, w \rangle : \text{Turing machine } M \text{ is a deciding Turing machine and } M \text{ rejects } w \}$.

6) Show that $L = \{ \langle M \rangle : \text{Turing machine } M, \text{ on input } \epsilon, \text{ ever writes } 0 \text{ on its tape} \}$ is in D iff H is in D. In other words, show that $L \leq H$ and $H \leq L$.

7) Show that each of the following questions is undecidable by recasting it as a language recognition problem and showing that the corresponding language is not in D:

- a) Given a program P , input x , and a variable n , does P , when running on x , ever assign a value to n ?

Can a compiler check to make sure every variable is initialized before it is used? \mathfrak{C} 669.

- b) Given a program P and code segment S in P , does P reach S on every input (in other words, can we guarantee that S happens)?
- c) Given a program P and a variable x , is x always initialized before it is used?
- d) Given a program P and a file f , does P always close f before it exits?
- e) Given a program P with an array reference of the form $a[i]$, will i , at the time of the reference, always be within the bounds declared for the array?
- f) Given a program P and a database of objects d , does P perform the function f on all elements of d ?

8) In \mathfrak{C} 718, we proved Theorem 41.1, which tells us that the safety of even a very simple security model is undecidable, by reduction from H_ϵ . Show an alternative proof that reduces $A = \{ \langle M, w \rangle : M \text{ is a Turing machine and } w \in L(M) \}$ to the language Safety.

9) Show that each of the following languages is not in SD:

- a) $\neg H_\epsilon$.
- b) EqTMs.

- c) TM_{REG} .
- d) $\{ \langle M \rangle : |L(M)| \text{ is even} \}$.
- e) $\{ \langle M \rangle : \text{Turing machine } M \text{ accepts all even length strings} \}$.
- f) $\{ \langle M \rangle : \text{Turing machine } M \text{ accepts no even length strings} \}$.
- g) $\{ \langle M \rangle : \text{Turing machine } M \text{ does not halt on input } \langle M \rangle \}$.
- h) $\{ \langle M, w \rangle : M \text{ is a deciding Turing machine and } M \text{ rejects } w \}$.
- 10) Do the other half of the proof of Rice's Theorem. In other words, show that the theorem holds if $P(\emptyset) = True$.
- 11) For each of the following languages L , do two things:
- State whether or not Rice's Theorem has anything to tell us about the decidability of L .
 - State whether L is in D, SD/D, or not in SD.
- a) $\{ \langle M \rangle : \text{Turing machine } M \text{ accepts all strings that start with } a \}$.
- b) $\{ \langle M \rangle : \text{Turing machine } M \text{ halts on } \epsilon \text{ in no more than 1000 steps} \}$.
- c) $\neg L_1$, where $L_1 = \{ \langle M \rangle : \text{Turing machine } M \text{ halts on all strings in no more than 1000 steps} \}$.
- d) $\{ \langle M, w \rangle : \text{Turing machine } M \text{ rejects } w \}$.
- 12) Use Rice's Theorem to prove that each of the following languages L is not in D:
- $\{ \langle M \rangle : \text{Turing machine } M \text{ accepts at least two odd length strings} \}$.
 - $\{ \langle M \rangle : M \text{ is a Turing machine and } |L(M)| = 12 \}$.
- 13) Prove that there exists no mapping reduction from H to the language L_2 that we defined in Theorem 21.9.
- 14) Let $\Sigma = \{1\}$. Show that there exists at least one undecidable language with alphabet Σ .
- 15) Give an example of a language L such that neither L nor $\neg L$ is decidable.
- 16) Let $repl$ be a function that maps from one language to another. It is defined as follows:
- $$repl(L) = \{ w : \exists x \in L \text{ and } w = xx \}.$$
- Are the context free languages closed under $repl$? Prove your answer.
 - Are the decidable languages closed under $repl$? Prove your answer.
- 17) For any nonempty alphabet Σ , let L be any decidable language other than \emptyset or Σ^* . Prove that $L \leq_M \neg L$.
- 18) We will say that L_1 is **doubly reducible** to L_2 , which we will write as $L_1 \leq_D L_2$, iff there exist two computable functions f_1 and f_2 such that:
- $$\forall x \in \Sigma^* ((x \in L_1) \text{ iff } (f_1(x) \in L_2) \text{ and } (f_2(x) \notin L_2)).$$
- Prove or disprove each of the following claims:
- If $L_1 \leq L_2$ and $L_2 \neq \Sigma^*$, then $L_1 \leq_D L_2$.
 - If $L_1 \leq_D L_2$ and $L_2 \in D$, then $L_1 \in D$.
 - For every language L_2 , there exists a language L_1 such that $\neg(L_1 \leq_D L_2)$.
- 19) Let L_1 and L_2 be any two SD/D languages such that $L_1 \subset L_2$. Is it possible that L_1 is reducible to L_2 ? Prove your answer.
- 20) If L_1 and L_2 are decidable languages and $L_1 \subseteq L \subseteq L_2$, must L be decidable? Prove your answer.

- 21) Goldbach's conjecture states that every even integer greater than 2 can be written as the sum of two primes. (Consider 1 to be prime.) Suppose that $A = \{ \langle M, w \rangle : M \text{ is a Turing machine and } w \in L(M) \}$ were decidable by some Turing machine *Oracle*. Define the following function:

$$G() = \begin{array}{l} \textit{True} \text{ if Goldbach's conjecture is true,} \\ \textit{False} \text{ otherwise.} \end{array}$$

Use *Oracle* to describe a Turing machine that computes G . You may assume the existence of a Turing machine P that decides whether a number is prime.

- 22) A language L is **D-complete** iff (1) L is in D , and (2) for every language L' in D , $L' \leq_M L$. Consider the following claim: If $L \in D$ and $L \neq \Sigma^*$ and $L \neq \emptyset$, then L is D-complete. Prove or disprove this claim.

22 Decidability of Languages That Do Not (Obviously) Ask Questions about Turing Machines ♦

If the only problems that were undecidable were questions involving the behavior of Turing Machines (and thus programs written in any reasonable formalism), we would still care. After all, being able to prove properties of the programs that we write is of critical importance when bugs could mean the loss of millions of dollars or hundreds of lives. But Turing Machines do not own the market in undecidable problems. In this chapter, we will look at some examples of undecidable problems that do not (at least directly) ask questions about Turing Machines.

Although the problems we will consider here do not appear to involve Turing Machines, each of the undecidability proofs that we will describe is based, either directly or indirectly, on a reduction from a language (such as H , A , or $\neg H_e$) whose definition does refer to the behavior of Turing machines. Many of these proofs are based on variants of a single idea, namely the fact that it is possible to encode a Turing machine configuration as a string. For example, the string $abq100cd$ can encode the configuration of a Turing machine that is in state 4, with $abcd$ on the tape and the read/write head positioned on the c . Then we can encode a computation of a Turing machine as a sequence of configurations, separated by delimiters. For example, we might have:

$\#q0\sqsupset abcd\#q1abcd\#aq1bcd\#$

Or we can encode a computation as a table, where each row corresponds to a configuration. So, for example, we might have:

$\#q0\sqsupset abcd\#$
 $\#\sqsupset q1abcd\#$
 $\#\sqsupset aq1bcd\#$

To show that a new language is not decidable, we will then define a reduction that maps from one of these representations of a Turing machine's computations to some essential structure of the new problem. We will design the reduction so that the new problem's structure possesses some key property iff the Turing machine's computation enters a halting state (or an accepting state) or fails to enter a halting state, or whatever it is we need to check. So, if there is a procedure to decide whether an instance of the new problem possesses the key property, then there is also a way to check whether a Turing machine halts (or accepts or whatever). So, for example, suppose that we are using the table representation and that M , the Turing machine whose computation we have described, does not halt. Then the table will have an infinite number of rows. In the proof we'll sketch in Section 22.3, the cells of the table will correspond to tiles that must be arranged according to a small set of rules. So, if we could tell whether an infinite arrangement of tiles exists, we could tell whether the table is infinite (and thus whether M fails to halt).

22.1 Diophantine Equations and Hilbert's 10th Problem

In 1900, the German mathematician David Hilbert presented a list of 23 problems that he argued should be the focus of mathematical research as the new century began. The 10th of his problems \square concerned systems of Diophantine equations (polynomials in any number of variables, all with integer coefficients), such as:


$$4x^3 + 7xy + 2z^2 - 23x^4z = 0.$$

A Diophantine problem is, "Given a system of Diophantine equations, does it have an integer solution?" Hilbert asked whether there exists a decision procedure for Diophantine problems. Diophantine problems are important in applications in which the variables correspond to quantities of indivisible objects in the world. For example, suppose x is the number of shares of stock A to be bought, y is the number of shares of stock B, and z is the number of shares of stock C. Since it is not generally possible to buy fractions of shares of stock, any useful solution to an equation involving x , y , and z would necessarily assign integer values to each variable.

We can recast the Diophantine problem as the language $TENTH = \{\langle w \rangle : w \text{ is a system of Diophantine equations that has an integer solution}\}$. In 1970, Yuri Matiyasevich proved a general result from which it follows that the answer to Hilbert's question is no: $TENTH$ is not in D . Using the Fibonacci sequence (defined in Example 24.4), Matiyasevich

proved that every semidecidable set S is Diophantine, by which we mean that there is a reduction from S to the problem of deciding whether some system of Diophantine equations has an integer solution. So, if the Diophantine problem were decidable, every semidecidable set would be decidable. But we know that there are semidecidable sets (e.g., the language H) that are not decidable. So the Diophantine problem is not decidable either.

As an aside, however, we should point out that when all of the terms are of degree 1, Diophantine problems are decidable, which is good news for the writers of puzzle problems of the sort:

A farmer buys 100 animals for \$100.00. The animals include at least one cow, one pig, and one chicken, but no other kind. If a cow costs \$10.00, a pig costs \$3.00, and a chicken costs \$0.50, how many of each did he buy? 

It is also true that Diophantine problems that involve just a single variable are decidable. And quadratic Diophantine problems of two variables are decidable. These are problems of the form $ax^2 + by = c$, where a , b , and c are positive integers and we ask whether there exist integer values of x and y that satisfy the equation.

We will return the question of the solvability of Diophantine problems in Part V. There we will see that:


- Diophantine problems of degree 1 (like the cows, pigs, and chickens problem) and Diophantine problems of a single variable of the form $ax^k = c$ are not only solvable, they are efficiently solvable (i.e., there exists a polynomial-time algorithm to solve them).
- Quadratic Diophantine problems are solvable, but there appears to exist no polynomial-time algorithm for doing so. The quadratic Diophantine problem belongs to the complexity class NP-complete.
- The general Diophantine problem is undecidable, so not even an inefficient algorithm for it exists.

22.2 Post Correspondence Problem

Consider two lists of strings over some alphabet Σ . The lists must be finite and of equal length. We can call the two lists X and Y . So we have:

$$X = x_1, x_2, x_3, \dots, x_n.$$

$$Y = y_1, y_2, y_3, \dots, y_n.$$

Now we ask a question about the lists: Does there exist some finite sequence of integers that can be viewed as indices of X and Y such that, when elements of X are selected as specified and concatenated together, we get the same string that we get when elements of Y are also concatenated together as specified? For example, if we assert that 1, 3, 4 is such a sequence, we're asserting that $x_1x_3x_4 = y_1y_3y_4$. Any problem of this form is an instance of the **Post Correspondence Problem** , first formulated in the 1940's by Emil Post.

Example 22.1 A PCP Instance with a Simple Solution

Let PCP_1 be:

	X	Y
1	b	bab
2	abb	b
3	aba	a
4	bbaaa	babaaa

PCP_1 has a very simple solution: 1, 2, 3, 1, which is a solution because (ignoring spaces, which are shown here just to make it clear what is happening):

$$b \text{ abb } aba \text{ b} = bab \text{ b } a \text{ bab}$$

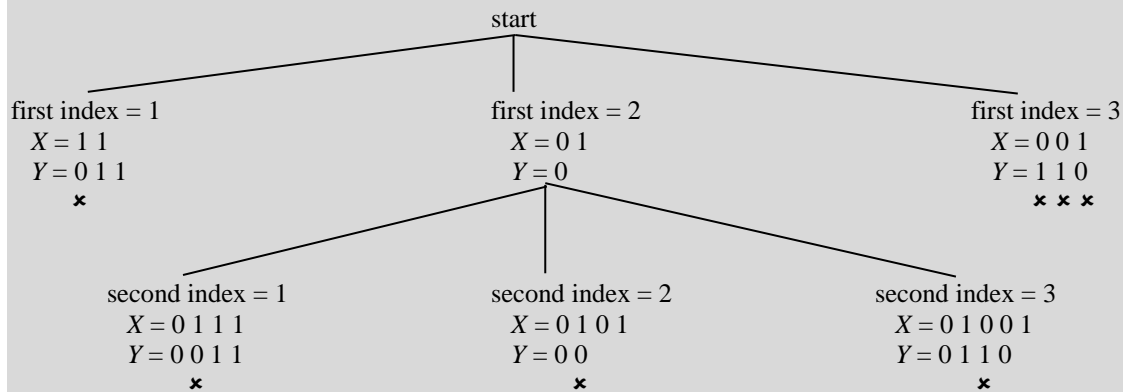
It also has an infinite number of other solutions, including 1, 2, 3, 1, 1, 2, 3, 1 and 1, 2, 3, 1, 1, 2, 3, 1, 1, 2, 3, 1.

Example 22.2 A PCP Instance with No Solution

Let PCP_2 be:

	X	Y
1	11	011
2	01	0
3	001	110

PCP_2 has no solution. It is straightforward to show this by trying candidate solutions. Mismatched symbols that cause the current path to die are marked with an \times .



All paths have failed to find a solution.

Example 22.3 A PCP Instance with No Simple Solutions

Let PCP_3 be:

	X	Y
1	1101	1
2	0110	11
3	1	110

PCP_3 has solutions (in fact, an infinite number of them), but the shortest one has length 252

We can formulate the Post Correspondence Problem as a language decision problem. To do that, we need to define a way to encode each instance of the Post Correspondence Problem as a string. Let Σ be any nonempty alphabet. Then an instance of the Post Correspondence Problem is a string $\langle P \rangle$ of the form:

$$\langle P \rangle = (x_1, x_2, x_3, \dots, x_n)(y_1, y_2, y_3, \dots, y_n), \text{ where } \forall j (x_j \in \Sigma^+ \text{ and } y_j \in \Sigma^+).$$

For example, $\langle PCP_1 \rangle = (b, abb, aba, bbaaa)(bab, b, a, babaaa)$.

We'll say that a PCP instance has *size* n whenever the number of strings in its X list is n . (In this case, the number of strings in its Y list is also n .) A solution to a PCP instance of size n is a finite sequence i_1, i_2, \dots, i_k of integers such that:

$$\forall 1 \leq j \leq k (1 \leq i_j \leq n \text{ and } x_{i_1}x_{i_2}\dots x_{i_k} = y_{i_1}y_{i_2}\dots y_{i_k}).$$

To define a concrete PCP language, we need to fix an alphabet. We'll let Σ be $\{0, 1\}$ and simply encode any other alphabet in binary. Now the problem of determining whether a particular instance P of the Post Correspondence Problem has a solution can be recast as the problem of deciding the language:

- $PCP = \{ \langle P \rangle : P \text{ is an instance of the Post correspondence problem and } P \text{ has a solution} \}$.

Theorem 22.1 The Undecidability of PCP

Theorem: The language $PCP = \{ \langle P \rangle : P \text{ is an instance of the Post correspondence problem and } P \text{ has a solution} \}$ is in SD/D.

Proof: We will first show that PCP is in SD by building a Turing machine $M_{PCP}(\langle P \rangle)$ that semidecides it. The idea is that M_{PCP} will simply try all possible solutions of length 1, then all possible solutions of length 2, and so forth. If there is any finite sequence of indices that is a solution, M_{PCP} will find it. To describe more clearly how M_{PCP} works, we first observe that any solution to a PCP problem $P = (x_1, x_2, x_3, \dots, x_n)(y_1, y_2, y_3, \dots, y_n)$ is a finite sequence of integers between 1 and n . We can build a Turing machine $M\#$ that lexicographically enumerates all such sequences. Now we define:

- $M_{PCP}(\langle P \rangle) =$
1. Invoke $M\#$.
 2. As each string is enumerated, see if it is a solution to P . If so, halt and accept.

Next we must prove that PCP is not in D. There are two approaches we could take to doing this proof. One is to use reduction from H. The idea is that, to decide whether $\langle M, w \rangle$ is in H, we create a PCP instance that simulates the computation history of M running on w . We do the construction in such a way that there exists a finite sequence that solves the PCP problem iff the computation halts. So, if we could decide PCP, we could decide H. An alternative is to make use of the grammar formalism that we will define in the next chapter. We'll show there that unrestricted grammars generate all and only the SD languages. So the problem of deciding whether a grammar G generates a string w is equivalent to deciding whether a Turing machine M accepts w and is thus undecidable. Given that result, we can prove that PCP is not in D by a construction that maps a grammar to a PCP instance in such a way that there exists a finite sequence that solves the PCP problem iff there is a finite derivation of w using the rules of G . This second approach is somewhat easier to explain, so it is the one we'll use. We give the proof in \mathfrak{B} 649. ■

It turns out that some special cases of PCP are decidable. For example, if we restrict our attention to problems of size two, then PCP is decidable. A bounded version of PCP is also decidable. Define the language:

- $BOUNDED-PCP = \{ \langle P, k \rangle : P \text{ is an instance of the Post Correspondence problem that has a solution of length less than or equal to } k \}$.

While BOUNDED-PCP is decidable (by a straightforward algorithm that simply tries all candidate solutions of length up to k), it appears not to be efficiently decidable. It is a member of the complexity class NP-complete, which we will define in Section 28.2.

The fact that PCP is not decidable in general is significant. As we will see in Section 22.5.3, reduction from PCP is a convenient way to show the undecidability of other kinds of problems, including some that involve context-free languages.

22.3 Tiling Problems

Consider a class of tiles called Wang tiles or Wang dominos \square . A Wang tile is a square that has been divided into four regions by drawing two diagonal lines, as shown in Figure 22.1. Each region is colored with one of a fixed set of colors.



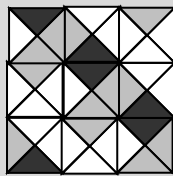
Figure 22.1 A tiling problem

Now suppose that you are given a finite set of such tile designs, all of the same size, for example the set of three designs shown here. Further suppose that you have an infinite supply of each type of tile. Then we may ask whether or not it possible to tile an arbitrary surface in the plane with the available designs while adhering to the following rules:

1. Each tile must be placed so that it is touching its neighbors on all four sides (if such neighbors exist). In other words, no gaps or overlaps are allowed.
2. When two tiles are placed so that they adjoin each other, the adjoining regions of the two tiles must be the same color.
3. No rotations or flips are allowed.

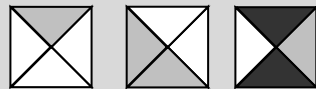
Example 22.4 A Set of Tiles that Can Tile the Plane

The set of tiles shown above can be used to tile any surface in the plane. Here is a small piece of the pattern that can be built and then repeated as necessary since its right and left sides match, as do its top and bottom:



Example 22.5 A Set of Tiles that Cannot Tile the Plane

Now consider a new set of tiles:



Only a small number of small regions can be tiled with this set. To see this, start with tile 1, add a tile below it and then try to extend the pattern sideways. Then start with tile 2 and show the same thing. Then observe that tile 3, the only one remaining, cannot be placed next to itself.

We can formulate the tiling problem, as we have just described it, as a language to be decided. To do that, we need to define a way to encode each instance of the tiling problem (i.e., a set of tile designs) as a string. We will represent each design as an ordered 4-tuple of values drawn from the set $\{G, W, B\}$. To describe a design, start in the top region and then go around the tile clockwise. So, for example, the first tile set we considered could be represented as:

$$(G \ W \ W \ W) (W \ W \ B \ G) (B \ G \ G \ W).$$


Now we can define: TILES =


- TILES = $\{ \langle T \rangle : \text{every finite surface on the plane can be tiled, according to the rules, with the tile set } T \}$.

The string $(G \ W \ W \ W) (W \ W \ B \ G) (B \ G \ G \ W)$, which corresponds to the tile set of Example 22.4, is in TILES. The string $(G \ W \ W \ W) (W \ W \ G \ G) (B \ G \ B \ W)$, which corresponds to the tile set of Example 22.5, is not in TILES.

Is TILES in D? In other words, does there exist a decision procedure that determines, for a given set of tiles, whether or not it can be used to tile an arbitrary surface in the plane? Consider the following conjecture, called Wang's conjecture: If a given set of tiles can be used to tile an arbitrary surface, then it can always do so periodically. In other words, there must exist a finite area that can be tiled and then repeated infinitely often to cover any desired surface. For example, the tile set of Example 22.4 covers the plane periodically using the 3×3 grid shown above. If Wang's conjecture were true, then the tiling question would be decidable by considering successively larger square

grids in search of one that can serve as the basis for a periodic tiling. If such a grid exists, it will be found. If no such grid exists, then it is possible to prove (using a result known as the König infinity lemma) that there must exist a finite square grid that cannot be tiled at all. So this procedure must eventually halt, either by finding a grid that can be used for a periodic tiling or by finding a grid that cannot be tiled at all (and thus discovering that no periodic tiling exists).

It is possible to make many kinds of changes to the kinds of tiles that are allowed without altering the undecidability properties of the tiling problem as we have presented it for Wang tiles. Tiling problems, in this broader sense, have widespread applications in the physical world . For example, the growth of crystals can often be described as a tiling.

As it turns out, Wang’s conjecture is false. There exist tile sets  that can tile an arbitrary area aperiodically (i.e., without any repeating pattern) but for which no periodic tiling exists. Of course, that does not mean that TILES must not be in D. There might exist some other way to decide it. But there does not. TILES is not in D. In fact, it is not even in SD, although \neg TILES is.

Theorem 22.2 The Undecidability of TILES

Theorem: The language $TILES = \{ \langle T \rangle : \text{every finite surface on the plane can be tiled, according to the rules, with the tile set } T \}$ is not in D. It is also not in SD. But \neg TILES is in SD.

Proof: We first prove that \neg TILES is in SD. Consider a search space defined as follows: The start state contains no tiles. From any state s , construct the set of successor states, each of which is built by adding one tile, according to the rules, to the configuration in s . We can build a Turing machine M that semidecides \neg TILES by systematically exploring this space. If and only if it ever happens that all the branches reach a dead end in which there is no legal move, then there is no tiling and M accepts.

If TILES were also in SD, then, by Theorem 20.6, it would be in D. But it is not. The proof that it is not is by reduction from $\neg H_\epsilon$. The idea behind the reduction is to describe a way to map an arbitrary Turing machine M into a set of tiles T in such a way that T is in TILES iff M does not halt on ϵ . The reduction uses a row of tiles to correspond to a configuration of M . It begins by creating a row that corresponds to M ’s initial configuration when started on a blank tape. Then the next row will correspond to M ’s next configuration, and so forth. There is always a next configuration of M and thus a next row in the tiling iff M does not halt. T is in TILES iff there is always a next row (i.e., T can tile an arbitrarily large area). So if it were possible to semidecide whether T is in TILES it would be possible to semidecide whether M fails to halt on ϵ . But we know (from Theorem 21.16) that $\neg H_\epsilon$ is not in SD. So neither is TILES. ■

The language TILES corresponds to an unbounded tiling problem. We can also formulate a bounded version: “Given a particular stack of n^2 tiles (for some value of n), is it possible to tile an $n \times n$ surface in the plane?” This problem is clearly decidable by the straightforward algorithm that simply tries all ways of placing the n^2 tiles on the n^2 cells of an $n \times n$ grid. But there is still bad news. The theory of time complexity that we will describe in Chapter 28 provides the basis for formalizing the following claim: The bounded tiling problem is apparently intractable. The time required to solve it by the best known algorithm grows exponentially with n . We will return to this discussion in Exercise 28.20).

22.4 Logical Theories

Even before anyone had seen a computer, mathematicians were interested in the question: “Does there exist an algorithm to determine, given a statement in a logical language, whether or not it is a theorem?” In other words, can it be proved from the available axioms plus the rules of inference. In the case of formulas in first order logic, this problem even had a specific name, the Entscheidungsproblem. With the advent of the computer, the Entscheidungsproblem acquired more than mathematical interest. If such an algorithm existed, it could play a key role in programs that:

- Decide whether other programs are correct.
- Determine that a plan for controlling a manufacturing robot is correct.
- Accept or reject interpretations for English sentences, based on whether or not they make sense.

22.4.1 Boolean Theories

If we consider only Boolean logic formulas, such as $(P \wedge (Q \vee \neg R) \rightarrow S)$, then there exist procedures to decide all of the following questions:

- Given a well-formed formula (wff) w , is w valid (i.e., is it true for all assignments of truth values to its variables)?
- Given a wff w , is w satisfiable (i.e., is there some assignment of truth values to its variables such that w is true)?
- Given a wff w and a set of axioms A , is w a theorem (i.e., can it be proved from A)?

Alternatively, all of the following languages are in D :

- $\text{VALID} = \{ \langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ is valid} \}$.
- $\text{SAT} = \{ \langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ is satisfiable} \}$.
- $\text{PROVABLE} = \{ \langle A, w \rangle : w \text{ is a wff in Boolean logic, } A \text{ is a set of axioms in Boolean logic and } w \text{ is provable from } A \}$

Suppose that the specification for a hardware device or a software system can be described in terms of a finite number of states. Then it can be written as a Boolean formula. Then one way to verify the correctness of a particular implementation is to see whether it satisfies the specification. The fact that SAT is decidable makes this approach, called model checking, possible. \mathfrak{B} 679.

There is a straightforward procedure for answering all of these questions since each wff contains only a finite number of variables and each variable can take on one of two possible values (*True* or *False*). So it suffices to try all the possibilities. A wff is valid iff it is true in all assignments of truth values to its variables. It is satisfiable iff it is true in at least one such assignment. A wff w is provable from A iff $(A \rightarrow w)$ is valid.

Unfortunately, if w contains n variables, then there are 2^n ways of trying all ways of assigning values to those variables. So any algorithm that does that takes time that grows exponentially in the size of w . The best known algorithms for answering any of these questions about an arbitrary wff do take exponential time in the worst case. We'll return to this issue when we consider complexity in Part V. However, we should note that there are techniques that can perform better than exponentially in many cases. One approach represents formulas as ordered binary decision diagrams (OBDDs). \mathfrak{B} 612.

22.4.2 First-Order Logical Theories

If we consider first-order logic (FOL) sentences, such as $\forall x (\exists y (P(x, y) \wedge Q(y, x) \rightarrow T(x)))$, then none of the questions we asked about Boolean logic (validity, satisfiability, and theoremhood) is decidable.

We'll focus here on the question of deciding, given a sentence w and a decidable set of axioms A , whether w can be proved from A . To do this, we'll define the language:

- $\text{FOL}_{\text{theorem}} = \{ \langle A, w \rangle : A \text{ is a decidable set of axioms in first-order logic, } w \text{ is a sentence in first-order logic, and } w \text{ is entailed by } A \}$.

Note that we do not require that the set of axioms be finite, but we do require that it be decidable. For example *Peano arithmetic* is a first-order logical theory that describes the natural numbers, along with the functions *plus* and *times* applied to them. Peano arithmetic exploits an infinite but decidable set of axioms.

Theorem 22.3 First-Order Logic is Semidecidable

Theorem: $\text{FOL}_{\text{theorem}} = \{ \langle A, w \rangle : A \text{ is a decidable set of axioms in first-order logic, } w \text{ is a sentence in first-order logic, and } w \text{ is entailed by } A \}$ is in SD.

Proof: The algorithm *proveFOL* semidecides $\text{FOL}_{\text{theorem}}$:

proveFOL(A : decidable set of axioms, w : sentence) =

1. Using some complete set of inference rules for first-order logic, begin with the sentences in A and lexicographically enumerate the sound proofs. If A is infinite, then it will be necessary to embed in that process a subroutine that lexicographically enumerates the sentences in the language of the theory of A and checks each to determine whether or not it is an axiom.
2. Check each proof as it is created. If it succeeds in proving w , halt and accept.

By Gödel's Completeness Theorem we know that there does exist a set of inference rules for first order logic that is complete in the sense that they are able to derive, from a set of axioms A , all sentences that are entailed by A . So step 1 of *proveFOL* can be correctly implemented. ■

There exist techniques for implementing *proveFOL* in a way that is computationally efficient enough for many practical applications. We describe one of them, resolution, in § 620.

Unfortunately, *proveFOL* is not a decision procedure since it may not halt. Also, unfortunately, it is not possible to do better, as we now show.

Logical reasoning provides a basis for many artificial intelligence systems. Does the fact that first-order logic is undecidable mean that artificial intelligence is impossible? § 769.

Theorem 22.4 First-Order Logic is not Decidable

Theorem: $\text{FOL}_{\text{theorem}} = \{ \langle A, w \rangle : A \text{ is a decidable set of axioms in first-order logic, } w \text{ is a sentence in first-order logic, and } w \text{ is entailed by } A \}$ is not in D.

Proof: Let T be any first-order theory with A (a decidable set of axioms) and M (an interpretation, i.e., a domain and an assignment of meanings to the constant, predicate, and function symbols of A). If T is not consistent, then all sentences are theorems in T . So the simple procedure that always returns *True* decides whether any sentence is a theorem in T . We now consider the case in which T is consistent.

If T is complete then, for any sentence w , either w or $\neg w$ is a theorem. So the set of theorems is decidable because it can be decided by the following algorithm:

decidecompletetheory(A : set of axioms, w : sentence) =

1. In parallel, use *proveFOL*, as defined above, to attempt to prove w and $\neg w$.
2. One of the proof attempts will eventually succeed. If the attempt to prove w succeeded, then return *True*. If the attempt to prove $\neg w$ succeeded, then return *False*.

A slightly different way to say this is that if the set of theorems is in SD and the set of nontheorems is also in SD, then by Theorem 20.6, both sets are also in D.

But we must also consider the case in which T is not complete. Now it is possible that neither w nor $\neg w$ is a theorem. Does there exist a decision procedure to determine whether w is a theorem? The answer is no, which we will show by exhibiting one particular theory for which no decision procedure exists. We use the theory of Peano arithmetic. Gödel proved (in a result that has come to be known as Gödel's Incompleteness Theorem) that the theory of Peano arithmetic cannot be both consistent and complete.

Following Turing’s argument, we show that $H_\varepsilon \leq_M \text{FOL}_{\text{theorem}}$ and so $\text{FOL}_{\text{theorem}}$ is not in D. Let R be a mapping reduction from $H_\varepsilon = \{\langle M \rangle : \text{Turing machine } M \text{ halts on } \varepsilon\}$ to $\text{FOL}_{\text{theorem}}$ defined as follows:

$R(\langle M \rangle) =$

1. From $\langle M \rangle$, construct a sentence F in the language of Peano arithmetic, such that F is a theorem, provable given the axioms of Peano arithmetic, iff M halts on ε .
2. Let P be the axioms of Peano arithmetic. Return $\langle P, F \rangle$.

If *Oracle* exists and decides $\text{FOL}_{\text{theorem}}$, then $C = \text{Oracle}(R(\langle M, w \rangle))$ decides H_ε :

- There exists an algorithm to implement R . It is based on the techniques described by Turing (although he actually proved first that, because H_ε is undecidable, it is also undecidable whether a Turing machine ever prints 0. He then showed how to create a logical expression that is a theorem of Peano arithmetic iff a Turing machine ever prints 0). We omit the details \square .
- C is correct:
 - If $\langle M, w \rangle \in H_\varepsilon$: M halts on ε . F is a theorem of Peano arithmetic. $\text{Oracle}(\langle P, F \rangle)$ accepts.
 - If $\langle M, w \rangle \notin H_\varepsilon$: M does not halt on ε . F is not a theorem of Peano arithmetic. $\text{Oracle}(\langle P, F \rangle)$ rejects.

But no machine to decide H_ε can exist, so neither does *Oracle*. ■

Is it decidable, given a system of laws, whether some consequence follows from those laws? \mathbb{C} 770.

Keep in mind that the fact that $\text{FOL}_{\text{theorem}}$ is undecidable means only that there is no algorithm to *decide* whether an arbitrary sentence is a theorem in an arbitrary theory. $\text{FOL}_{\text{theorem}}$ is semidecidable and the algorithm, *proveFOL*, that we described in the proof of Theorem 22.3, provides a way to *discover* a proof if one exists. Although efficiency issues arise, we shouldn’t write off first-order systems as practical tools, despite the negative results that we have just shown.

Also note that, just as the unsolvability of the halting problem doesn’t say that there are not some cases in which we can show that a program halts or other cases in which we can show that it doesn’t, the fact that $\text{FOL}_{\text{theorem}}$ is undecidable doesn’t prove that there are not some theories for which it is possible to decide theoremhood.

For example, consider *Presburger arithmetic*, a theory of the natural numbers and the single function *plus* \square . The following is a theorem of Presburger arithmetic (where “number” means natural number):

- The sum of two odd numbers is even: $\forall x (\forall y ((\exists u (x = u + u + 1) \wedge \exists v (y = v + v + 1)) \rightarrow \exists z (x + y = z + z)))$.

Presburger arithmetic is decidable (although unfortunately, as we will see in Section 28.9.3, no *efficient* procedure for deciding it exists).

Because Presburger arithmetic is decidable, it has been used as a basis for verification systems that prove the correctness of programs. We’ll say more about program verification in \mathbb{C} 678.

22.5 Undecidable Problems about Context-Free Languages

Recall from Chapter 9 that we were able to find a decision procedure for all of the questions that we asked about regular languages. We have just seen, at the other extreme, that almost all the questions we ask about Turing machines and the languages they define are undecidable. What about context-free languages? In Chapter 14, we described two decision procedures for them:

1. Given a CFL L and a string s , is $s \in L$?
2. Given a CFL L , is $L = \emptyset$?

What about other questions we might like to ask, including:

3. Given a CFL L , is $L = \Sigma^*$?
4. Given two CFLs L_1 and L_2 , is $L_1 = L_2$?
5. Given two CFLs L_1 and L_2 , is $L_1 \subseteq L_2$?
6. Given a CFL L , is $\neg L$ context-free?
7. Given a CFL L , is L regular?
8. Given two CFLs L_1 and L_2 , is $L_1 \cap L_2 = \emptyset$?
9. Given a CFL L , is L inherently ambiguous?

Since we have proven that there exists a grammar that generates L iff there exists a PDA that accepts it, these questions will have the same answers whether we ask them about grammars or about PDAs. In addition, there are questions that we might like to ask specifically about PDAs, including:

10. Given two PDAs M_1 and M_2 , is M_2 a minimization of M_1 ? Define M_2 to be a *minimization* of M_1 iff $L(M_1) = L(M_2)$ and there exists no other PDA M' such that $L(M_2) = L(M')$ and M' has fewer states than M_2 has.

And there are other questions specifically about grammars, including:

11. Given a CFG G , is G ambiguous?

Questions 3-11 are all undecidable. Alternatively, if these problems are stated as languages, the languages are not in D. Keep in mind however that just as there are programs that can be shown to halt (or not to halt), there are context-free languages about which various properties can be proven. For example, although question 11 is undecidable (for an arbitrary CFG), some grammars can easily be shown to be ambiguous by finding a single string for which two parses exist. And other grammars can be shown to be unambiguous, for example by showing that they are LL(1), as described in Section 15.2.3.

There are two strategies that we can use to show that these problems are in general undecidable. The first is to exploit the idea of a computation history to enable us to reduce H to one of these problems. The second is to show that a problem is not in D by reduction from the Post Correspondence Problem. We will use both, starting with the computation history approach.

22.5.1 Reduction via Computation History

We will first show that question 3 is undecidable by reducing H to the language $\text{CFG}_{\text{ALL}} = \{ \langle G \rangle : G \text{ is a CFG and } L(G) = \Sigma^* \}$. To do this reduction, we will have to introduce a new technique in which we create strings that correspond to the computation history of some Turing machine M . But, once we have shown that CFG_{ALL} is not in D, the proofs of claims 4, 5, and 10 are quite straightforward. They use reduction from CFG_{ALL} .

Recall from Section 17.1 that a *configuration* of a Turing machine M is a 4 tuple:

(M 's current state,
the nonblank portion of the tape before the read/write head,
the character under the read/write head,
the nonblank portion of the tape after the read/write head)

A *computation* of M is a sequence of configurations C_0, C_1, \dots, C_n for some $n \geq 0$ such that C_0 is the initial configuration of M , C_n is a halting configuration of M , and

$$C_0 \mid\text{-}M \ C_1 \mid\text{-}M \ C_2 \mid\text{-}M \ \dots \mid\text{-}M \ C_n.$$

Notice that, under this definition, a computation is a finite sequence of configurations, the last of which must be a halting configuration. So, if M does not halt when started in configuration C_0 , there exists no computation that starts in C_0 . That doesn't mean that M can't compute from C_0 . It just means that there is no finite sequence that records what it does and it is that sequence that we are calling a computation.

A **computation history** of M is a string that encodes a computation. We will write each configuration in the history as a 4-tuple, as described above. Then we will encode the entire history by concatenating the configurations together. So, assuming that s is M 's start state and h is a halting state, here's an example of a string that could represent a computation history of M :

$$(s, \varepsilon, \sqcup, abba)(q_1, \varepsilon, a, bba)(q_2, a, b, ba)(q_2, ab, b, a)(q_2, abb, a, \sqcup)(h, abba, \sqcup, \sqcup).$$

Theorem 22.5 CFG_{ALL} is Undecidable

Theorem: The language $\text{CFG}_{\text{ALL}} = \{ \langle G \rangle : G \text{ is a CFG and } L(G) = \Sigma^* \}$ is not in D.

Proof: We show that CFG_{ALL} is not in D by reduction from $H = \{ \langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w \}$. The reduction we will use exploits two functions, R and \neg . R will map instances of H to instances of CFG_{ALL} . As in the proof of Theorem 21.9, \neg will simply invert *Oracle*'s response (turning an accept into a reject and vice versa).

The idea behind R is that it will build a grammar G that generates the language $L\#$ composed of all strings in Σ^* *except* any that represent a computation history of M on w . If M does not halt on w , there are no computation histories of M on w (since a computation history must be of finite length and end in a halting state) so G generates Σ^* and *Oracle* will accept. If, on the other hand, there exists a computation history of M on w , then there will be a string that G will not generate and *Oracle* will reject. So *Oracle* makes the correct distinction but accepts when we need it to reject and vice versa. But since *Oracle* is a deciding machine, \neg can invert its response.

It turns out to be easier for R to build a PDA to accept $L\#$ than it is to build a grammar to generate it. But we have an algorithm to build, from any PDA, the corresponding grammar. So R will first build a PDA P , then convert P to a grammar.

In order for a string s to be a computation history of M on w , it must possess four properties:

1. It must be a syntactically valid computation history.
2. C_0 must correspond to M being in its start state, with w on the tape, and with the read/write head positioned just to the left of w .
3. The last configuration must be a halting configuration.
4. Each configuration after C_0 must be derivable from the previous one according to the rules in M 's transition relation δ_M .

We want P to accept any string that is *not* a computation history of M on w . So if P finds even one of these conditions violated it will accept. P will nondeterministically choose which of the four conditions to check. It can then check the one it picked as follows:

1. We can write a regular expression to define the syntax of the language of computation histories. So P can easily check for property 1 and accept if the string is ill-formed.
2. R builds P from a particular $\langle M, w \rangle$ pair, so it can hardwire into P what the initial configuration would have to be if s is to be a computation history of M on w .
3. Again, R can hardwire into P what a halting configuration of M is, namely one in which M is in some state in H_M .
4. This is the only hard one. To show that a string s is not composed of configurations that are derivable from each other, it suffices to find even one adjacent pair where the second configuration cannot be derived from the first. So P can nondeterministically pick one configuration and then check to see whether the one that comes after it is not correct, according to the rules of δ_M .

But how exactly, can we implement the test for property 4? Suppose that we have an adjacent pair of configurations. If they are part of a computation history of M , then they must be identical except:

- The state of the second must have changed as specified in δ_M .
- Right around the read/write head, the change specified by δ_M must have occurred on the tape.

So, for example, it is possible, given an appropriate δ_M , that the following string could be part of a computation history:
 $(q_1, aaaa, b, aaaa)(q_2, aaa, a, baaaa)$.


Here M moved the read/write head one square to the left. But it is not possible for the following string to be part of any computation history:

$(q_1, aaaa, b, aaaa)(q_2, bbbb, a, bbbb)$.

M cannot change any squares other than the one directly under its read/write head.

So P must read the first configuration, remember it, and then compare it to the second. Since a configuration can be of arbitrary length and P is a PDA, the only way P can remember a configuration is on the stack. But then it has a problem. When it tries to pop off the symbols from the first configuration to compare it to the second, they will be backwards.

To solve this problem, we will change slightly our statement of the language that P will accept. Now it will be $B\#$, the *boustrophedon* version of $L\#$. In $B\#$ every odd numbered configuration will be written backwards. The word “boustrophedon” aptly describes $B\#$. It is derived from a Greek word that means turning as oxen do in plowing. It is used to describe a writing scheme in which alternate lines are written left to right and then right to left (so that the scribe wastes no effort moving his hand across the page without writing). With this change, P can compare two adjacent configurations and determine whether one could have been derived from the other via δ .

Boustrophedon writing  has been used in ancient Greek texts and for inscriptions on statues on Easter Island. Much more recently, dot matrix printers used back and forth writing, but they adjusted their fonts so that it was not the case that every other line appeared backwards.

We are now ready to state:

$R(\langle M, w \rangle) =$

1. Construct the description of a PDA P that accepts all strings in $B\#$.
2. From P , construct a grammar G that generates $L(P)$.
3. Return $\langle G \rangle$.

$\{R, \neg\}$ is a reduction from H to CFG_{ALL} . If *Oracle* exists and decides CFG_{ALL} , then $C = \neg Oracle(R(\langle M, w \rangle))$ decides H . R can be implemented as a Turing machine. And C is correct:

- If $\langle M, w \rangle \in H$: M halts on w so there exists a computation history of M on w . So there is a string that G does not generate. *Oracle*($\langle G \rangle$) rejects. C accepts.
- If $\langle M, w \rangle \notin H$: M does not halt on w , so there exists no computation history of M on w . G generates Σ^* . *Oracle*($\langle G \rangle$) accepts. C rejects.

But no machine to decide H can exist, so neither does *Oracle*. ■

22.5.2 Using the Undecidability of CFG_{ALL}

Now that we have proven our first result about the undecidability of a question about context-free grammars, others can be proven by reduction from it. For example:

Theorem 22.6 “Are Two CFGs Equivalent?” is Undecidable

Theorem: The language $GG_{=} = \{\langle G_1, G_2 \rangle : G_1 \text{ and } G_2 \text{ are CFGs and } L(G_1) = L(G_2)\}$ is not in D .

Proof: We show that $CFG_{ALL} \leq_M GG_{=}$ and so $GG_{=}$ is not in D . Let R be a mapping reduction from CFG_{ALL} to $GG_{=}$ defined as follows:

$R(\langle G \rangle) =$

1. Construct the description $\langle G\# \rangle$ of a new grammar $G\#$ such that $G\#$ generates Σ^* .
2. Return $\langle G\#, G \rangle$.

If *Oracle* exists and decides $GG_=$, then $C = Oracle(R(\langle G \rangle))$ decides CFG_{ALL} :

- R can be implemented as a Turing machine.
- C is correct:
 - If $\langle G \rangle \in CFG_{ALL}$: G is equivalent to $G\#$, which generates everything. $Oracle(\langle G\#, G \rangle)$ accepts.
 - If $\langle G \rangle \notin CFG_{ALL}$: G is not equivalent to $G\#$, which generates everything. $Oracle(\langle G\#, G \rangle)$ rejects.

But no machine to decide CFG_{ALL} can exist, so neither does *Oracle*. ■

Theorem 22.7 “Is One CFL a Subset of Another?” is Undecidable

Theorem: The language $\{\langle G_1, G_2 \rangle : G_1 \text{ and } G_2 \text{ are context-free grammars and } L(G_1) \subseteq L(G_2)\}$ is not in D.

Proof: The proof is by reduction from $GG_=$ and is left as an exercise. ■

The undecidability of so many questions about context-free languages makes optimizing programs to work with them more difficult than optimizing FSMs. For example, in Chapter 5 we described an algorithm for minimizing DFSMs. But now, in discussing context-free languages and PDAs, we must accept that the problem of determining whether one PDA is a minimization of another, is undecidable. This result can be proved quite easily by reduction from G_{ALL} .

Theorem 22.8 “Is One PDA a Minimization of Another?” is Undecidable

Theorem: The language $PDA_{MIN} = \{\langle M_1, M_2 \rangle : PDA M_2 \text{ is a minimization of PDA } M_1\}$ is undecidable.

Proof: We show that $CFG_{ALL} \leq_M PDA_{MIN}$ and so PDA_{MIN} is not in D. Before we start the reduction, recall that M_2 is a minimization of M_1 iff:

$$(L(M_1) = L(M_2)) \wedge M_2 \text{ is minimal.}$$

Let R be a mapping reduction from CFG_{ALL} to PDA_{MIN} defined as follows:

$R(\langle G \rangle) =$

1. Invoke $cfptoPDA_{topdown}(G)$ to construct the description $\langle P \rangle$ of a PDA that accepts the language that G generates.
2. Write $\langle P\# \rangle$ such that $P\#$ is a PDA with a single state s that is both the start state and an accepting state. Make a transition from s back to itself on each input symbol. Never push anything onto the stack. Note that $L(P\#) = \Sigma^*$ and $P\#$ is minimal.
3. Return $\langle P, P\# \rangle$.

If *Oracle* exists and decides PDA_{MIN} , then $C = Oracle(R(\langle G \rangle))$ decides CFG_{ALL} . R can be implemented as a Turing machine. And C is correct:

- If $\langle G \rangle \in CFG_{ALL}$: $L(G) = \Sigma^*$. So $L(P) = \Sigma^*$. Since $L(P\#) = \Sigma^*$, $L(P) = L(P\#)$. And $P\#$ is minimal. Thus $P\#$ is a minimization of P . $Oracle(\langle P, P\# \rangle)$ accepts.
- If $\langle G \rangle \notin CFG_{ALL}$: $L(G) \neq \Sigma^*$. So $L(P) \neq \Sigma^*$. But $L(P\#) = \Sigma^*$. So $L(P) \neq L(P\#)$. So $Oracle(\langle P, P\# \rangle)$ rejects.

But no machine to decide CFG_{ALL} can exist, so neither does *Oracle*. ■

22.5.3 Reductions from PCP

Several of the context-free language problems that we listed at the beginning of this section can be shown not to be decidable by showing that the Post Correspondence Problem is reducible to them. The key observation that forms the basis for those reductions is the following: Consider P , a particular instance of PCP. If P has a solution, then there is a sequence of indexes that makes it possible to generate the same string from the X list and from the Y list. If there isn't a solution, then there is no such sequence.

We start by defining a mapping between instances of PCP and context-free grammars. Recall that, given some nonempty alphabet Σ , an instance of PCP is a string of the form:

$$\langle P \rangle = (x_1, x_2, x_3, \dots, x_n)(y_1, y_2, y_3, \dots, y_n), \text{ where } \forall j (x_j \in \Sigma^+ \text{ and } y_j \in \Sigma^+).$$

To encode solutions to P , we'll need a way to represent the integers that correspond to the indexes. Since all the integers must be in the range $1:n$, we can do this with n symbols. So let Σ_n be a set of n symbols such that $\Sigma \cap \Sigma_n = \emptyset$. We'll use the j^{th} element of Σ_n to encode the integer j .

Given any PCP instance P , we'll define a grammar G_x that generates one string for every candidate solution to P . The string will have a second half that is the sequence of indices that is the solution, except that that sequence will be reversed. The first half of the string will be the concatenation of the elements from the X list that were selected by the indices. So suppose that $x_1 = aaa$, $x_2 = bbc$, and $x_3 = dd$. Then the index sequence 1, 2, 3 would produce the string $aaabbcdd$. So G_x will generate the string $aaabbcdd321$. (Note that the index sequence appears reversed.) We'll also build the grammar G_y , which does the same thing for the sequences that can be formed from the Y list. Note that there is no commitment at this point that any of the strings generated by either G_x or G_y corresponds to a solution of P . What we'll see in a moment is that a string s corresponds to such a solution iff it is generated by both G_x and G_y .

More formally, for any PCP instance P , define the following two grammars G_x and G_y :

- $G_x = (\{S_x\} \cup \Sigma \cup \Sigma_n, \Sigma \cup \Sigma_n, R_x, S_x)$, where R_x contains these two rules for each value of i between 1 and n :

$$S_x \rightarrow x_i S_x i$$

$$S_x \rightarrow x_i i$$

In both rules, i is represented by the i^{th} element of Σ_n .

- $G_y = (\{S_y\} \cup \Sigma \cup \Sigma_n, \Sigma \cup \Sigma_n, R_y, S_y)$, where R_y contains these two rules for each value of i between 1 and n :

$$S_y \rightarrow y_i S_y i$$

$$S_y \rightarrow y_i i$$

In both rules, i is represented by the i^{th} element of Σ_n .

Every string that G_x generates will be of the form $x_{i_1} x_{i_2} \dots x_{i_k} (i_1, i_2, \dots, i_k)^R$. Every string that G_y generates will be of the form $y_{i_1} y_{i_2} \dots y_{i_k} (i_1, i_2, \dots, i_k)^R$.

Any solution to P is a finite sequence i_1, i_2, \dots, i_k of integers such that:

$$\forall j (1 \leq i_j \leq n \text{ and } x_{i_1} x_{i_2} \dots x_{i_k} = y_{i_1} y_{i_2} \dots y_{i_k}).$$

If any such solution i_1, i_2, \dots, i_k exists, let $w = x_{i_1} x_{i_2} \dots x_{i_k} = y_{i_1} y_{i_2} \dots y_{i_k}$. Then both G_x and G_y will generate the string:

$$w(i_1, i_2, \dots, i_k)^R.$$

Example 22.6 Defining Grammars for a PCP Instance

Consider again PCP_1 , as defined in Example 22.1:

i	X	Y
1	b	bab
2	abb	b
3	aba	a
4	bbaaa	babaaa

PCP_1 is represented as the string $(b, abb, aba, bbaaa)(bab, b, a, babaaa)$.

The rules in G_x are:

$$\begin{aligned} S_x &\rightarrow bS_x1, S_x \rightarrow b1, \\ S_x &\rightarrow abbS_x2, S_x \rightarrow abb2, \\ S_x &\rightarrow abaS_x3, S_x \rightarrow aba3 \\ S_x &\rightarrow bbaaaS_x4, S_x \rightarrow bbaaa4 \end{aligned}$$

The rules in G_y are:

$$\begin{aligned} S_y &\rightarrow babS_y1, S_y \rightarrow bab1, \\ S_y &\rightarrow bS_y2, S_y \rightarrow b2, \\ S_y &\rightarrow aS_y3, S_y \rightarrow a3 \\ S_y &\rightarrow babaaaS_y4, S_y \rightarrow babaaa4 \end{aligned}$$

G_x generates strings of the form wv^R , where w is a sequence of strings from column X and v is the sequence of indices that were used to form w . G_y does the same for strings from column Y . So, for example, since 1, 2, 3, 1 is a solution to PCP_1 , G_x can generate the following string (with blanks inserted to show the structure and with the index sequence reversed):

b abb aba b 1321

G_y can also generate that string, although it derives it differently:

bab b a bab 1321

Using the ideas that we have just described, we are ready to show that some significant questions about the context-free languages are undecidable. We'll do so by converting each question to a language and then exhibiting a reduction from $PCP = \{ \langle P \rangle : P \text{ has a solution} \}$ to that language.

Theorem 22.9 "Is the Intersection of Two CFLs Empty?" is Undecidable

Theorem: The language $IntEmpty = \{ \langle G_1, G_2 \rangle : G_1 \text{ and } G_2 \text{ are context-free grammars and } L(G_1) \cap L(G_2) = \emptyset \}$ is not in D.

Proof: We show that $IntEmpty$ is not in D by reduction from $PCP = \{ \langle P \rangle : P \text{ has a solution} \}$. The reduction we will use exploits two functions, R and \neg . R will map instances of PCP to instances of $IntEmpty$. As before, \neg will simply invert *Oracle's* response (turning an *Accept* into a *Reject* and vice versa).

Define R as follows:

$$\begin{aligned} R(\langle P \rangle) = & \\ & \begin{aligned} 1. & \text{ From } P \text{ construct } G_x \text{ and } G_y \text{ as described above.} \\ 2. & \text{ Return } \langle G_x, G_y \rangle. \end{aligned} \end{aligned}$$

$\{R, \neg\}$ is a reduction from PCP to $IntEmpty$. If *Oracle* exists and decides $IntEmpty$, then $C = \neg Oracle(R(\langle P \rangle))$ decides PCP. R and \neg can be implemented as Turing machines. And C is correct:

- If $\langle P \rangle \in PCP$: P has at least one solution. So both G_x and G_y will generate some string:

$$w(i_1, i_2, \dots, i_k)^R, \text{ where } w = x_{i_1}x_{i_2}\dots x_{i_k} = y_{i_1}y_{i_2}\dots y_{i_k}.$$

So $L(G_1) \cap L(G_2) \neq \emptyset$. *Oracle*($\langle G_x, G_y \rangle$) rejects, so C accepts.

- If $\langle P \rangle \notin \text{PCP}$: P has no solution. So there is no string that can be generated by both G_x and G_y . So $L(G_1) \cap L(G_2) = \emptyset$. $\text{Oracle}(\langle G_x, G_y \rangle)$ accepts, so C rejects.

But no machine to decide PCP can exist, so neither does Oracle . ■

In Chapter 11 we spent a good deal of time worrying about whether the context-free grammars that we built were unambiguous. Yet we never gave an algorithm to determine whether or not a context-free grammar was ambiguous. Now we can understand why. No such algorithm exists.

Theorem 22.10 “Is a CFG Ambiguous?” is Undecidable

Theorem: The language $\text{CFG}_{\text{UNAMBIG}} = \{\langle G \rangle : G \text{ is a context-free grammar and } G \text{ is ambiguous}\}$ is not in D.

Proof: We show that $\text{PCP} \leq_M \text{CFG}_{\text{UNAMBIG}}$ and so $\text{CFG}_{\text{UNAMBIG}}$ is not in D. Let R be a mapping reduction from PCP to $\text{CFG}_{\text{UNAMBIG}}$ defined as follows:

$R(\langle P \rangle) =$

1. From P construct G_x and G_y as described above.
2. Construct G as follows:
 - 2.1. Add to G all the symbols and rules of both G_x and G_y .
 - 2.2. Add a new start symbol S and the two rules $S \rightarrow S_x$ and $S \rightarrow S_y$.
3. Return $\langle G \rangle$.

G generates $L(G_1) \cup L(G_2)$. Further, it does so by generating all the derivations that G_1 can produce as well as all the ones that G_2 can produce, except that each has a prepended $S \Rightarrow S_x$ or $S \Rightarrow S_y$.

If Oracle exists and decides $\text{CFG}_{\text{UNAMBIG}}$, then $C = \text{Oracle}(R(\langle P \rangle))$ decides PCP. R can be implemented as a Turing machine. And C is correct:

- If $\langle P \rangle \in \text{PCP}$: P has at least one solution. So both G_x and G_y will generate some string:

$$w(i_1, i_2, \dots, i_k)^R, \text{ where } w = x_{i_1}x_{i_2}\dots x_{i_k} = y_{i_1}y_{i_2}\dots y_{i_k}.$$

So G can generate that string in two different ways. G is ambiguous. $\text{Oracle}(\langle G \rangle)$ accepts.

- If $\langle P \rangle \notin \text{PCP}$: P has no solution. So there is no string that can be generated by both G_x and G_y . Since both G_x and G_y are unambiguous, so is G . $\text{Oracle}(\langle G \rangle)$ rejects.

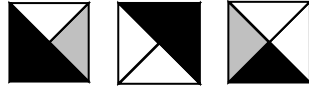
But no machine to decide PCP can exist, so neither does Oracle . ■

22.6 Exercises

- 1) Solve the linear Diophantine farmer problem presented in Section 22.1 .
- 2) Consider the following instance of the Post Correspondence Problem. Does it have a solution? If so, show one.

i	X	Y
1	a	bab
2	bbb	bb
3	aab	ab
4	b	a

- 3) Prove that, if we consider only PCP instances with a single character alphabet, PCP is decidable.
- 4) Prove that, if an instance of the Post Correspondence Problem has a solution, it has an infinite number of solutions.
- 5) Recall that the size of an instance P of the Post Correspondence Problem is the number of strings in its X list. Consider the following claim about the Post Correspondence problem: for any n , if P is a PCP instance of size n and if no string in either its X or its Y list is longer than n , then, if P has any solutions, it has one of length less than or equal to 2^n . Is this claim true or false? Prove your answer.
- 6) Let $\text{TILES} = \{ \langle T \rangle : \text{any finite surface on the plane can be tiled, according to the rules described in the book, with the tile set } T \}$. Let s be the string that encodes the following tile set:



Is $s \in \text{TILES}$? Prove your answer.

- 7) For each of the following languages L , state whether or not it is in D and prove your answer.
 - a) $\{ \langle G \rangle : G \text{ is a context-free grammar and } \varepsilon \in L(G) \}$.
 - b) $\{ \langle G \rangle : G \text{ is a context-free grammar and } \{ \varepsilon \} = L(G) \}$.
 - c) $\{ \langle G_1, G_2 \rangle : G_1 \text{ and } G_2 \text{ are context-free grammars and } L(G_1) \subseteq L(G_2) \}$.
 - d) $\{ \langle G \rangle : G \text{ is a context-free grammar and } \neg L(G) \text{ is context free} \}$.
 - e) $\{ \langle G \rangle : G \text{ is a context-free grammar and } L(G) \text{ is regular} \}$.

23 Unrestricted Grammars ✦

Consider a language like $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$. We know that we cannot write a context-free grammar for it. But could we create a new grammar formalism that is powerful enough to describe it and other languages like it? The answer to this question is yes. Recall that we moved from the power to define the regular languages to the power to define the context-free languages by removing constraints on the form of the rules that are allowed. We will do that again now. This time we will remove all constraints. We will prove that the class of languages that can be generated by one of these new, unrestricted grammars is exactly SD.

23.1 Definition and Examples

An *unrestricted grammar* G is a quadruple (V, Σ, R, S) , where:

- V is an alphabet that may contain terminal and nonterminal symbols,
- Σ (the set of terminals) is a subset of V ,
- R (the set of rules) is a finite subset of $V^+ \times V^*$,
- S (the start symbol) is an element of $V - \Sigma$.

Note that now the right-hand side of a rule may contain multiple symbols. So we might, have, for example:

$$\begin{aligned} a X a &\rightarrow a a a \\ b X b &\rightarrow a b a \end{aligned}$$

In this case, the derivation of X depends on its context. It is thus common to call rules like this “context-sensitive”. We will avoid using this terminology, however, because in the next chapter we will describe another formalism that we will call a context-sensitive grammar. While it, too, allows rules such as these, it does impose one important constraint that is lacking in the definition of an unrestricted grammar. It is thus less powerful, in a formal sense, than the system that we are describing here.

An unrestricted grammar G (just like a context-free grammar) derives strings by applying rules, beginning with its start symbol. So, to describe its behavior, we define the *derives-in-one-step* relation (\Rightarrow) analogously to the way we defined it for context-free grammars:

Given a grammar $G = (V, \Sigma, R, S)$, define $x \Rightarrow_G y$ to be a binary relation such that:

$$\begin{aligned} \forall x, y \in V^* (x \Rightarrow_G y \text{ iff } &x = \alpha\beta\phi, \\ &y = \alpha\gamma\phi, \\ &\alpha, \phi, \text{ and } \gamma \in V^*, \\ &\beta \in V^+, \text{ and} \\ &\text{there is a rule } \beta \rightarrow \gamma \text{ in } R). \end{aligned}$$

Any sequence of the form $w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \dots \Rightarrow_G w_n$ is called a derivation in G . As before, \Rightarrow_G^* is the reflexive, transitive closure of \Rightarrow_G .

The language generated by G is $\{w \in \Sigma^* : S \Rightarrow_G^* w\}$. So, just as before, $L(G)$ is the set of all strings of terminal symbols derivable from S via the rules of G .

Unrestricted grammars are sometimes called *phrase structure grammars* or *type 0 grammars*, the latter because of their place in the Chomsky hierarchy (which we will describe in Section **Error! Reference source not found.**). Some books also use the term *semi-Thue system* synonymously with unrestricted grammar. While the two formalisms are very similar, they model different computational processes and so must be considered separately. We will describe semi-Thue systems in Section 23.5.

Example 23.1 $A^nB^nC^n$

Consider $A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$. We build a grammar $G = (V, \{a, b, c\}, R, S)$, where V and R are as described below and $L(G) = A^nB^nC^n$. We first observe that any grammar for $A^nB^nC^n$ must generate all and only those strings with two properties:

- Equal numbers of a's, b's, and c's, and
- Letters in the correct order.

Just as with context-free grammars, the only way to guarantee that there are equal numbers of a's, b's, and c's is to generate them in parallel. The problem, though, is that there is no way to generate them in the correct order. For example, we could try a rule like:

$$S \rightarrow abSc$$

But if we apply that rule twice, we will generate the string $ababScC$. So what we will have to do is to generate each string in two phases:

1. Generate the correct number of each symbol.
2. Move the symbols around until they are in the correct order. This is the step that is possible in an unrestricted grammar but was not possible in a context-free one.

But we must be careful. As soon as G has generated a string that contains only terminal symbols, it is done, and that string is in $L(G)$. So we must make sure that, until the string is ready, it still contains at least one nonterminal. We can do that by creating one or more nonterminals that will stand in for their corresponding terminals and will be replaced once they have been moved into position. We'll use one such symbol, B .

We begin building the rules of G by inserting into R two rules that will generate strings with equal numbers of a's, b's, and c's:

- (1) $S \rightarrow aBSc$
- (2) $S \rightarrow \epsilon$

To generate the string $a^i b^i c^i$, we will apply rule (1) i times. Then we will apply rule 2 once. Suppose we want to generate $a^3 b^3 c^3$. Then we will apply rule (1) three times, then rule (2) once, and we will generate $aB a B a B c c c$. Because of the nonterminal symbol B , this string is not an element of $L(G)$. We still have the opportunity to rearrange the symbols, which we can do by adding one swapping rule:

- (3) $Ba \rightarrow aB$

Rule 3 can be applied as many times as necessary to push all the a's to the front of the string. But what *forces* it to be applied? The answer is the B 's. Until all the B 's are gone, the string that has been generated is not in $L(G)$. So we need rules to transform each B into b . We will design those rules so that they cannot be applied to any B until it is where it belongs. We can assure that with the following two rules:

- (4) $Bc \rightarrow bc$
- (5) $Bb \rightarrow bb$

Rule (4) transforms the rightmost B into b . Rule (5) transforms a B if it has an already transformed b directly to its right.

Having written a grammar such as the one we just built for $A^nB^nC^n$, can we prove that it is correct (i.e., that it generates exactly the strings in the target language?) Yes. Just as with a context-free grammar, we can prove that a grammar G is correct by:

1. Showing that G generates *only* strings in L , and
2. Showing that G generates *all* the strings in L .

We show 1 by defining an invariant I that is true of S and is maintained each time a rule in G is fired. Call the string that is being derived st . Then to prove the correctness of the grammar we just showed for $A^nB^nC^n$, we let I be:

$\#_a(st) = \#_b(st) + \#_c(st) = \#_c(st) \wedge$ all c 's occur to the right of all a 's, b 's, and B 's \wedge all b 's occur together and immediately to the left of the c region.

We show 2 by induction on n . Both of these steps are straightforward in this simple case. The same general strategy works for other unrestricted grammars, but it may be substantially more difficult to implement.

Next we consider another two examples of unrestricted grammars so that we can get a better idea of how they work.

Example 23.2 Equal Numbers of a', b's and c's

Let $L = \{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w)\}$. We build a grammar $G = (V, \{a, b, c\}, R, S)$, where V and R are as described below and $L(G) = L$. L is similar to $A^nB^nC^n$ except that the letters may occur in any order. So, again, we will begin by generating matched sets of a 's, b 's, and c 's. But, this time, we need to allow the second phase of the grammar to perform arbitrary permutations of the characters. So we will start with the rules:

- (1) $S \rightarrow ABSC$
- (2) $S \rightarrow \varepsilon$

Next we need to allow arbitrary permutations, so we add the rules:

- | | |
|-------------------------|-------------------------|
| (3) $AB \rightarrow BA$ | (6) $CA \rightarrow AC$ |
| (4) $BA \rightarrow AB$ | (7) $BC \rightarrow CB$ |
| (5) $AC \rightarrow CA$ | (8) $CB \rightarrow BC$ |

Finally, we need to generate terminal symbols. Remember that every rule in G must have at least one nonterminal on its left-hand side. In contrast to $A^nB^nC^n$, here the job of the nonterminals is not to *force* reordering but to *enable* it. This means that a nonterminal symbol can be replaced by its corresponding terminal symbol at any time. So we add the rules:

- (9) $A \rightarrow a$
- (10) $B \rightarrow b$
- (11) $C \rightarrow c$

Example 23.3 WW

Consider $WW = \{ww : w \in \{a, b\}^*\}$. We build a grammar $G = (V, \{a, b\}, R, S)$, where V and R are as described below and $L(G) = WW$. The strategy we will use is the following:

1. Generate $w C w^R \#$, where C will serve as a temporary middle marker and $\#$ will serve as a temporary right boundary. This is easy to do with a context-free grammar.
2. Reverse w^R . We will do that by viewing $\#$ as a wall and jumping the characters in w^R , *leftmost first*, over the wall.
3. Finally, clean up by removing C and $\#$.

Suppose that, after step 1, we have $aabCbbaa\#$. We let C spawn a pusher P , yielding:

$aabCPbaa\#$.

The job of P is to push the character just to its right rightward to the wall so that it can hop over. To do this, we will write rules that swap Pb with each character between it and the wall. Those rules will generate the following sequence of strings:

$aabCaPba\#$
 $aabCaaPb\#$

The last step in getting the pushed character (in this case, b) where it belongs is to jump it over the wall and then erase P , yielding:

$aabCaa\#b$

Next, C will spawn another pusher P and use it to push the first a up to the wall. At that point, we will have:

$aabCaPa\#b$

Then a jumps the wall, landing immediately after it, yielding:

$aabCa\#ab$

Notice that the substring ba has now become ab , as required. Now the remaining a can be pushed and jumped, yielding:

$aabC\#aab$

The final step is to erase $C\#$ as soon as they become adjacent to each other.

The following set of rules R implements this plan:

$S \rightarrow T\#$	/* Generate the wall exactly once.
$T \rightarrow aTa$	/* Generate wCw^R .
$T \rightarrow bTb$	"
$T \rightarrow C$	"
$C \rightarrow CP$	/* Generate a pusher P .
$Pa a \rightarrow aPa$	/* Push one character to the right to get ready to jump.
$Pab \rightarrow bPa$	"
$Pba \rightarrow aPb$	"
$Pbb \rightarrow bPb$	"
$Pa\# \rightarrow \#a$	/* Hop a character over the wall.
$Pb\# \rightarrow \#b$	"
$C\# \rightarrow \varepsilon$	

We have described the way that we want G to work. It clearly can do exactly what we have said it will do. But can we be sure that it does nothing else? Remember that, at any point, any rule whose left-hand side matches can be applied. So, for example, what prevents C from spawning a new pusher P before the first character has jumped the wall? Nothing. But the correctness of G is not affected by this. Pushers (and the characters they are pushing) cannot jump over each other. If C spawns more pushers than are necessary, the resulting string cannot be transformed into a string containing just terminal symbols. So any path that does that dies without generating any strings in $L(G)$.

If we want to decrease the number of dead-end paths that a grammar G can generate, we can write rules that have more restrictive left-hand sides. So, for example, we could replace the rule:

$C \rightarrow CP$ /* Generate a pusher P .

with the rules:

$$\begin{array}{ll} C_a \rightarrow CP_a & /* \text{Generate a pusher } P. \\ C_b \rightarrow CP_b & " \end{array}$$

Now C can only generate one pusher for each character in w^R .

Unrestricted grammars often have a strong procedural feel that is typically absent from restricted grammars. Derivations usually proceed in phases. When we design a grammar G , we make sure that the phases work properly by using nonterminals as flags that tell G what phase it is in. It is very common to have three phases:

- Generate the right number of the various symbols.
- Move them around to get them in the right order.
- Clean up.

In implementing these phases, there are some quite common idioms:

- Begin by creating a left wall, a right wall, or both.
- Reverse a substring by pushing the characters, one at a time, across the wall at the opposite end from where the first character originated.
- Use nonterminals to represent terminals that need additional processing (such as shifting from one place to another) before the final string should be generated.

Now that we have seen the extent to which unrestricted grammars can feel like programs, it may come as no surprise that they are general purpose computing devices. In Section 23.3 we will show how they can be used not just to define languages but also to compute functions. But first we will show that the class of languages that can be generated by an unrestricted grammar is exactly SD. So, sadly, although these grammars can be used to define decidable languages like $A^nB^nC^n$, unfortunately there is no parsing algorithm for them. Given an unrestricted grammar G and a string w , it is undecidable whether G generates w . We will prove that in Section 23.2.

23.2 Equivalence of Unrestricted Grammars and Turing Machines

Recall that, in our discussion of the Church-Turing thesis, we mentioned several formalisms that can be shown to be equivalent to Turing machines. We can now add unrestricted grammars to our list.

Since rewrite systems can have the same computational power as the Turing machine, they have been used to define programming languages such as Prolog. © 762.

Theorem 23.1 Turing Machines and Unrestricted Grammars Describe the Same Class of Languages

Theorem: A language L is generated by an unrestricted grammar iff it is semidecided by some Turing machine M .

Proof: We will prove each direction of this claim separately:

- a) We show that the existence of an unrestricted grammar G for L implies the existence of a semideciding Turing machine for L . We do this by construction of a nondeterministic Turing machine that, on input x , simulates applying the rules of G , checking at each step to see whether G has generated x .
- b) We show that the existence of a semideciding Turing machine M for L implies the existence of an unrestricted grammar for L . We do this by construction of a grammar that mimics the execution of M .

Proof of claim a: Given an unrestricted grammar $G = (V, \Sigma, R, S)$, we construct a nondeterministic Turing machine M that semidecides $L(G)$. The idea is that M , on input x , will start with S , apply rules in R , and see whether it can generate x . If it ever does, it will halt and accept. M will be nondeterministic so that it can try all possible derivations in G . Each nondeterministic branch will use two tapes:

- Tape 1 holds M 's input string x .
- Tape 2 holds the string that has so far been derived using G .

At each step, M nondeterministically chooses a rule to try to apply and a position on tape 2 to start looking for the left-hand side of the rule. If the rule's left-hand side matches, M applies the rule. Then it checks whether tape 2 equals tape 1. If any such branch succeeds in generating x , M accepts. Otherwise, it keeps looking. If a branch generates a string to which no rules in R can apply, it rejects. So some branch of M accepts iff there is a derivation of x in G . Thus M semidecides $L(G)$.

Proof of claim b: Given a semideciding Turing machine $M = (K, \Sigma, \Gamma, \delta, s, H)$, we construct an unrestricted grammar $G = (\{ \#, q, 0, 1, A \} \cup \Gamma, \Sigma, R, S)$ such that $L(G) = L(M)$. The idea is that G will exploit a generate-and-test strategy in which it first creates candidate strings in Σ^* and then simulates running M on them. If there is some string s that M would accept, then G will cleanup its working symbols and generate s . G operates in three phases:

- Phase 1 can generate all strings of the following form, where q_s is the binary encoding in $\langle M \rangle$ of M 's start state, n is any positive integer, and each of the characters a_1 through a_n is in Σ^* :

$$\# \square \square q_s a_1 a_1 a_2 a_2 a_3 a_3 \dots a_n a_n \square \square \#$$

The #'s will enable G to exploit rules that need to locate the beginning and the end of the string that it has derived so that they can scan the string. The rest of the string directly encodes M 's state and the contents of its tape (with each tape symbol duplicated). It also encodes the position of M 's read/write head by placing the encoding of the state immediately to the left of the character under the read/write head. So the strings that are generated in Phase 1 can be used in Phase 2 to begin simulating M , starting in state q_s , with the string $a_1 a_2 a_3 \dots a_n$ on its tape and the read/write head positioned immediately to the left of a_1 . Each character on the tape is duplicated so that G can use the second instance as though it were on the tape, writing on top of it as necessary. Then, if M accepts, G will use the first instance to reconstruct the input string that was accepted.

- Phase 2 simulates the execution of M on a particular string w . So, for example, suppose that Phase 1 generated the following string:

$$\# \square \square q000 a a b b c c b b a a \square \square \#$$

Then Phase 2 begins simulating M on the string $abcba$. The rules of G are constructed from δ_M . At some point, G might generate

$$\# \square \square a 1 b 2 c c b 4 q011 a 3 \square \square \#$$

if M , when invoked on $abcba$ could be in state 3, with its tape equal to $12c43$, and its read/write head positioned on top of the final 3.

To implement Phase 2, G contains one or more rules for each element of δ_M . The rules look like these:

$$q100 b b \rightarrow b 2 q101 \quad /* \text{ if } M, \text{ in state 4 looking at } b, \text{ would rewrite it as } 2, \text{ go to state 5, and move right.} */$$

$$a a q011 b 4 \rightarrow q011 a a b 4 \quad /* \text{ if } M, \text{ in state 3 looking at } 4, \text{ would rewrite it as } 4, \text{ go to state 3, and move left. Notice that to encode moving left we must create a separate rule for each pair of characters that could be to the left of the read/write head.} */$$

In Phase 2, all of M 's states are encoded in their standard binary form except its accepting state(s), all of which will be encoded as Λ .

- Phase 3 cleans up by erasing all the working symbols if M ever reaches an accepting state. It will leave in its derived string only those symbols corresponding to the original string that M accepted. Once the derived string contains only terminal symbols, G halts and outputs the string.

To implement Phase 3, G contains one rule of the following form for each character other than Λ and $\#$ in its alphabet:

$$x \Lambda \rightarrow \Lambda x \quad /* \text{ If } M \text{ ever reaches an accepting state, sweep } \Lambda \text{ all the way to the left of the string until it is next to } \#.$$

It also has one rule of the following form for each pair x, y of characters other than Λ and $\#$:

$$\# \Lambda x y \rightarrow x \# \Lambda \quad /* \text{ Sweep } \# \Lambda \text{ rightward, deleting the working copy of each symbol and keeping the original version.}$$

And then it has the final rule:

$$\# \Lambda \# \rightarrow \varepsilon \quad /* \text{ At the end of the sweep, wipe out the last working symbols.}$$

23.3 Grammars Compute Functions

We have now shown that grammars and Turing machines are equivalent in their power to define languages. But we also know that Turing machines can do something else: they can compute functions by leaving a meaningful result on their tape when they halt. Can unrestricted grammars do that as well? The answer is yes. Suppose that, instead of starting with just the start symbol S , we allow a grammar G to be invoked with some input string w . G would apply its rules as usual. If it then halted, having derived some new string w' that is the result of applying function f to w , we could say that G computed f . The only details we need to work out are how to format the input so G will be able to work effectively and how to tell when G should halt.

We say that a grammar G *computes* f iff, $\forall w, v \in \Sigma^* (S w S \Rightarrow^* v \leftrightarrow v = f(w))$. We use G 's start symbol S to solve both of the problems we just mentioned: S serves as a delimiter for the input string so that G will be able to perform actions like, "Start at the left-hand end of the string and move rightward doing something". And, as usual, G continues to apply rules until either there are no more rules that can be applied or the derived string is composed entirely of terminal symbols. So, to halt and report a result, G must continue until both delimiting S 's are removed. A function f is called *grammatically computable* iff there is a grammar G that computes it.

Recall the family of functions, $value_k(n)$. For any positive integer k , $value_k(n)$ returns the nonnegative integer that is encoded, base k , by the string n . For example $value_2(101) = 5$.

Example 23.4 Computing the Successor Function in Unary

Let f be the successor function $succ(n)$ on the unary representations of natural numbers. Specifically, define $f(n) = m$, where $value_1(m) = value_1(n) + 1$. If G is to compute f , it will need to produce derivations such as:

$$\begin{aligned} S1S &\Rightarrow^* 11 \\ S1111S &\Rightarrow^* 11111 \end{aligned}$$

We need to design G so that it adds exactly one more 1 to the input string and gets rid of both S 's. The following two-rule grammar $G = (\{S, 1\}, \{1\}, R, S)$ does that with $R =$

$$\begin{aligned} S11 &\rightarrow 1S1 & /* \text{ Move the first } S \text{ rightward past all but the last } 1. \\ S1S &\rightarrow 11 & /* \text{ When it reaches the last } 1, \text{ add a } 1 \text{ and remove the } S\text{'s.} \end{aligned}$$

Example 23.5 Multiplying by 2 in Unary

Let $f(n) = m$, where $value_1(n)$ is a natural number and $value_1(m) = 2 \cdot value_1(n)$. G should produce derivations such as:

$S11S \Rightarrow^* 1111$
 $S1111S \Rightarrow^* 11111111$

G needs to go through its input string, turning every 1 into 11. Again, a simple two-rule grammar $G = (\{S, 1\}, \{1\}, R, S)$ is sufficient, with $R =$

$S1 \rightarrow 11S$ /* Starting from the left, duplicate a 1. Shift the initial S so that only the 1's that still need to be duplicated are on its right.
 $SS \rightarrow \epsilon$ /* When all 1's have been duplicated, stop.

Example 23.6 Squeezing Out Extra Blanks

Let $f(x: x \in \{a, b, \square\}^*) = x$ except that extra blanks will be squeezed out. More specifically, blanks will be removed so that there is never more than one \square between “words”, i.e., sequences of a's and b's, and there are no leading or trailing \square 's}. G should produce derivations such as:

$Saa\square b\square\square aa\square\square\square\square b\square S \Rightarrow^* aa\square b\square aa\square b$
 $S\square aa\square b\square\square aa\square b\square\square S \Rightarrow^* aa\square b\square aa\square b$

This time, G is more complex. It must reduce every $\square\square$ string to \square . And it must get rid of *all* \square 's that occur adjacent to either S .

$G = (\{S, T, a, b, \square\}, \{a, b, \square\}, R, S)$, where $R =$

$S\square \rightarrow S$ /* Get rid of leading \square 's. All blanks get squeezed, not just repeated ones.
 $Sa \rightarrow aT$ /* T replaces S to indicate that we are no longer in a leading \square 's region. For the rest of G 's operation, all characters to the left of T will be correct. Those to the right still need to be processed.
 $Sb \rightarrow bT$ "
 $Ta \rightarrow aT$ /* Sweep T across a's and b's.
 $Tb \rightarrow bT$ "
 $T\square\square \rightarrow T\square$ /* Squeeze repeated \square 's.
 $T\square a \rightarrow \square aT$ /* Once there is a single \square , sweep T past it and the first letter after it.
 $T\square b \rightarrow \square bT$ "
 $T\square S \rightarrow \epsilon$ /* The $T\square\square$ rule will get rid of all but possibly one \square at the end of the string.
 $TS \rightarrow \epsilon$ /* If there were no trailing \square 's, this rule finishes up.

From this last example, it is easy to see how we can construct a grammar G to compute some function f . G can work in very much the way a Turing machine would, sweeping back and forth through its string. In fact, it is often easier to build a grammar to compute a function than it is to build a Turing machine because, with grammars, we do not have to worry about shifting the string if we want to add or delete characters from somewhere in the middle.

Recall that in Section 17.2.2, we defined a *computable function* to be a function that can be computed by a Turing machine that always halts and, on input x , leaves $f(x)$ on its tape. We now have an alternative definition:

Theorem 23.2 Turing Machines and Unrestricted Grammars Compute the Same Class of Functions

Theorem: A function f is computable iff it is grammatically computable. In other words a function f can be computed by a Turing machine iff there is an unrestricted grammar that computes it.

Proof: The proof requires two constructions. The first shows that, for any grammar G , there is a Turing machine M that simulates G , halts whenever G produces a terminal string s , and leaves s on its tape when it halts. The second shows the other direction: for any Turing machine M , there is a grammar G that simulates M and produces a terminal string s whenever M halts with s on its tape. These constructions are similar to the ones we used to prove Theorem 23.1. We omit the details. ■

23.4 Undecidable Problems About Unrestricted Grammars

Consider the following questions that we might want to ask about unrestricted grammars:

- Given an unrestricted grammar G and a string w , is $w \in L(G)$?
- Given an unrestricted grammar G , is $\varepsilon \in L(G)$?
- Given two unrestricted grammars G_1 and G_2 , is $L(G_1) = L(G_2)$?
- Given an unrestricted grammar G , is $L(G) = \emptyset$?

Does there exist a decision procedure to answer any of these questions? Or, formulating these problems as language recognition tasks, are any of these languages decidable:

- $L_a = \{ \langle G, w \rangle : G \text{ is an unrestricted grammar and } w \in L(G) \}$.
- $L_b = \{ \langle G \rangle : G \text{ is an unrestricted grammar and } \varepsilon \in L(G) \}$.
- $L_c = \{ \langle G_1, G_2 \rangle : G_1 \text{ and } G_2 \text{ are unrestricted grammars and } L(G_1) = L(G_2) \}$.
- $L_d = \{ \langle G \rangle : G \text{ is an unrestricted grammar and } L(G) = \emptyset \}$.

The answer to all these questions is, “no”. If any of these problems involving grammars were solvable, then the corresponding problem involving Turing machines would also be solvable. But it isn’t. We can prove each of these cases by reduction. We will do one here and leave the others as exercises. They are very similar.

Theorem 23.3 Undecidability of Unrestricted Grammars

Theorem: The language $L_a = \{ \langle G, w \rangle : G \text{ is an unrestricted grammar and } w \in L(G) \}$ is not in D.

Proof: We show that $A \leq_M L_a$ and so L_a is not decidable. Let R be a mapping reduction from $A = \{ \langle M, w \rangle : \text{Turing machine } M \text{ accepts } w \}$ to L_a , defined as follows:

$R(\langle M, w \rangle) =$

1. From M , construct the description $\langle G\# \rangle$ of a grammar $G\#$ such that $L(G\#) = L(M)$.
2. Return $\langle G\#, w \rangle$.

If *Oracle* exists and decides L_a , then $C = \text{Oracle}(R(\langle M, w \rangle))$ decides A . R can be implemented as a Turing machine using the algorithm presented in Section 23.2. And C is correct:

- If $\langle M, w \rangle \in A$: $M(w)$ halts and accepts. $w \in L(M)$. So $w \in L(G\#)$. *Oracle*($\langle G\#, w \rangle$) accepts.
- If $\langle M, w \rangle \notin A$: $M(w)$ does not halt. $w \notin L(M)$. So $w \notin L(G\#)$. *Oracle*($\langle G\#, w \rangle$) rejects.

But no machine to decide A can exist, so neither does *Oracle*. ■

So, unrestricted grammars, although powerful, are very much less useful than context-free ones, since even the most basic question, “Given a string w , does G generate w ?” is undecidable.

23.5 The Word Problem for Semi-Thue Systems

Unrestricted grammars can generate languages and they can compute functions. A third way of characterizing their computational power has played an important historical role in the development of formal language theory. Define a **word problem** to be the following:

Given two strings, w and v , and a rewrite system T , determine whether v can be derived from w using T .

An important application of the word problem is in logical reasoning. If we can encode logical statements as strings (in the obvious way) and if we can also define a rewrite system T that corresponds to a set of inference rules, then determining whether v can be rewritten as w using T is equivalent to deciding whether the sentence that corresponds to v is entailed by the sentence that corresponds to w .

Any rewrite system whose job is to transform one string into another must be able to start with an arbitrary string (not just some unique start symbol). Further, since both the starting string and the ending one may contain any symbols in the alphabet of the system, the distinction between terminal symbols and working symbols goes away. Making those two changes to the unrestricted grammar model we get:

A **semi-Thue system** T is a pair (Σ, R) , where:

- Σ is an alphabet,
- R (the set of rules) is a subset of $\Sigma^+ \times \Sigma^*$.

Semi-Thue systems were named after their inventor, the Norwegian mathematician Axel Thue. Just as an aside: A **Thue system** is a semi-Thue system with the additional property that if R contains the rule $x \rightarrow y$ then it also contains the rule $y \rightarrow x$.

We define for semi-Thue systems the *derives-in-one-step* relation (\Rightarrow_T) and its transitive closure *derives* (\Rightarrow_T^*) exactly as we did for unrestricted grammars. Since there is no distinguished start symbol, it doesn't make sense to talk about the language that can be derived from a semi-Thue system. It does, however, make sense to talk about the word problem: Given a semi-Thue system T and two strings w and v , determine whether $w \Rightarrow_T^* v$. We have already seen that it is undecidable, for an unrestricted grammar with a distinguished start symbol S , whether S derives some arbitrary string w . So too it is undecidable, for a semi-Thue system, whether one arbitrary string can derive another.

Theorem 23.4 Undecidability of the Word Problem for Semi-Thue Systems

Theorem: The word problem for semi-Thue systems is undecidable. In other words, given a semi-Thue system T and two strings w and v , it is undecidable whether $w \Rightarrow_T^* v$.

Proof: The proof is by reduction from the halting problem language $H = \{\langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w\}$. Given a Turing machine M and an input string w , we will build a semi-Thue system T with the property that M halts on w iff $w \Rightarrow_T^* S$. The construction that is used in the reduction mirrors the construction that we used to prove (Theorem 23.1) that Turing machines and unrestricted grammars define the same class of languages. Given a Turing machine M , we first build a semi-Thue system T whose rules simulate the operation of M . Assume that the symbol S has not been used in the construction so far. We now add rules to T as follows:

- For every halting state q in M , add the rule $q \rightarrow S$. These rules guarantee that, if M eventually enters some halting state, the symbol S will appear in a string derived by T . Since S is an otherwise unused symbol, that is the only case in which it will appear in a string that T derives.
- For every other symbol c in T 's alphabet, add the rules $cS \rightarrow S$ and $S_c \rightarrow S$. These rules enable T to transform any string containing the symbol S into a string that is just S .

So if, on input w , M would ever enter a halting state, the rules of T will enable w to derive some string that contains the symbol S . That string will then derive the string consisting of only S . And M entering a halting state is the only

way to generate an S . So M halts on input w iff $w \Rightarrow_T^* S$. Thus, if the word problem were decidable, H would also be. But it isn't. ■

23.6 Exercises

- 1) Show an unrestricted grammar that generates each of the following languages:
 - a) $\{a^{2^n} b^{2^n}, n \geq 0\}$.
 - b) $\{a^m b^m c^{n+m} : n, m > 0\}$.
 - c) $\{a^m b^m c^{nm} : n, m > 0\}$.
 - d) $\{a^n b^{2^n} c^{3^n} : n \geq 1\}$.
 - e) $\{ww^R w : w \in \{a, b\}^*\}$.
 - f) $\{a^n b^n a^n b^n : n \geq 0\}$.
 - g) $\{xy\#x^R : x, y \in \{a, b\}^* \text{ and } |x| = |y|\}$.
 - h) $\{wc^m d^n : w \in \{a, b\}^* \text{ and } m = \#_a(w) \text{ and } n = \#_b(w)\}$.

- 2) Show a grammar that computes each of the following functions (given the input convention described in Section 23.3):
 - a) $f: \{a, b\}^+ \rightarrow \{a, b\}^+$, where $f(s = a_1 a_2 a_3 \dots a_{|s|}) = a_2 a_3 \dots a_{|s|} a_1$. For example $f(aabbbaa) = abbbaaa$.
 - b) $f: \{a, b\}^+ \rightarrow \{a, b, 1\}^+$, where $f(s) = s1^n$, where $n = \#_a(s)$. For example $f(aabbbaa) = aabbbaa11111$.
 - c) $f: \{a, b\}^* \# \{a, b\}^* \rightarrow \{a, b\}^*$, where $f(x\#y) = xy^R$.
 - d) $f: \{a, b\}^+ \rightarrow \{a, b\}^+$, where $f(s) =$ if $\#_a(s)$ is even then s , else s^R .
 - e) $f: \{a, b\}^* \rightarrow \{a, b\}^*$, where $f(w) = ww$.
 - f) $f: \{a, b\}^+ \rightarrow \{a, b\}^*$, where $f(s) =$ if $|s|$ is even then s , else s with the middle character chopped out. (Hint: the answer to this one is fairly long, but it is not very complex. Think about how you would use a Turing machine to solve this problem.)
 - g) $f(n) = m$, where $value_1(n)$ is a natural number and $value_1(m) = value_1(\lfloor n/2 \rfloor)$. Recall that $\lfloor x \rfloor$ (read as "floor of x ") is the largest integer that is less than or equal to x .
 - h) $f(n) = m$, where $value_2(n)$ is a natural number and $value_2(m) = value_2(n) + 5$.

- 3) Show that, if G is an unrestricted grammar, then each of the following languages, defined in Section 23.4, is not in D :
 - a) $L_b = \{\langle G \rangle : \varepsilon \in L(G)\}$.
 - b) $L_c = \{\langle G_1, G_2 \rangle : L(G_1) = L(G_2)\}$.
 - c) $L_d = \{\langle G \rangle : L(G) = \emptyset\}$.

- 4) Show that, if G is an unrestricted grammar, then each of the following languages is not in D :
 - a) $\{\langle G \rangle : G \text{ is an unrestricted grammar and } a^* \subseteq L(G)\}$.
 - b) $\{\langle G \rangle : G \text{ is an unrestricted grammar and } G \text{ is ambiguous}\}$. Hint: Prove this by reduction from PCP.

- 5) Let G be the unrestricted grammar for the language $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$, shown in Example 23.1. Consider the proof, given in \mathfrak{B} 649, of the undecidability of the Post Correspondence Problem. The proof is by reduction from the membership problem for unrestricted grammars.
 - a) Define the MPCP instance MP that will be produced, given the input $\langle G, abc \rangle$, by the reduction that is defined in the proof of Theorem 36.1.
 - b) Find a solution for MP .
 - c) Define the PCP instance P that will be built from MP by the reduction that is defined in the proof of Theorem 36.2.
 - d) Find a solution for P .

24 The Chomsky Hierarchy and Beyond ❖

So far, we have described a hierarchy of language classes, including the regular languages, the context-free languages, the decidable languages (D), and the semidecidable languages (SD). The smaller classes have useful properties, including efficiency and decidability, that the larger classes lack. But they are more limited in what they can do. In particular, PDAs are not powerful enough for most applications. But, to do better, we have jumped to Turing machines and, in so doing, have given up the ability to decide even the most straightforward questions.

So the question naturally arises, “Are there other formalisms that can effectively describe useful languages?” The answer to that question is yes and we will consider a few of them in this chapter.

24.1 The Context-Sensitive Languages

We would like a computational formalism that accepts exactly the set D. We have one: the set of Turing machines that always halt. But that set is itself undecidable. What we would like is a computational model that comes close to describing exactly the class D but that is itself decidable in the sense that we can look at a program and tell whether or not it is an instance of our model.

In this section we’ll describe the *context-sensitive languages*, which fit into our existing language hierarchy as shown in Figure 24.1. The context-sensitive languages can be decided by a class of automata called linear bounded automata. They can also be described by grammars we will call context-sensitive (because they allow multiple symbols on the left-hand sides of rules). The good news about the context-sensitive languages is that many interesting languages that are not context-free are context-sensitive. But the bad news is that, while a parsing algorithm for them does exist, no efficient one is known nor is one likely to be discovered.

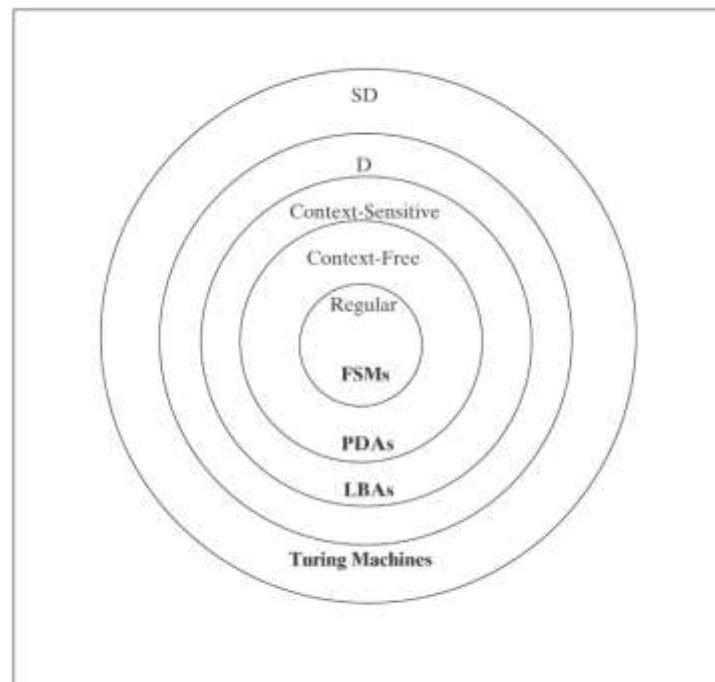


Figure 24.1: A hierarchy of language classes

24.1.1 Linear Bounded Automata

There are two common definitions of a linear bounded automaton (or LBA). The crucial aspect of both is that an LBA is a Turing machine whose tape is limited by the length of its input. The two definitions can be stated informally as:

1. An LBA is a Turing machine that cannot move past the blank square on either side of its input.
2. An LBA is a Turing machine that cannot use more than $k \cdot |w|$ tape squares, where w is its input and k is some fixed positive integer.

The second definition seems, at first glance, to be less restrictive, since it allows for additional working space on the tape. But, in fact, the two definitions are equivalent, since we can implement a definition₂ LBA as a definition₁ LBA whose tape is divided into tracks to simulate k tapes. So it is just a tradeoff between more tape squares and a larger tape alphabet. Because the first definition is slightly simpler, we will use it.

A **linear bounded automaton** (or LBA) $B = (K, \Sigma, \Gamma, \Delta, s, H)$ is a nondeterministic Turing machine that cannot move off the tape region that starts at the blank to the left of the input and ends at the blank immediately after the input. If an LBA attempts to move off that region, the read/write head simply stays where it is. This definition is slightly nonstandard. The usual one restricts the head to the input string proper, but the version we give here lets us maintain the programming style that we have been using, in which the read/write head starts on the blank just to the left of the input and we detect the end of the input when we find the blank immediately to its right.

A language L is **context-sensitive** iff there exists an LBA that accepts it.

Almost all of the deterministic deciding Turing machines that we have described so far have been LBAs. For example, the machines we built in Chapter 17 for $A^n B^n C^n$ and $W \subset W$ are both LBAs. So $A^n B^n C^n$ and $W \subset W$ are context-sensitive languages.

And now to the reason that it made sense to define the LBA: the halting problem for LBAs is decidable and thus the membership question for context-sensitive languages is decidable.

Theorem 24.1 Decidability of LBAs

Theorem: The language $L = \{ \langle B, w \rangle : \text{LBA } B \text{ accepts } w \}$ is in D.

Proof: Although L looks very much like A , the acceptance language for Turing machines, its one difference, namely that it asks about an LBA rather than an arbitrary Turing machine, is critical. We observe the following property of an LBA B operating on some input w : B can be in any one of its $|K|$ states. The tape that B can look at has exactly $|w| + 2$ squares. Each of those squares can contain any value in Γ and the read/write head can be on any one of them. So the number of distinct configurations of B is:

$$\text{MaxConfigs} = |K| \cdot |\Gamma|^{(|w|+2)} \cdot (|w| + 2).$$

If B ever reaches a configuration that it has been in before, it will do the same thing the second time that it did the first time. So, if it runs for more than MaxConfigs steps, it is in a loop and it is not going to halt.

We are now ready to define a nondeterministic Turing machine that decides L :

$$M(\langle B, w \rangle) =$$

1. Simulate all paths of B on w , running each for MaxConfigs steps or until B halts, whichever comes first.
2. If any path accepted, accept; else reject.

Since, from each configuration of B , there is a finite number of branches and each branch is of finite length, M will be able to try all branches of B in a finite number of steps. M will accept the string $\langle B, w \rangle$ if any path of B , running on w , accepts (i.e., B itself would accept) and it will reject the string $\langle B, w \rangle$ if every path of B on w either rejects or loops. ■

We defined an LBA to be a *nondeterministic* Turing machine with bounded tape. Does nondeterminism matter for LBAs? Put another way, for any nondeterministic LBA B does there exist an equivalent deterministic LBA? No one knows.

24.1.2 Context-Sensitive Grammars

Why have we chosen to call the class of languages that can be accepted by an LBA “context-sensitive”? Because there exists a grammar formalism that exactly describes these languages and this formalism, like the unrestricted grammar formalism on which it is based, allows rules whose left-hand sides describe the context in which the rules may be applied.

A context-sensitive grammar $G = (V, \Sigma, R, S)$ is an unrestricted grammar in which R satisfies the following constraints:

- The left-hand side of every rule contains at least one nonterminal symbol.
- If R contains the rule $S \rightarrow \epsilon$ then S does not occur on the right-hand side of any rule.
- With the exception of the rule $S \rightarrow \epsilon$, if it exists, every rule $\alpha \rightarrow \beta$ in R has the property that $|\alpha| \leq |\beta|$. In other words, with the exception of the rule $S \rightarrow \epsilon$, there are no length-reducing rules in R .

We should point out here that this definition is a bit nonstandard. The more common definition allows no length-reducing rules at all. But without the exception for $S \rightarrow \epsilon$, it is not possible to generate any language that contains ϵ . So the class of languages that could be generated would not include any of the classes that we have so far considered.

We define \Rightarrow (*derives-in-one-step*), \Rightarrow^* (*derives*), and $L(G)$ analogously to the way they were defined for context-free and unrestricted grammars.

Some of the grammars (both context-free and unrestricted) that we have written so far are context-sensitive. But many are not.

Example 24.1

For the language A^nB^n , we wrote the grammar:

$$\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow \epsilon \end{aligned}$$

That grammar is not context-sensitive. But the following equivalent grammar is:

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow S_1 \\ S_1 &\rightarrow aS_1b \\ S_1 &\rightarrow ab \end{aligned}$$

Because of the prohibition against length-reducing rules, the problem of determining whether a context-sensitive grammar G generates some string w is decidable. Recall that it was not decidable for unrestricted grammars, so this is a significant change.

Theorem 24.2 Decidability of Context-Sensitive Grammars

Theorem: The language $L = \{ \langle G, w \rangle : \text{context sensitive grammar } G \text{ generates string } w \}$ is in D.

Proof: We construct a nondeterministic Turing machine M to decide L . M will explore all derivations that G can produce starting from its start symbol. Eventually one of the following things must happen on every derivation path:

- G will generate w .
- G will generate a string to which no rules can be applied. The path will end.

- G will keep generating strings of the same length. Since there is a finite number of strings of a given length, G must eventually generate the same one twice. Whenever that happens, the path can be terminated since it is not getting any closer to generating w .
- G will generate a string s that is longer than w . The path can be terminated. Since there are no length-reducing rules, there is no way that w could ever be derived from s . It is this case that distinguishes context-sensitive grammars from unrestricted ones.

Since G has only a finite number of choices at each derivation step and since each path that is generated must eventually end, the Turing machine M that explores all derivation paths will eventually halt. If at least one path generates w , M will accept. If no path generates w , M will reject. ■

24.1.3 Equivalence of Linear Bounded Automata and Context-Sensitive Grammars

We now have a new computational model, the LBA, and we have shown that it is decidable whether some LBA B accepts a string w . We also have a new grammatical framework, context-sensitive grammars, and we have shown that it is decidable whether or not a context-sensitive grammar G generates some string w . That similarity, along with the terminology that we have been using, should cause the following theorem to come as no surprise:

Theorem 24.3 Equivalence of LBAs and Context-Sensitive Grammars

Theorem: The class of languages that can be described with a context-sensitive grammar is exactly the same as the class of languages that can be accepted by some LBA. Alternatively, a language is context-sensitive iff it can be generated by some context-sensitive grammar.

Proof: The proof is very similar to the one that we did of Theorem 23.1, which asserted the equivalence of unrestricted grammars and Turing machines. We must do two proofs:

- We show that, given a context-sensitive grammar G , there exists an LBA B such that $L(G) = L(B)$. We do this by construction of B from G . B uses a two-track tape (simulated on one tape). On input w , B keeps w on the first tape. On the second tape, it nondeterministically constructs a derivation using G , but with one exception. Any path that is about to generate a string that is longer than w will halt immediately. So B never needs a tape longer than $|w|$ and is thus an LBA.
- We show that, given an LBA B , there exists a context-sensitive grammar G such that $L(B) = L(G)$. As in the proof of Theorem 23.1, G will simulate the operation of B . The design of G is a bit more complex now because it cannot use working symbols that get erased at the end. However, that problem can be solved with an appropriate encoding of the nonterminal symbols. ■

24.1.4 Where Do Context-Sensitive Languages Fit in the Language Hierarchy?

Our motivation in designing the LBA was to get the best of both worlds:

- something closer to the power of a Turing machine, but with
- the decidability properties of a PDA.

Have we succeeded? Both of the languages $\{ \langle B, w \rangle : \text{LBA } B \text{ accepts } w \}$ and $\{ \langle G, w \rangle : \text{context sensitive grammar } G \text{ generates string } w \}$ are decidable. And we have seen at least one example, $A^n B^n C^n$, of a language that is not context-free but is context-sensitive. In this section, we state and prove two theorems that show that the picture at the beginning of this chapter is correctly drawn.

Theorem 24.4 The Context-Sensitive Languages are a Proper Subset of D

Theorem: The context-sensitive languages are a proper subset of D.

Proof: We divide the proof into two parts. We first show that every context-sensitive language is in D. Then we show that there exists at least one language that is in D but that is not context-sensitive.

The first part is easy. Every context-sensitive language L is accepted by some LBA B . So the Turing machine that simulates B as described in the proof of Theorem 24.1 decides L .

Second, we must prove that there exists at least one language that is in D but that is not context-sensitive. It is not easy to do this by actually exhibiting such a language. But we can use diagonalization to show that one exists.

We consider only languages with $\Sigma = \{a, b\}$. First we must define an enumeration of all the context-sensitive grammars with $\Sigma = \{a, b\}$. To do that, we need an encoding of them. We can use a technique very much like the one we used to encode Turing machines. Specifically, we will encode a grammar $G = (V, \Sigma, R, S)$ as follows:

- Encode the nonterminal alphabet: Let $k = |V - \Sigma|$ be the number of nonterminal symbols in G . Let n be the number of binary digits required to represent the integers 0 to $k - 1$. Encode the set of nonterminal symbols $(V - \Sigma)$ as $x_1d_1d_2\dots d_n$, where each $d_i \in \{0, 1\}$. Let $x_00\dots 0_n$ correspond to S .
- Encode the terminal alphabet, $\{a, b\}$, as a and b .
- Encode each rule $\alpha \rightarrow \beta$ in R as: $A \rightarrow B$, where A is the concatenation of the encodings of all of the symbols of α and B is the concatenation of the encodings of all of the symbols of β . So the encoding of a rule might look like:
 $ax_01b \rightarrow bx_01a$
- Finally, encode G by concatenating together its rules, separated by ;'s. So a complete grammar G might be encoded as:

$$x_00 \rightarrow ax_00a; x_00 \rightarrow x_01; x_01 \rightarrow bx_01b; x_01 \rightarrow b$$

Let $Enum_G$ be the lexicographic enumeration of all encodings, as just described, of context-sensitive grammars with $\Sigma = \{a, b\}$. Let $Enum_{a,b}$ be the lexicographic enumeration of $\{a, b\}^*$. We can now imagine the infinite table shown in Table 24.1. Column 0 contains the elements of $Enum_G$. Row 0 contains the elements of $Enum_{a,b}$. Each other cell, with index (i, j) is 1 if $grammar_i$ generates $string_j$ and 0 otherwise. Because $\{ \langle G, w \rangle : \text{context sensitive grammar } G \text{ generates string } w \}$ is in D, there exists a Turing machine that can compute the values in this table as they are needed.

	String ₁	String ₂	String ₃	String ₄	String ₅	String ₆	...
Grammar ₁	1	0	0	0	0	0	...
Grammar ₂	0	1	0	0	0	0	...
Grammar ₃	1	1	0	0	0	0	...
Grammar ₄	0	0	1	0	0	0	...
Grammar ₅	1	0	1	0	0	0	...
...

Table 24.1 Using diagonalization to show that there exist decidable languages that are not context-sensitive

Now define the language $L_D = \{string_i : string_i \notin L(G_i)\}$. L_D is:

- In D because it is decided by the following Turing machine M :

$M(x) =$

1. Find x in the list $Enum_{a,b}$. Let its index be i . (In other words, column i corresponds to x .)
2. Lookup cell (i, i) in the table.
3. If the value is 0, then x is not in $L(G_i)$ so x is in L_D , so accept.
4. If the value is 1, then x is in $L(G_i)$ so x is not in L_D , so reject.

- Not context-sensitive because it differs, in the case of at least one string, from every language in the table and so is not generated by any context-sensitive grammar. ■

Theorem 24.5 The Context-Free Languages are a Proper Subset of the Context-Sensitive Languages

Theorem: The context-free languages are a proper subset of the context-sensitive languages.

Proof: We know one language, $A^nB^nC^n$, that is context-sensitive but not context-free. So it remains only to show that every context-free language is context-sensitive.

If L is a context-free language then there exists some context-free grammar $G = (V, \Sigma, R, S)$ that generates it. Convert G to Chomsky normal form, producing G' . G' generates $L - \{\epsilon\}$. G' is a context-sensitive grammar because it has no length-reducing rules. If $\epsilon \in L$, then create in G' a new start symbol S' (distinct from any other symbols already in G'), and add the rules $S' \rightarrow \epsilon$ and $S' \rightarrow S$. G' is still a context-sensitive grammar and it generates L . So L is a context-sensitive language. ■

24.1.5 Closure Properties of the Context-Sensitive Languages

The context-sensitive languages exhibit strong closure properties. In order to prove that, it is useful first to prove a normal form theorem for context-sensitive grammars. We will do that here, and then go on to prove a set of closure theorems.

A context-sensitive grammar $G = (V, \Sigma, R, S)$ is in **nonterminal normal form** iff all rules in R are of one of the following two forms:

- $\alpha \rightarrow c$, where α is an element of $(V - \Sigma)$ and $c \in \Sigma$, or
- $\alpha \rightarrow \beta$, where both α and β are elements of $(V - \Sigma)^+$.

In other words, the set of nonterminals includes one for each terminal symbol and it is the job of that nonterminal simply to generate its associated terminal symbol. G does almost all of its work manipulating only nonterminals. At the end, the terminal symbols are generated. Once terminal symbols have been generated, no further rules can apply to them since no rules have any terminals in their left-hand sides.

Theorem 24.6 Nonterminal Normal Form for Context-Sensitive Grammars

Theorem: Given a context-sensitive grammar G , there exists an equivalent nonterminal normal form grammar G' such that $L(G') = L(G)$.

Proof: The proof is by construction. From G we create G' using the algorithm *converttononterminal* defined as follows:

converttononterminal(G : context-sensitive grammar) =

1. Initially, let $G' = G$.
2. For each terminal symbol c in Σ , create a new nonterminal symbol T_c and add to $R_{G'}$ the rule $T_c \rightarrow c$.
3. Modify each of the original rules (not including the ones that were just created) so that every occurrence of a terminal symbol c is replaced by the nonterminal symbol T_c .
4. Return G' .

Note that no length-reducing rules have been introduced, so if G is a context-sensitive grammar, so is G' . ■

We can now state a set of closure theorems. The proofs of two of these theorems will exploit nonterminal normal form as just defined.

Theorem 24.7 Closure under Union

Theorem: The context-sensitive languages are closed under union.

Proof: The proof is by construction of a context-sensitive grammar. The construction is identical to the one we gave in the proof of Theorem 13.5 that the context-free languages are closed under union: If L_1 and L_2 are context-sensitive languages, then there exist context-sensitive grammars $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$ such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$. If necessary, rename the nonterminals of G_1 and G_2 so that the two sets are disjoint and so that neither includes the symbol S . We will build a new grammar G such that $L(G) = L(G_1) \cup L(G_2)$. G will contain all the rules of both G_1 and G_2 . We add to G a new start symbol, S , and two new rules, $S \rightarrow S_1$ and $S \rightarrow S_2$. The two new rules allow G to generate a string if at least one of G_1 or G_2 generates it. So $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}, S)$. Note that no length-reducing rules are introduced, so the grammar that results is a context-sensitive grammar. ■

Theorem 24.8 Closure under Concatenation

Theorem: The context-sensitive languages are closed under concatenation.

Proof: The proof is by construction of a context-sensitive grammar. Again we use the construction of Theorem 13.5: if L_1 and L_2 are context-sensitive languages, then there exist context-sensitive grammars $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$ such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$. If necessary, rename the nonterminals of G_1 and G_2 so that the two sets are disjoint and so that neither includes the symbol S . We will build a new grammar G such that $L(G) = L(G_1) L(G_2)$. G will contain all the rules of both G_1 and G_2 . We add to G a new start symbol, S , and one new rule, $S \rightarrow S_1 S_2$. So $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}, S)$.

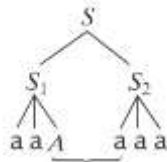


Figure 24.2 Two subtrees may interact

However, now there is one problem that we need to solve: Suppose that one of the original grammars contained a rule with Aa as its left hand side. Figure 24.2 shows a partial parse tree that might be generated by the new grammar. The problem is that Aa can match at the boundary between the substring that was generated from S_1 and the one that was generated from S_2 . That could result in a string that is not the concatenation of a string in L_1 with a string in L_2 . If only nonterminal symbols could occur on the left-hand side of a rule, this problem would be solved by the renaming step that guarantees that the two sets of nonterminals (in the two original grammars) are disjoint. If G were in nonterminal normal form, then that condition would be met.

So, to build a grammar G such that $L(G) = L(G_1) L(G_2)$, we do the following:

1. Convert both G_1 and G_2 to nonterminal normal form.
2. If necessary, rename the nonterminals of G_1 and G_2 so that the two sets are disjoint and so that neither includes the symbol S .
3. $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}, S)$. ■

Theorem 24.9 Closure under Kleene Star

Theorem: The context-sensitive languages are closed under Kleene star.

Proof: The proof is by construction of a context-sensitive grammar. If L_1 is a context-sensitive language, then there exists a context-sensitive grammar $G_1 = (V_1, \Sigma_1, R_1, S_1)$ such that $L_1 = L(G_1)$. To build a grammar G such that $L(G) = L(G_1)^*$, we can use a construction similar to that of Theorem 13.5, in which we create a new start symbol and let it generate zero or more copies of the original start symbol. But now we have two problems. The first is dealing with

ϵ . To solve it, we'll introduce two new symbols, S and T , instead of one. We'll add to the original grammar G_1 a new start symbol S , which will be able to be rewritten as either ϵ or T . T will then be recursive and will be able to generate $L(G_1)^+$. If necessary, rename the nonterminals of G_1 so that V_1 does not include the symbol S or T . G will contain all the rules of G_1 . Then we add to G a new start symbol, S , another new nonterminal T , and four new rules, $S \rightarrow \epsilon$, $S \rightarrow T$, $T \rightarrow T S_1$, and $T \rightarrow S_1$.

But we also run into a problem like the one we just solved above for concatenation. Suppose that the partial tree shown in Figure 24.3 (a) can be created and that there is a rule whose left-hand side is AA . Then that rule could be applied not just to a string that was generated by S_1 but to a string at the boundary between two instances of S_1 . To solve this problem we can again convert the original grammar to nonterminal normal form before we start. But now the two symbols, AA , that are spuriously adjacent to each other were both derived from instances of the same nonterminal (S_1), so creating disjoint sets of nonterminals won't solve the problem. What we have to do this time is to create a copy of the rules that can be used to derive an S_1 . Let those rules derive some new nonterminal S_2 . The two sets of rules will do exactly the same thing but they will exploit disjoint sets of nonterminals as they do so. Then we'll alternate them. So, for example, to generate a string that is the concatenation of four strings from L_1 , we'll create the parse tree shown in Figure 24.3 (b). Now, since neither S_1 nor S_2 can generate ϵ , it can never happen that nonterminals from two separate subtrees rooted by S_1 can be adjacent to each other, nor can it happen from two separate subtrees rooted by S_2 .

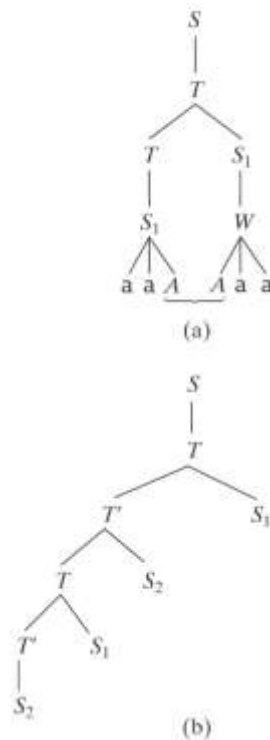


Figure 24.3 Preventing two subtrees from interacting

We can now state the complete construction of a grammar G such that $L(G) = L(G_1)^*$:

1. Convert G_1 to nonterminal normal form.
2. If necessary, rename the nonterminals so that they do not include S , T , T' , or S_2 .
3. Create a new nonterminal S_2 and create copies (with different names) of all the nonterminals and the rules in G_1 so that $L(S_2) = L(S_1)$.

4. $G = (V_1 \cup \{S, T, T'\} \cup \{S_2 \text{ and the other nonterminals generated in step 3}\},$
 $\Sigma_1,$
 $R_1 \cup \{S \rightarrow \varepsilon, S \rightarrow T, T \rightarrow T'S_1, T \rightarrow S_1, T' \rightarrow TS_2, T' \rightarrow S_2\}$
 $\cup \{\text{the rules that derive } S_2, \text{ as generated in step 3}\},$
 $S).$

Theorem 24.10 Closure under Intersection

Theorem: The context-sensitive languages are closed under intersection.

Proof: This time we cannot pattern a proof after one we did for the context-free languages since the context-free languages are not closed under intersection. But we can do a proof by construction of an LBA. If L_1 and L_2 are context-sensitive languages, then there exist LBAs $B_1 = (K_1, \Sigma_1, \Gamma_1, \Delta_1, s_1, H_1)$ and $B_2 = (K_2, \Sigma_2, \Gamma_2, \Delta_2, s_1, H_1)$ such that $L_1 = L(B_1)$ and $L_2 = L(B_2)$. We construct a new LBA B such that $L(B) = L(B_1) \cap L(B_2)$. B will treat its tape as though it were divided into two tracks. It will first copy its input from track 1 to track 2. Then it will simulate B_1 on track 1. If that simulation accepts, then B will simulate B_2 on track 2. If that simulation also accepts, then B will accept. So B will accept iff both B_1 and B_2 do.

Theorem 24.11 Closure under Complement

Theorem: The context-sensitive languages are closed under complement.

Proof: The proof of this claim is based on a complexity argument, so we will delay it until Chapter 29, but see \square .

24.1.6 Decision Procedures for the Context-Sensitive Languages

We have already shown that the membership question for context-sensitive languages is decidable. Unfortunately, it does not appear to be efficiently decidable. Comparing the situation of context-free languages and context-sensitive languages, we have, where w is a string and G a grammar:

- If G is a context-free grammar, then there exists a $\mathcal{O}(n^3)$ algorithm (as we saw in Chapter 15) to decide whether $w \in L(G)$.
- If G is a context-sensitive grammar, then the problem of deciding whether $w \in L(G)$ can be solved by the algorithm that we presented in the proof of Theorem 24.2. It is not certain that no more efficient algorithm exists, but it is known that the decision problem for context-sensitive languages is PSPACE-complete. (We'll define PSPACE-completeness in Chapter 29.) The fact that the problem is PSPACE-complete means that no polynomial-time algorithm exists for it unless there also exist polynomial-time algorithms for large classes of other problems for which no efficient algorithm has yet been found. More precisely, no polynomial-time algorithm for deciding membership in a context-sensitive language exists unless $P = NP = PSPACE$, which is generally thought to be very unlikely.

Because no efficient parsing techniques for the context-sensitive languages are known, practical parsers for programming languages $\text{C } 669$ and natural languages $\text{C } 751$ typically use a context-free grammar core augmented with specific other mechanisms. They do not rely on context-sensitive grammars.

What about other questions we might wish to ask about context-sensitive languages? We list some questions in Table 24.2, and we show their decidability for the context-sensitive languages and also, for comparison, for the context-free languages.

	Decidable for context-free languages?	Decidable for context-sensitive languages?
Is $L = \Sigma^*$?	No	No
Is $L_1 = L_2$?	No (but Yes for deterministic CFLs)	No
Is $L_1 \subseteq L_2$?	No	No
Is L regular?	No	No
Is $\neg L$ also context-free?	No	
Is $\neg L$ also context-sensitive?		Yes, trivially since the context-sensitive languages are closed under complement.
Is $L = \emptyset$?	Yes	No
Is $L_1 \cap L_2 = \emptyset$?	No	No

Table 24.2 Decidability of questions about context-free and context-sensitive languages

We prove two of these claims about the context-sensitive languages here and leave the others as exercises. Since we have shown that context-sensitive grammars and LBAs describe the same class of languages, any question that is undecidable for one will also be undecidable for the other. So we can prove the decidability of a question by using either grammars or machines, whichever is more straightforward. We'll do one example of each.

Theorem 24.12 “Is a Context-Sensitive Language Empty?” is Undecidable

Theorem: The language $L_2 = \{ \langle B \rangle : B \text{ is a LBA and } L(B) = \emptyset \}$ is not in D.

Proof: The proof is by reduction from $H_{\neg \text{ANY}} = \{ \langle M \rangle : \text{there does not exist any string on which Turing machine } M \text{ halts} \}$, which we showed, in Theorem 21.15, is not even in SD. We will define R , a mapping reduction from $H_{\neg \text{ANY}}$ to L_2 . The idea is that R will use the reduction via computation history technique described in Section 22.5.1. Given a particular Turing machine M , it is straightforward to build a new Turing machine $M\#$ that can determine whether a string x is a valid computation history of M . $M\#$ just needs to check four things:

- The string x must be a syntactically legal computation history.
- The first configuration of x must correspond to M being in its start state, with its read/write head positioned just to the left of the input.
- The last configuration of x must be a halting configuration.
- Each configuration after the first must be derivable from the previous one according to the rules in M 's transition relation δ .

In order to check these things, $M\#$ need never move off the part of its tape that contains its input, so $M\#$ is in fact an LBA. Since a computation history must end in a halting state, there will be no valid computation histories for M iff M halts on nothing. So R is defined as follows:

$R(\langle M \rangle) =$

1. Construct the description $\langle M\#(x) \rangle$ of an LBA $M\#(x)$, which operates as follows:
 - 1.1. If x is a valid computation history of M , accept, else reject.
2. Return $\langle M\# \rangle$.

If *Oracle* exists and decides L_2 , then $C = \text{Oracle}(R(\langle M \rangle))$ decides $H_{\neg \text{ANY}}$.

- R can be implemented as a Turing machine.
- C is correct:
 - If $\langle M \rangle \in H_{\neg \text{ANY}}$: There are no valid computation histories of the Turing machine M , so the LBA $M\#$ accepts nothing. $\text{Oracle}(\langle M\# \rangle)$ accepts.
 - If $\langle M \rangle \notin H_{\text{ANY}}$: There is at least one valid computation history of M , so $M\#$ accepts at least one string. $\text{Oracle}(\langle M\# \rangle)$ rejects.

But no machine to decide H_{-ANY} can exist, so neither does *Oracle*. ■

Theorem 24.13 “Is the Intersection of Two Context-Sensitive Languages Empty?” is Undecidable

Theorem: The language $L_2 = \{ \langle G_1, G_2 \rangle : G_1 \text{ and } G_2 \text{ are context-sensitive grammars and } L(G_1) \cap L(G_2) = \emptyset \}$ is not in D.

Proof: The proof is by reduction from $L_1 = \{ \langle G_1, G_2 \rangle : G_1 \text{ and } G_2 \text{ are context-free grammars and } L(G_1) \cap L(G_2) = \emptyset \}$, which we showed, in Theorem 22.9, is not in D. Let R be a mapping reduction from L_1 to L_2 defined as follows:

$R(\langle G_1, G_2 \rangle) =$

1. Using the procedure that was described in the proof of Theorem 24.5, construct from the two context-free grammars G_1 and G_2 , two context-sensitive grammars G_3 and G_4 such that $L(G_3) = L(G_1)$ and $L(G_4) = L(G_2)$.
2. Return $\langle G_3, G_4 \rangle$.

If *Oracle* exists and decides L_2 , then $C = Oracle(R(\langle G_1, G_2 \rangle))$ decides L_1 . But no machine to decide L_1 can exist, so neither does *Oracle*. ■

24.2 The Chomsky Hierarchy

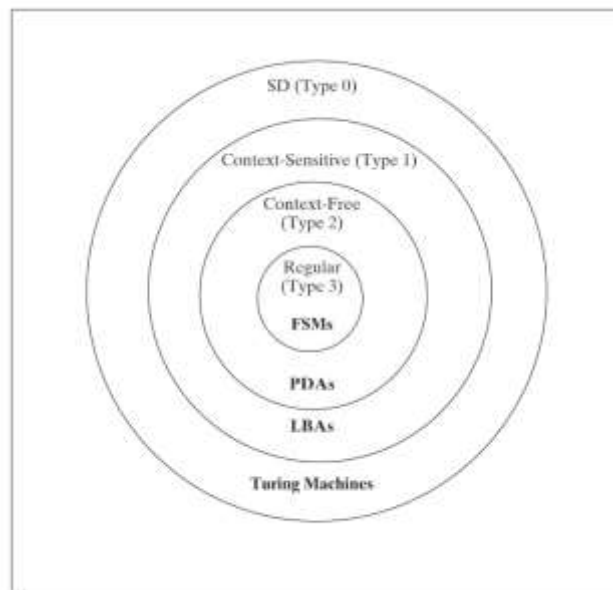


Figure 24.4 The Chomsky hierarchy

In 1956, Noam Chomsky described a slightly different version of the onion diagram that we have been using. Chomsky’s version, commonly called the Chomsky hierarchy, is shown in Figure 24.4. This version is appealing because, for each level, there exists both a grammar formalism and a computational structure. Chomsky used the terms type 0, type 1, type 2, and type 3 to describe the four levels in his model and those terms are still used in some treatments of this topic.

The basis for the Chomsky hierarchy is the amount and organization of the memory required to process the languages at each level:

- **Type 0** (semidecidable): no memory constraint.
- **Type 1** (context-sensitive): memory limited by the length of the input string.
- **Type 2** (context-free): unlimited memory but accessible only in a stack (so only a finite amount is accessible at any point).
- **Type 3** (regular): finite memory.

The Chomsky hierarchy makes an obvious suggestion: Different grammar formalisms offer different descriptive power and may be appropriate for different tasks. In the years since Chomsky published the hierarchy, that idea, coupled with the need to solve real problems, has led to the development of many other formalisms. We will sketch two of them in the rest of this chapter.

24.3 Attribute, Feature, and Unification Grammars

For many applications, context-free grammars are almost, but not quite, good enough. While they may do a good job of describing the primary structure of the strings in a language, they make it difficult, and in some cases impossible, to describe constraints on the way in which sibling constituents may be derived. For example, we saw that no context-free grammar exists for the simple artificial language $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$. The context-free grammar formalism provides no way to express the constraint that the numbers of a's, b's, and c's must be equal. We've seen (in Example 23.1) an unrestricted grammar for $A^n B^n C^n$. But we've also seen that unrestricted grammars are impractical.

What we need is a new technique for describing constraints on sets of constituents. The approach that we describe next treats both terminals and nonterminals not as atomic symbols but rather as clusters of *features* (or *attributes*) and associated values. Then it allows rules to:

- Define ways in which features are passed up and down in parse trees.
- Describe constraints on feature values that must be satisfied before the rules can be applied.

Example 24.2 An Attribute Grammar for $A^n B^n C^n$

We'll show an attribute grammar G for the language $A^n B^n C^n$. G will be a context-free grammar that has been augmented with one feature, *size*. The rules in G will define how *size* is used. Some rules will compute *size* and pass it up, from the terminal nodes, to the root. The single S rule will contain the description of a *size* constraint that must be satisfied before the rule can be applied.

$G = (\{S, A, B, C, a, b, c\}, \{a, b, c\}, R, S)$, where:

$$R = \left\{ \begin{array}{ll} S \rightarrow A B C & (\text{size}(A) = \text{size}(B) = \text{size}(C)) \\ A \rightarrow a & (\text{size}(A) \leftarrow 1) \\ A \rightarrow A_2 a & (\text{size}(A) \leftarrow \text{size}(A_2) + 1) \\ B \rightarrow a & (\text{size}(B) \leftarrow 1) \\ B \rightarrow B_2 a & (\text{size}(B) \leftarrow \text{size}(B_2) + 1) \\ C \rightarrow a & (\text{size}(C) \leftarrow 1) \\ C \rightarrow C_2 a & (\text{size}(C) \leftarrow \text{size}(C_2) + 1) \end{array} \right\}.$$

In this example, each rule has been annotated with an attribute expression. Read the notation A_2 as the name for the daughter constituent, rooted at A , created by the rule that refers to it. This grammar could easily be used by a bottom-up parser that builds the maximal A , B , and C constituents, assigns a size to each, and then attempts to combine the three of them into a single S . The combination will succeed only if all the sizes match.

The fact that it could be useful to augment context-free grammars with various kinds of features and constraints has been observed both by the writers of grammars for programming languages and the writers of grammars of natural languages, such as English. In the programming languages and compilers world, these grammars tend to be called *attribute grammars*. In the linguistics world, they tend to be called *feature grammars* or *unification grammars* (the

latter because of their reliance on a matching process, called unification, that decides when there is a match between features and constraints).

Example 24.3 A Unification Grammar Gets Subject/Verb Agreement Right

In Example 11.6, we presented a simple fragment of an English grammar. That fragment is clearly incomplete; it fails to generate most of the sentences of English. But it also overgenerates. For example, it can generate the following sentence (marked with an * to show that it is ungrammatical):

```
*The bear like chocolate.
```

The problem is that this sentence was generated using the rule $S \rightarrow NP VP$. Because the grammar is context-free, the NP and VP constituents must be realized independently. So there is no way to implement the English constraint that present tense verbs must agree with their subjects in number and gender.

We can solve this problem by replacing the simple nonterminals NP (Noun Phrase) and VP (Verb Phrase) by compound ones that include features corresponding to person and number. One common way to do that is to represent everything, including the primary category, as a feature. So, instead of NP and VP , we might have:

```
[ CATEGORY NP          [ CATEGORY VP
  PERSON THIRD        PERSON THIRD
  NUMBER SINGULAR]    NUMBER SINGULAR]
```



Instead of atomic terminal symbols like `bear`, we might have:

```
[ CATEGORY N
  LEX bear
  PERSON THIRD
  NUMBER SINGULAR]
```

Instead of grammar rules like $S \rightarrow NP VP$, we will now have rules that are stated in terms of feature sets. The idea is that we will write rules that describe constraints on how features must match in order for constituents to be combined. So, for example the $S \rightarrow NP VP$ rule might become:


```
[ CATEGORY S ] → [ CATEGORY NP          [ CATEGORY VP
                  NUMBER  $x_1$           NUMBER  $x_1$ 
                  PERSON  $x_2$  ]       PERSON  $x_2$  ]
```

This rule exploits two variables, x_1 and x_2 , to describe the values of the `NUMBER` and `PERSON` features. Whenever a particular NP is constructed, it will (by a mechanism that we won't go into) acquire values for its `NUMBER` and `PERSON` features from its constituents (usually the head noun, such as `bear`). The same thing will happen for each individual VP . The scope of the variables x_1 and x_2 should be taken to be the entire rule, which will thus be interpreted to say that an NP and a VP can be combined to form an S iff they have matching values for their `NUMBER` and `PERSON` features. We've oversimplified here by suggesting that the only way for values to match is for them to be identical. Practical systems typically exploit a more powerful notion of matching. For example, past tense verbs in English aren't marked for number. So a VP that dominated the verb `shot`, for instance, would have a `NUMBER` value that would enable it to combine with an NP whose `NUMBER` was either `SINGULAR` or `PLURAL`.

Several important natural language grammar formalisms are feature (unification)-based . Grammars written in those formalisms exploit features that describe agreement constraints between subjects and verbs, between nouns and their modifiers, and between verbs and their arguments, to name just a few.  750. They may also use semantic features, both as additional constraints on the way in which sentences can be generated and as the basis for assigning meanings to sentences once they have been parsed.

Both the formal power and the computational efficiency of attribute/feature/unification grammars depend on the details of how features are defined and used. Not all attribute/feature/unification grammar formalisms are stronger than context-free grammars. In particular, consider a formalism that requires that both the number of features and the number of values for each feature must be finite. Then, given any grammar G in that formalism, there exists an equivalent context-free grammar G' . The proof of this claim is straightforward and is left as an exercise. With this restriction then, attribute/feature/unification grammars are simply notational conveniences. In English, there are only two values (singular and plural) for syntactic number and only three values (first, second and third) for person. So the grammar that we showed in Example 24.3 can be rewritten as a (longer and more complex) context-free grammar.

Now consider the grammar that we showed in Example 24.2. The single attribute *size* can take an arbitrary integer value. We know that no context-free equivalent of that grammar exists. When the number of attribute-value pairs is not finite, the power of a grammar formalism depends on the way in which attributes can be computed and evaluated. Some formalisms have the power of Turing machines.

Grammars, augmented with attributes and constraints, can be used in a wide variety of applications. For example, they can describe component libraries and product families .


Particularly in the attribute grammar tradition, it is common to divide attributes into two classes:

- *Synthesized attributes*, which are passed up the parse tree, and
- *Inherited attributes*, which are passed down the tree.

Both of the examples that we have presented use synthesized attributes, which are particularly well-suited to use by bottom-up parsers. Inherited attributes, on the other hand, are well-suited to use by top-down parsers.

One appeal of attribute/feature/unification grammars is that features can be used not just as a way to describe constraints on the strings that can be generated. They may also be used as a way to construct the meanings of strings. Assume that we are working with a language for which a compositional semantic interpretation function (as defined in Section 2.2.6) exists. Then the meaning of anything other than a primitive structure is a function of the meanings of its constituent structures. So, to use attributes as a way to compute meanings for the strings in a language L , we must:

- Create a set of attributes whose values will describe the meanings of the primitives of L . For English, the primitives will typically be words (or possibly smaller units, like morphemes). For programming languages, the primitives will be variables, constants, and the other primitive language constructs.
- Associate with each grammar rule a rule that describes how the meaning attributes of each element of the rule's right hand side should be combined to form the meaning of the left-hand side. For example, the English rule $S \rightarrow NP VP$ can specify that the meaning of an S is structure whose subject is the meaning of the constituent NP and whose predicate is the meaning of the constituent VP .

Attribute grammars for programming languages were introduced as a way to define the semantics of programs that were written in those languages. They can be a useful tool for parser generators .

24.4 Lindenmayer Systems

Lindenmayer systems, or simply L-systems, were first described by Aristid Lindenmayer, a biologist whose goal was to model plant development and growth. L-systems are grammars. They use rules to derive strings. But there are three differences between L-systems and the other grammar formalisms we have discussed so far. These differences arise from the fact that L-systems were designed not to define languages but rather to model ongoing, dynamic processes.

The first difference is in the way in which rules are applied. In all of our other grammar formalisms, rules are applied sequentially. In L-systems, as in the Game of Life and the other cellular automata that we mentioned in Chapter 18, rules are applied, in parallel, to all the symbols in the working string. For example, think of each working string as representing an organism at some time t . At time $t+1$, each of its cells will have changed according to the rules of cell development. Or think of each working string as representing a population at some time t . At time $t+1$, each of the individuals will have matured, died, or reproduced according to the rules of population change.

The second difference is in what it means to generate a string. In all our other grammar formalisms, derivation continues at least until no nonterminal symbols remain in the working string. Only strings that contain no nonterminal symbols are considered to have been generated by the grammar. In L-systems, because we are modeling a process, each of the working strings will be considered to have been generated by the grammar. The distinction between terminals and nonterminals disappears, although there may be some symbols that will be treated as constants (i.e., no rules apply to them).

The third difference is that we will start with an initial string (of one or more symbols), rather than just an initial symbol.

An L-system G is a triple (Σ, R, ω) , where:

- Σ is an alphabet, which may contain a subset C of constants, to which no rules will apply,
- R is a set of rules,
- ω (the start sequence) is an element of Σ^+ .




Each rule in R is of the form: $\alpha A \beta \rightarrow \gamma$, where:

- $A \in \Sigma$. A is the symbol that is to be rewritten by the rule.
- $\alpha, \beta \in \Sigma^*$. α and β describe context that must be present in order for the rule to fire. If they are equal to ϵ , no context is checked.
- $\gamma \in \Sigma^*$. γ is the string that will replace A when the rule fires.

The most straightforward way to describe $L(G)$, the set of strings generated by an L-system G is to specify an interpreter for G . We do that as follows:

L-system-interpret(G : L-system) =

1. Set *working-string* to ω .
2. Do forever:
 - 2.1. Output *working-string*.
 - 2.2. *new-working-string* = ϵ .
 - 2.3. For each character c in *working-string* (moving left to right) do:
 - If possible, choose a rule r whose left-hand side matches c and where c 's neighbors (in *working-string*) satisfy any context constraints included in r .
 - If a rule r was found, concatenate its right-hand side to the right end of *new-working-string*.
 - If none was found, concatenate c to the right end of *new-working-string*.
 - 2.4. *working-string* = *new-working-string*.

In addition to their original purpose, L-systems have been used for applications ranging from composing music  778 to predicting protein folding  to designing buildings .

Because each successive string is built by recursively applying the rules to the symbols in the previous string, the strings that L-systems generate typically exhibit a property called *self-similarity*. We say that an object is self-similar whenever the structure exhibited by its constituent parts is very similar to the structure of the object taken as a whole.

Example 24.4 Fibonacci's Rabbits

Let G be the L-system defined as follows:

$$\begin{aligned}\Sigma &= \{I, M\}. \\ \omega &= I. \\ R &= \{ I \rightarrow M, \\ &\quad M \rightarrow MI \}.\end{aligned}$$

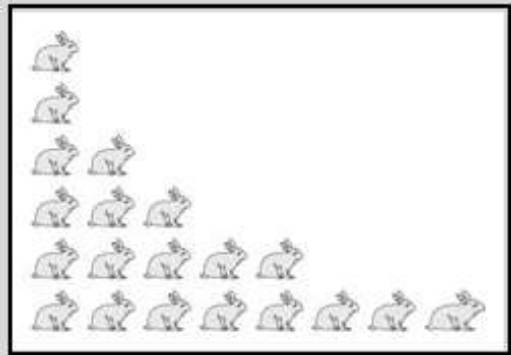
The sequence of strings generated by G begins:

0. I
1. M
2. MI
3. MIM
4. MIIMI
5. MIIMIMIM
6. MIIMIMIMIMIMIM

If we describe each string by its length, then the sequence that G generates is known as the Fibonacci sequence $\{F_n\}$, defined as:

$$\begin{aligned}Fibonacci_0 &= 1. \\ Fibonacci_1 &= 1. \\ \text{For } n > 1, Fibonacci_n &= Fibonacci_{n-1} + Fibonacci_{n-2}.\end{aligned}$$

Fibonacci's goal, in defining the sequence that bears his name, was to model the growth of an idealized rabbit population in which no one dies and each mature pair produces a new male-female pair at each time step. Assume that it takes one time step for each rabbit to reach maturity and mate. Also assume that the gestation period of rabbits is one time step and that we begin with one pair of (immature) rabbits. So at time step 0, there is 1 pair. At time step 1 there is still 1 pair, but they have matured and mated. So, at time step 2, the original pair is alive and has produced one new one. At time step 3, all pairs from time step 2 (of which there are 2) are still alive and all pairs (of which there is just 1) that have been around at least two time steps have produced a new pair. So there are $2 + 1 = 3$ pairs. At time step 4, the 3 pairs from the previous step are still alive and the 2 pairs from two steps ago have reproduced, so there are $3 + 2 = 5$ pairs. And so forth.



Notice that the strings that G produces mirror this structure. Each I corresponds to one immature pair of rabbits and each M corresponds to one mature pair. Each string is the concatenation of its immediately preceding string (the survivors) with the string that preceded it two steps back (the breeders).

Leonardo Pisano Fibonacci lived from 1170 to 1250. Much more recently, the L-system that describes the sequence that bears his name has been used to model things as various as plant structure \mathbb{C} 809, limericks \mathbb{L} and ragtime music \mathbb{L} .

L-systems can be used to model two and three-dimensional structures by assigning appropriate meanings to the symbols that get generated. For example, the turtle geometry system \mathbb{L} provides a set of basic drawing primitives. A turtle program is simply a string of those symbols. So we can use L-systems to generate turtle programs and thus to generate two-dimensional images. Three-dimension structures can be built in a similar way.

Fractals \mathbb{L} are self-similar, recursive structures, so they are easy to generate using L-systems.

Example 24.5 Sierpinski Triangle

Let G be the L-system defined as follows:

$$\Sigma = \{A, B, +, -\}.$$

$$\omega = A.$$

$$R = \{ A \rightarrow B - A - B, \\ B \rightarrow A + B + A \}.$$

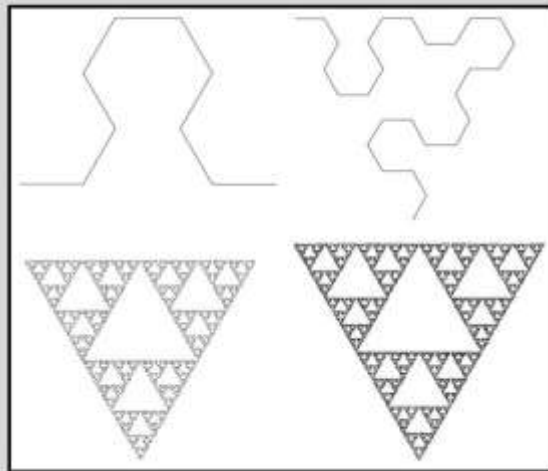
Notice that $+$ and $-$ are constants. No rules transform them so they are simply copied to each successive output string. The sequence of strings generated by G begins:

1. A
2. B - A - B
3. A + B + A - B - A - B - A + B + A
4. B - A - B + A + B + A + B - A - B - A + B + A - B - A - B - A + B + A + B + A + B - A - B

We can interpret these strings as turtle programs by choosing a line length k and then attaching meanings to the symbols in Σ as follows:

- A and B mean move forward, drawing a line of length k .
- $+$ means turn to the left 60° .
- $-$ means turn to the right 60° .

Strings 3, 4, 8, and 10 then correspond to turtle programs that can draw the following sequence of figures (scaling k appropriately):



The limit of this sequence (assuming that an appropriate scaling factor is applied at each step) is the fractal known as the *Sierpinski triangle* .

The growth of many natural structures can most easily be described as the development of branches, which split into new branches, which split into new branches, and so forth. We can model this process with an L-system by introducing into V two new symbols: $[$ will correspond to a push operation and $]$ will correspond to a pop. If we are interpreting the strings the L-system generates as turtle programs, push will push the current pen position onto a stack. Pop will pop off the top pen position, pick up the pen, return it to the position that is on the top of the stack, and then put it down again.

Example 24.6 Trees

Let G be the L-system defined as follows:

$$\Sigma = \{F, +, -, [,]\}.$$

$$\omega = F.$$

$$R = \{F \rightarrow F[-F]F[+F][F]\}.$$

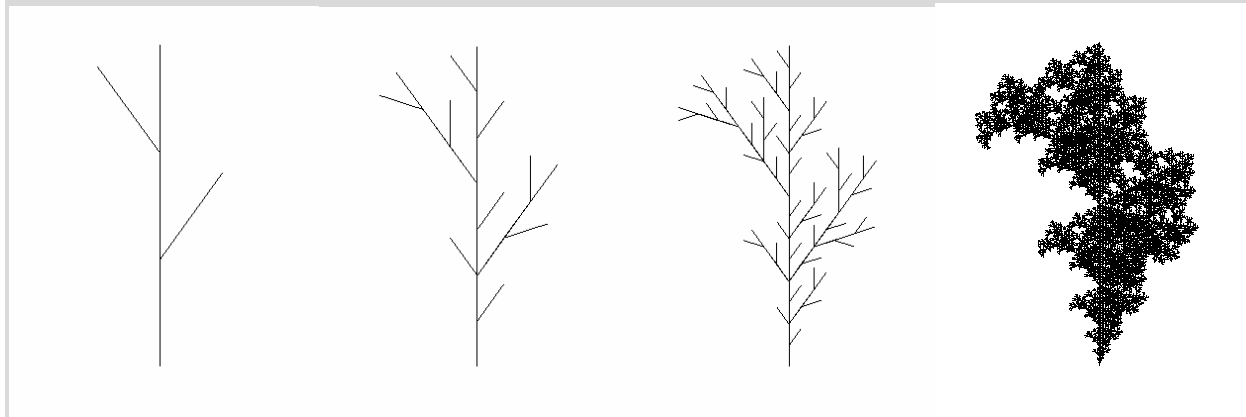
The sequence of strings generated by G begins:

1. F
2. $F[-F]F[+F][F]$
3. $F[-F]F[+F][F][-F[-F]F[+F][F]]F[-F]F[+F][F][+F[-F]F[+F][F]][F[-F]F[+F][F]]$

We can interpret these strings as turtle programs by choosing a line length k and then attaching meanings to the symbols in Σ as follows:

- F means move forward, drawing a line of length k .
- $+$ means turn to the left 36° .
- $-$ means turn to the right 36° .
- $[$ means push the current pen position and direction onto the stack.
- $]$ means pop the top pen position/direction off the stack, lift up the pen, move it to the position that is now on the top of the stack, put it back down, and set its direction to the one on the top of the stack.

Strings 2, 3, 4, and 8 then correspond to turtle programs that can draw the following sequence of figures (scaling k appropriately):



One note about these pictures: The reason that the number of line segments is not consistently a power of 5 is that some lines are drawn on top of others.

Much more realistic trees, as well as other biological structures, can also be described with L-systems. © 809.

So far, all of the L-systems that we have considered are context-free (because we have put no context requirements on the left-hand sides of any of the rules) and deterministic (because there is no more than one rule that matches any symbol). Deterministic, context-free L systems are called *DOL-systems* and are widely used. But we could, for example, give up determinism and allow competing rules with the same left-hand side. In that case, one common way to resolve the competition is to attach probabilities to the rules and thus to build a stochastic L-system.

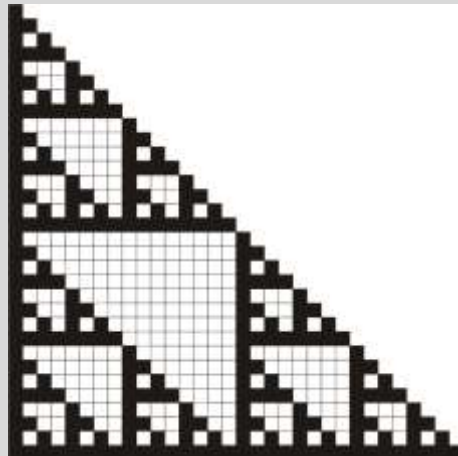
We can also build L-systems that are not context-free. In such systems, the left-hand side of a rule may include contextual constraints. These constraints will be checked before a rule can be applied. But the constraints do not participate in the substitution that the rule performs. Each rule still describes how a single symbol is to be rewritten.

Example 24.7 Sierpinski Triangle, Again

Imagine a one-dimensional cellular automaton. Each cell may contain the value black or white. At any point in time, the automaton consists of a finite number of cells, although it may grow to the right from one step to the next. We will display successive time steps on successive lines, with each cell immediately below its position at the previous time step. With this arrangement, define the *parents* of a cell at time t to be the cell immediately above it (i.e., itself at time $t-1$) and (if it exists) the one that is one row above it but shifted one cell to the left (i.e., its left neighbor at time $t-1$). Now we can state the rule for moving from one time step to the next: At each time step after the first, a cell will exist if it would have at least one parent. And its value will be:

- Black if it has exactly one black parent.
- White otherwise.

Note that, at each time step, one additional cell to the right will have a parent (the cell above and to its left). So the automaton will grow to the right by one black cell at each time step. We will start with the one-cell automaton \blacksquare . After 32 steps, our sequence of automata will draw the following Sierpinski triangle:



We can define G , a context-sensitive L-system to generate this sequence. We'll use the following notation for specifying context in the rules of G : The left-hand side $(a, b, \dots, c) m (x, y, \dots, z)$ will match the symbol m iff the symbol to its left is any of the symbols in the list (a, b, \dots, c) and the symbol to its right is any of the symbols in the list (x, y, \dots, z) . The symbol ε will match iff the corresponding context is empty. (Note that this differs from our usual interpretation of ε , in which it matches everywhere.)

With those conventions, G is:

$$\Sigma = \{\blacksquare, \square\}.$$

$$\omega = \blacksquare.$$

$$R = \{(\varepsilon \mid \square) \blacksquare (\varepsilon) \rightarrow \blacksquare \blacksquare, \quad /* This square is black with no black one to the left, so at $t+1$ there's exactly one black parent. The new cell is black. And there's no cell to the right, so add one, which also has one black parent so it too is black.$$

$$(\varepsilon \mid \square) \blacksquare (\blacksquare \mid \square) \rightarrow \blacksquare, \quad /* This square is black and no black one to the left, so at $t+1$ there's exactly one black parent. The new cell is black.$$

$$(\blacksquare) \blacksquare (\varepsilon) \rightarrow \square \blacksquare, \quad /* Black, plus black to the left. Two black parents. New cell is white. No cell to the right, so add one. It has one black parent so it is black.$$

```

(■) ■ (■ | □) → □, /* Two black parents. New one is white.
(ε | □) □ (ε) → □ ■, /* Two white parents. New one is white. Add cell to right.
(ε | □) □ (■ | □) → □, } /* Two white parents. New one is white.
(■) □ (ε) → ■ ■, /* One black parent. New one is white. Add cell to right.
(■) □ (■ | □) → ■ }. /* One black parent. New one is black.

```

G generates a Sierpinski triangle point wise, while the L-system we described in Example 24.5 generates one by drawing lines.

Context-free L-systems do not have the power of Turing machines. But, if context is allowed, L-systems are equivalent in power to Turing machines. So we can state the following theorem:

Theorem 24.14 Context-Sensitive L-Systems are Turing Equivalent

Theorem: The computation of any context-sensitive L-system can be simulated by some Turing machine. And the computation of any Turing machine can be simulated by some deterministic, context-sensitive L-system.

Proof: The computation of any L-system can be simulated by a Turing machine that implements the algorithm *L-system-interpret*. So it remains to show the other direction.

The proof that the execution of any Turing machine can be simulated by some deterministic, context-sensitive L-system is by construction. More precisely, we'll show that Turing machine M , on input w , halts in some halting state q and with tape contents v iff L-system L converges to the static string qv .

If M is not deterministic, create an equivalent deterministic machine and proceed with it. Then, given M and w , define L as follows:

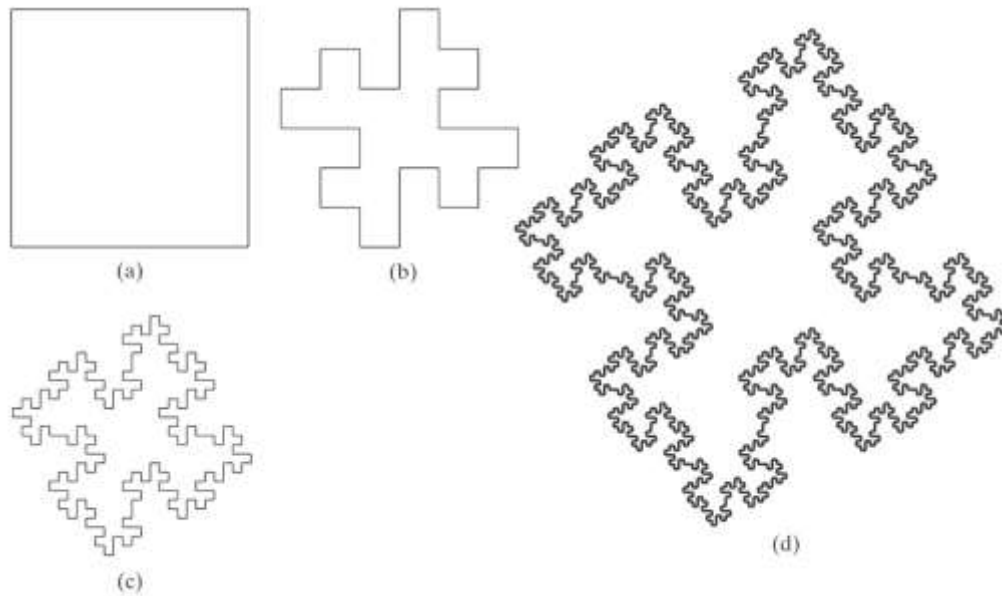
- Let Σ_L be Σ_M , augmented as follows:
 - Add the symbol 0 to encode M 's start state.
 - If M has the halting state y , add the symbol γ to encode it.
 - If M has the halting state n , add the symbol n to encode it.
 - If M has any other halting states, add the symbol h to encode all of them.
 - Add one distinct symbol for each nonhalting state of M .
- Let ω (L 's start string) encode M 's initial configuration. Configurations will be encoded by a string that represents M 's active tape, plus two blank squares on each end. The symbol that represents M 's current state will be inserted into the string immediately to the left of the tape symbol that is under the read/write head. We will follow our usual convention that, just before it starts, M 's read/write is on the blank square just to the left of the first input character. So $\omega = \square\square 0 \square w \square\square$.
- Let the rules R of L encode M 's transitions. To do this, we exploit the fact that the action of a Turing machine is very local. Things only change near the read/write head. So, letting integers correspond to states, suppose that the working string of L is $ga4bcde$. This encodes a configuration in which M 's read/write head is on the b and M is in state 4. The read/write head can move one square to the left or one square to the right. Whichever way it moves, the character under it can change. So, if it moves left, the a changes to some state symbol, the 4 changes to an a , and the b changes to whatever it gets rewritten as. If, on the other hand, the read/write head moves to the right, the 4 changes to whatever the b gets rewritten as and the b gets rewritten as the new state symbol. To decide how to rewrite some character in the working string, it is sufficient to look at one character to its left and two to its right. If there is no state symbol in that area, the symbol gets rewritten as itself. No rule need be specified to make this happen. For all the combinations that do involve a state symbol, we add to R rules that cause the system to behave as M behaves. Finally, add rules so that, if h , γ , or n is ever generated, it will be pushed all the way to the left, leaving the rest of the string unchanged. Add no other rules to R (and in particular no other rules involving any of the halting state symbols).

L will converge to qv iff M halts, in state q , with v on its tape. ■

24.5 Exercises

- 1) Write context-sensitive grammars for each of the following languages L . The challenge is that, unlike with an unrestricted grammar, it is not possible to erase working symbols.
 - a) $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$.
 - b) $WW = \{ww : w \in \{a, b\}^*\}$.
 - c) $\{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w)\}$.
- 2) Prove that each of the following languages is context-sensitive:
 - a) $\{a^n : n \text{ is prime}\}$.
 - b) $\{a^{n^2} : n \geq 0\}$.
 - c) $\{xwx^R : x, w \in \{a, b\}^+ \text{ and } |x| = |w|\}$.
- 3) Prove that every context-free language is accepted by some deterministic LBA.
- 4) Recall the diagonalization proof that we used in the proof of Theorem 24.5, which tells us that the context-sensitive languages are a proper subset of D. Why cannot that same proof technique be used to show that there exists a decidable language that is not decidable or an SD language that is not decidable?
- 5) Prove that the context-sensitive languages are closed under *reverse*.
- 6) Prove that each of the following questions is undecidable:
 - a) Given a context-sensitive language L , is $L = \Sigma^*$?
 - b) Given a context-sensitive language L , is L finite?
 - c) Given two context-sensitive languages L_1 and L_2 , is $L_1 = L_2$?
 - d) Given two context-sensitive languages L_1 and L_2 , is $L_1 \subseteq L_2$?
 - e) Given a context-sensitive language L , is L regular?
- 7) Prove the following claim, made in Section 24.3: Given an attribute/feature/unification grammar formalism that requires that both the number of features and the number of values for each feature must be finite and a grammar G in that formalism, there exists an equivalent context-free grammar G' .

8) The following sequence of figures corresponds to a fractal called a *Koch island*:



These figures were drawn by interpreting strings as turtle programs, just as we did in Example 24.5 and Example 24.6. The strings were generated by an L-system G , defined with:

$$\Sigma = \{F, +, -\}.$$

$$\omega = F - F - F - F.$$

To interpret the strings as turtle programs, attach meanings to the symbols in Σ as follows (assuming that some value for k has been chosen):

- F means move forward, drawing a line of length k .
- $+$ means turn left 90° .
- $-$ means turn right 90° .

Figure (a) was drawn by the first generation string ω . Figure (b) was drawn by the second generation string, and so forth. R_G contains a single rule. What is it?

25 Computable Functions

In almost all of our discussion so far, we have focused on exactly one kind of problem: deciding a language. We saw in Chapter 2 that other kinds of problems can be recast as language-decision problems and so can be analyzed within the framework that we have described. But, having introduced the Turing machine, we now also have a way to analyze programs that compute functions whose range is something other than $\{Accept, Reject\}$.

25.1 What is a Computable Function?

Informally, a function is computable iff there exists a Turing machine that can compute it. In this section we will formalize that notion.

25.1.1 Total and Partial Functions

We begin by considering two classes of functions. Let f be an arbitrary function. Then:

- f is a **total function** on the domain Dom iff f is defined on all elements of Dom . This is the standard mathematical definition of a function on a domain.
- f is a **partial function** on the domain Dom iff f is defined on zero or more elements of Dom . This definition allows for the existence of elements of the domain on which the function is not defined.

Example 25.1 Total and Partial Functions

- Consider the successor function $succ$ and the domain \mathbb{N} (the natural numbers). $Succ$ is a total function on \mathbb{N} . It is also a partial function on \mathbb{N} .
- Consider the simple string function $midchar$, which returns the middle character of its argument string if there is one. The $midchar$ function is a partial function on the domain of strings. But it is not a total function on the domain of strings, since it is undefined for strings of even length. It is, however, a total function on the smaller domain of odd length strings.
- Consider the function $steps$, defined on inputs of the form $\langle M, w \rangle$. It returns the number of steps that Turing machine M executes, on input w , before it halts. The $steps$ function is a partial function on the domain $\{\langle M, w \rangle\}$. But it is not a total function on that domain, since it is undefined for values of $\langle M, w \rangle$ where M does not halt on w . It is, however, a total function on the smaller domain $\{\langle M, w \rangle: \text{Turing machine } M \text{ halts on input } w\}$.

Why do we want to expand the notion of a function to allow for partial functions? A cleaner approach is simply to narrow the domain so that it includes only values on which the function is defined. So, for example, in the case of the $midchar$ function, we simply assert that its domain is the set of odd length strings. Then we have a total function and thus a function in the standard mathematical sense. Of course we can do the same thing with the function $steps$: we can refine its domain to include only values on which it is defined. But now we face an important problem given that our task is to write programs (more specifically, to design Turing machines) that can compute functions. The set of values on which $steps$ is defined is the language H . And H is not in D (i.e., it is not a decidable set). So, no matter what Turing machine we might build to compute $steps$, there exists no other Turing machine that can examine a value and decide whether the $steps$ machine should be able to run. Another way to think of this problem is that it is impossible for any implementation of $steps$ to check its precondition. The only way it is going to be possible to build an implementation of $steps$ is going to be to define its domain as some decidable set and then allow that there are elements of that domain for which $steps$ will not return a value. Thus $steps$ will be a partial and not a total function of the domain on which the program that implements it runs. So any such program will fail to halt on some inputs.

25.1.2 Partially Computable and Computable Functions

Recall that, in Section 17.2.2, we introduced the notion of a Turing machine that computes an arbitrary function. In the rest of this section we will expand on the ideas that we sketched there. In particular, we will now consider functions, like $midchar$ and $steps$, that are not defined on all elements of Σ^* .

We begin by restating the basic definitions that we gave in Section 17.2.2:

- Let M be a Turing machine with start state s , halting state h , input alphabet Σ , and tape alphabet Γ . The initial configuration of M will be $(s, \sqcup w)$, where $w \in \Sigma^*$.
- Define $M(w) = z$ iff $(s, \sqcup w) \vdash_{M^*} (h, \sqcup z)$. In other words $M(w) = z$ iff M , when started on a string w in Σ^* , halts with z on its tape and its read/write head is just to the left of z .
- We say that a Turing machine M **computes** a function f iff, for all $w \in \Sigma^*$:
 - If w is an input on which f is defined, $M(w) = f(w)$. In other words, M halts with $f(w)$ on its tape.
 - Otherwise $M(w)$ does not halt.
- A function f is **recursive** or **computable** iff there is a Turing machine M that computes it and that always halts.

But what about functions that are not defined on all elements of Σ^* . They are not computable under this definition. Let f be any function defined on some subset of Σ^* . Then f is **partially computable** iff there exists a Turing machine M that computes it. In other words, M halts and returns the correct value for all inputs on which f is defined. On all other inputs, M fails to halt.

Let f be any partially computable function whose domain is only a proper subset of Σ^* . Then any Turing machine that computes f will fail to halt on some inputs. But now consider only those functions f such that the set of values Dom on which f is defined is decidable. In other words, f is a total function on the decidable set Dom . For example, $midchar$ is such a function, defined on the decidable set of odd length strings. For any such function f , we define a new function f' that is identical to f except that its range includes one new value, which we will call *Error*. On any input z on which f is undefined, $f'(z) = Error$. Given a Turing machine M that computes f , we can construct a new Turing machine M' that computes f' and that always halts. Let Dom be the set of values on which f is defined. Since Dom is in D , there is some Turing machine TF that decides it. Then the following Turing machine M' computes f' :

$M'(x) =$

1. Run TF on x .
2. If it rejects, output *Error*.
3. If it accepts, run M on x .

We have simply put a wrapper around M . The job of the wrapper is to check M 's precondition and only run M when its precondition is satisfied. This is the technique we use all the time with real programs.

Using the wrapper idea we can now offer a broader and more useful definition of computability:

Let f be a function whose domain is some subset of Σ^* . Then f is **computable** iff there exists a Turing machine M that computes f' (as described above) and that halts on all inputs. Equivalently, f is computable iff it is partially computable and its domain is a decidable set.

Now suppose that f is a function whose domain and/or range is not a set of strings. For example, both the domain and the range of the successor function $succ$ are the integers. Then f is computable iff all of the following conditions hold:

- There exist alphabets Σ and Σ' .
- There exists an encoding of the elements of the domain of f as strings in Σ^* .
- There exists an encoding of the elements of the range of f as strings in Σ'^* .
- There exists some computable function f' with the property that, for every $w \in \Sigma^*$:
 - If $w = \langle x \rangle$ and x is an element of f 's domain, then $f'(w) = \langle f(x) \rangle$, and
 - If w is not the encoding of any element of f 's domain (either because it is not syntactically well formed or because it encodes some value on which f is undefined), then $f'(w) = Error$.

Example 25.2 The Successor Function *succ*

Consider again the successor function:

$$\begin{aligned} \text{succ}: \mathbb{N} &\rightarrow \mathbb{N}, \\ \text{succ}(x) &= x + 1. \end{aligned}$$

We can encode both the domain and the range of *succ* in unary (i.e., as strings drawn from $\{1\}^*$). Then we can define the following Turing machine *M* to compute it:

$$M(x) = \begin{aligned} & \\ & 1. \text{ Write } 1. \\ & 2. \text{ Move left once.} \\ & 3. \text{ Halt.} \end{aligned}$$

The function *succ* is a total function on \mathbb{N} . Every element of $\Sigma^* = \{1\}^*$ is the encoding of some element of \mathbb{N} . For each such element *x*, *M* computes *f(x)* and halts. So *succ* is computable.

Example 25.3 The Function *midchar*

Consider again the function *midchar* that we introduced in Example 25.1. Recall that *midchar* is a total function on the set of odd length strings and a partial function on the set of strings. Now we want to build a Turing machine *M* to compute *midchar*.

The most straightforward way to encode a string *x* as input to *M* is as itself. If we do that, then we can build a straightforward Turing machine *M*:

$$M(x) = \begin{aligned} & \\ & 1. \text{ If the length of } x \text{ is odd, compute } \text{midchar}(x). \\ & 2. \text{ If the length of } x \text{ is even, then what? By the definition of a machine that computes a function } f, M \text{ should} \\ & \quad \text{loop on all values for which } f \text{ is not defined. So it must loop on all even length inputs.} \end{aligned}$$

The existence of *M* proves that *midchar* is partially computable. But *midchar* is also computable because the following Turing machine *M'*, which halts on all inputs, computes *midchar*:

$$M'(x) = \begin{aligned} & \\ & 1. \text{ If the length of } x \text{ is even, output } \textit{Error}. \\ & 2. \text{ Otherwise, find the middle character of } x \text{ and output it.} \end{aligned}$$

Example 25.4 The Function *steps*

Consider again the function *steps* that we introduced in Example 25.1. Recall that *steps* is a total function on the set $\langle M, w \rangle$: Turing machine *M* halts on input *w*. It is a partial function on the set $\{ \langle M, w \rangle \}$. And it is also a partial function on the larger set of strings that includes syntactically ill-formed inputs. *Steps* is a partially computable function because the following three-tape Turing machine *S* computes it:

$$S(x) = \begin{aligned} & \\ & 1. \text{ If } x \text{ is not a syntactically well formed } \langle M, w \rangle \text{ string then loop.} \\ & 2. \text{ If } x \text{ is a syntactically well formed } \langle M, w \rangle \text{ string then:} \\ & \quad 2.1. \text{ Copy } M \text{ to tape 3.} \\ & \quad 2.2. \text{ Copy } w \text{ to tape 2.} \\ & \quad 2.3. \text{ Write 0 on tape 1.} \\ & \quad 2.4. \text{ Simulate } M \text{ on } w \text{ on tape 2, keeping a count on tape 1 of each step that } M \text{ makes.} \end{aligned}$$

S halts whenever its input is well-formed and M halts on w . If it halts, it has the value of $steps(\langle M, w \rangle)$ on tape 1. By Theorem 17.1, there exists a one-tape Turing machine S' whose output is identical to the value that S placed on tape 1. So S' is a standard Turing machine that computes $steps$. The existence of S' proves that $steps$ is partially computable.

But $steps$ is not computable. We show that it is not by showing that there exists no Turing machine that computes the function $steps'$, defined as:

$steps'(x) =$ If x is not a syntactically well-formed $\langle M, w \rangle$ string, then *Error*.
 If x is well-formed but $steps(\langle M, w \rangle)$ is undefined (i.e., M does not halt on w), then *Error*.
 If $steps(\langle M, w \rangle)$ is defined (i.e., M halts on w), then $steps(\langle M, w \rangle)$.

We prove that no such Turing machine exists by reduction from H . Suppose that there did exist such a machine. Call it ST . Then the following Turing machine DH would decide the language $H = \{\langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w\}$:

$DH(\langle M, w \rangle) =$
 1. Run $ST(\langle M, w \rangle)$.
 2. If the result is *Error* then reject. Else accept.

But we know that there can exist no Turing machine to decide H . So ST must not exist. So $steps$ is not computable.

25.1.3 Functions That Are Not Partially Computable

There exist functions like *succ* and *midchar* that are computable. There exist functions like *steps* that are partially computable but not computable. But there also exist functions that are not even partially computable.

Theorem 25.1 There Exist Functions that are Not Even Partially Computable

Theorem: There exist (a very large number of) functions that are not partially computable.

Proof: We will use a counting argument similar to the one we used to prove a similar result, Theorem 20.3, which says that there exist languages that are not semidecidable. We will consider only unary functions from some subset of \mathbb{N} (the nonnegative integers) to \mathbb{N} . Call the set of all such functions U . We will encode both the input to functions in U and their outputs as binary strings.

Lemma: There is a countably infinite number of partially computable functions in U .

Proof of Lemma: Every partially computable function in U is computed by some Turing machine M with Σ and Γ equal to $\{0, 1\}$. By Theorem 17.7, there exists an infinite lexicographic enumeration of all such syntactically legal Turing machines. So, by Theorem 32.1, there is a countably infinite number of Turing machines that compute functions in U . There cannot be more partially computable functions than there are Turing machines, so there is at most a countably infinite number of partially computable functions in U . There is not a one-to-one correspondence between partially computable functions and the Turing machines that compute them since there is an infinite number of Turing machines that compute any given function. But the number of partially computable functions must be infinite because it includes all the constant functions (which are also computable):

$$cf_1(x) = 1, cf_2(x) = 2, cf_3(x) = 3, \dots$$

So there is a countably infinite number of partially computable functions in U .

Lemma: There is an uncountably infinite number of functions in U .

Proof of Lemma: For any element s in $\mathcal{P}(\mathbb{N})$ (the power set of \mathbb{N}), let f_s be the characteristic function of s . So $f_s(x) = 1$ if $x \in s$ and 0 otherwise. No two elements of $\mathcal{P}(\mathbb{N})$ have the same characteristic function. By Theorem 32.4, there

is an uncountably infinite number of elements in $\mathcal{P}(\mathbb{N})$, so there is an uncountably infinite number of such characteristic functions, each of which is in U .

Proof of Theorem: Since there is only a countably infinite number of partially computable functions in U and an uncountably infinite number of functions in U , there is an uncountably infinite number of functions in U that are not partially computable. ■

Now we know that there exist many functions that are not partially computable. But can we describe one? The answer is yes. One way to do so is by diagonalization. Let E be a lexicographic enumeration of the Turing machines that compute the partially computable functions in U . Let M_i be the i^{th} machine in that enumeration. Define a new function $notcomp(x)$ as follows:

$$\begin{aligned} notcomp: \mathbb{N} &\rightarrow \{0, 1\}, \\ notcomp(x) &= 1 \text{ if } M_x(x) = 0, 0 \text{ otherwise.} \end{aligned}$$

So $notcomp(x) = 0$ if either $M_i(x)$ is defined and the value is something other than 0 or if $M_i(x)$ is not defined. This new function $notcomp$ is in U , but it differs, in at least one place, from every function that is computed by a Turing machine whose encoding is listed in E . So there is no Turing machine that computes it. Thus it is not partially computable.

25.1.4 The Busy Beaver Functions

There exist even more straightforward total functions that are not partially computable. One well known example is a family of functions called **busy beaver functions** □. To define two of these functions, consider the set T of all standard Turing machines M (i.e., deterministic, one-tape machines of the sort defined in Section 17.1), where M has tape alphabet $\Gamma = \{\square, 1\}$ and M halts on a blank tape. Then:

- $S(n)$ is defined by considering all machines that are in T and that have n nonhalting states. The value of $S(n)$ is the maximum number of steps that are executed by any such n -state machine, when started on a blank tape, before it halts.
- $\Sigma(n)$ is defined by again considering all machines that are in T and that have n nonhalting states. The value of $\Sigma(n)$ is the maximum number of 1's that are left on the tape by any such n -state machine, when started on a blank tape, when it halts.

A variety of other busy beaver functions have also been defined. Some of them allow three or more tape symbols (instead of the two we allow). Some use variants of our Turing machine definition. For example, our versions are called quintuple versions, since our Turing machines both write and move the read/write head at each step (so each element of the transition function is a quintuple). One common variant allows machines to write or to move, but not both, at each step (so each element of the transition function is a quadruple). Quadruple machines typically require more steps than quintuple machines require to perform the same task.

All of the busy beaver functions provide a measure of how much work a Turing machine with n states can do before it halts. And none of them is computable. In a nutshell, the reason is that their values grow too fast, as can be seen from Table 25.1, which summarizes some of what is known about the values of S and Σ , as we defined them above. For values of n greater than 4 (in the case of S) or 5 in the case of Σ , the actual values are not known but lower bounds on them are, as shown in the table. For the latest results in determining these bounds, see □.

n	$S(n)$	$\Sigma(n)$
1	1	1
2	6	4
3	21	6
4	107	13
5	$\geq 47,176,870$	4098
6	$\geq 3 \cdot 10^{1730}$	$\geq 1.29 \cdot 10^{865}$

Table 25.1 Some values for the busy beaver functions

Theorem 25.2 S and Σ are Total Functions

Theorem: Both S and Σ are total functions on the positive integers.

Proof: For any value n , both $S(n)$ and $\Sigma(n)$ are defined iff there exists some standard Turing machine M , with tape alphabet $\Gamma = \{\square, 1\}$, where:

- M has n nonhalting states, and
- M halts on a blank tape.

We show by construction that such a Turing machine M exists for every integer value of $n \geq 1$. We will name the nonhalting states of M with the integers $1, \dots, n$. We can build M as follows:

1. Let state 1 be the start state of M .
2. For all i such that $1 < i \leq n$, add to δ_M the transition $((i - 1, \square), (i, \square, \rightarrow))$.
3. Let M have a single halting state called h .
4. Add to δ_M the transition $((n, \square), (h, \square, \rightarrow))$.

M is a standard Turing machine with tape alphabet $\Gamma = \{\square, 1\}$, it has n nonhalting states, and it halts on a blank tape. It is shown in Figure 25.1:



Figure 25.1 A halting Turing machine with n nonhalting states

■

So both S and Σ are defined on all positive integers. If they are not computable, it is not because their domains are not in D . But they are not computable. We first prove a lemma and then use it to show that both busy beaver functions, S and Σ , are not computable.

Theorem 25.3 The Busy Beaver Functions are Strictly Monotonically Increasing

Theorem: Both S and Σ are strictly monotonically increasing functions. In other words:

$$\begin{array}{lll}
 S(n) < S(m) & \text{iff} & n < m, \text{ and} \\
 \Sigma(n) < \Sigma(m) & \text{iff} & n < m.
 \end{array}$$

Proof: We must prove four claims:

- $n < m \rightarrow S(n) < S(m)$: let $S(n) = k$. Then there exists an n -state Turing machine TN that runs for k steps and then halts. From TN we can build an m -state Turing machine TM that runs for $k + (m - n)$ steps and then halts. We add

$m - n$ states to TN . Let any state that was a halting state of TN cease to be a halting state. Instead, make it go, on any input character, to the first new state, write a 1, and move right. From that first new state, go, on any input character, to the second new state, write a 1, and move right. Continue through all the new states. Make the last one a halting state. This new machine executes k steps, just as TN did, and then an additional $m - n$ steps. Then it halts. So $S(m) \geq S(n) + (m - n)$. Since $m > n$, $(m - n)$ is positive. So $S(m) > S(n)$.

- $S(n) < S(m) \rightarrow n < m$: We can rewrite this as $\neg(n < m) \rightarrow \neg(S(n) < S(m))$ and then as $n \geq m \rightarrow S(n) \geq S(m)$. If $n = m$, then $S(m) = S(n)$. If $n > m$, then by the first claim, proved above, $S(n) > S(m)$.
- $n < m \rightarrow \Sigma(n) < \Sigma(m)$: Analogously to the proof that $n < m \rightarrow S(n) < S(m)$ but substitute Σ for S .
- $\Sigma(n) < \Sigma(m) \rightarrow n < m$: Analogously to the proof that $S(n) < S(m) \rightarrow n < m$ but substitute Σ for S . ■

Theorem 25.4 The Busy Beaver Functions are Not Computable

Theorem: Neither S nor Σ is computable.

Proof: We will prove that S is not computable. We leave the proof of Σ as an exercise.

Suppose that S were computable. Then there would be some Turing machine BB , with some number of states that we can call b , that computes it. For any positive integer n , we can define a Turing machine $Write_n$ that writes n 1's on its tape, one at a time, moving rightwards, and then halts with its read/write head on the blank square immediately to the right of the rightmost 1. $Write_n$ has n nonhalting states plus one halting state. We can also define a Turing machine $Multiply$ that multiplies two unary numbers, written on its tape and separated by the ; symbol. The design of $Multiply$ was an exercise in Chapter 17. Let m be the number of states in $Multiply$.

Using the macro notation we described in Section 17.1.5, we can define, for any positive integer n , the following Turing machine, which we can call $Trouble_n$:

$$>Write_n ; R Write_n L_{\square} Multiply L_{\square} BB$$

$Trouble_n$ first writes a string of the form $1^n;1^n$. It then moves its read/write head back to the left so that it is on the blank square immediately to the left of that string. It invokes $Multiply$, which results in the tape containing a string of exactly n^2 1's. It moves its read/write head back to the left and then invokes BB , which outputs $S(n^2)$. The number of states in $Trouble_n$ is shown in Table 25.2.

Component	Number of States
$Write_n$	$n + 1$
; R	1
$Write_n$	$n + 1$
L_{\square}	2
$Multiply$	m
L_{\square}	2
BB	b
Total	$2n + m + b + 7$

Table 25.2 The number of states in $Trouble_n$

Since BB , the final step of $Trouble_n$, writes a string of length $S(n^2)$ and it can write only one character per step, $Trouble_n$ must run for at least $S(n^2)$ steps. Since, for any $n > 0$, $Trouble_n$ is a Turing machine with $2n + m + b + 7$ states that runs for at least $S(n^2)$ steps, we know that:

$$S(2n + m + b + 7) \geq S(n^2).$$

By Theorem 25.3, we know that S is monotonically increasing, so it must also be true that, for any $n > 0$:

$$2n + m + b + 7 \geq n^2.$$

But, since n^2 grows faster than n does, that cannot be true. In assuming that BB exists, we have derived a contradiction. So BB does not exist. So S is not computable. ■

25.1.5 Languages and Functions

It should be clear by now that there is a natural correspondence between languages, which may be in D , SD/D , or $\neg SD$, and functions, which may be computable, partially computable, or neither. We can construct Table 25.3, which show us three ways to present a computational problem:

The Problem View	The Language View	The Functional View
Given three natural numbers, x , y , and z , is $z = x \cdot y$?	$\{ \langle x \rangle * \langle y \rangle = \langle z \rangle : x, y, z \in \{0, 1\}^* \text{ and } num(x) \cdot num(y) = num(z) \}$. <i>D</i>	$f: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, $f(x, y) = x \cdot y$. <i>computable</i>
Given a Turing machine M , does M have an even number of states?	$\{ \langle M \rangle : \text{TM } M \text{ has an even number of states} \}$. <i>D</i>	$f: \{ \langle M \rangle \} \rightarrow \text{Boolean}$, $f(\langle M \rangle) = \text{True}$ if TM M has an even number of states, False otherwise. <i>computable</i>
Given a Turing machine M and a string w , does M halt on w in n steps?	$\{ \langle M, w, n \rangle : \text{TM } M \text{ halts on } w \text{ in } n \text{ steps} \}$. <i>SD/D</i>	$f: \{ \langle M, w \rangle \} \rightarrow \mathbb{N}$, $f(\langle M, w \rangle) =$ if TM M halts on w then the number of steps it executes before halting, else undefined. <i>partially computable</i>
Given a Turing machine M , does M halt on all strings in no more than n steps?	$\{ \langle M, n \rangle : \text{TM } M \text{ halts on each element of } \Sigma^* \text{ in no more than } n \text{ steps} \}$. <i>\neg SD</i>	$f: \{ \langle M \rangle \} \rightarrow \mathbb{N}$, $f(\langle M \rangle) =$ if TM M halts on all strings then the maximum number of steps it executes before halting else undefined. <i>not partially computable</i>

Table 25.3 The problem, language, and functional views

25.2 Recursive Function Theory

We have been using the terms:

- *decidable*, to describe languages that can be decided by some Turing machine,
- *semidecidable*, to describe languages that can be semidecided by some Turing machine,

- *partially computable*, to describe functions that can be computed by some Turing machine, and
- *computable*, to describe functions that can be computed by some Turing machine that halts on all inputs.

The more traditional terminology is:

- *recursive* for *decidable*.
- *recursively enumerable* for *semidecidable*. The recursively enumerable languages are often called just the *RE* or *r.e.* languages.
- *partial recursive* for *partially computable*.
- *recursive* for *computable*.

Before we continue, we need to issue one warning about the fact that there is no standard definition for some of these terms. The terms *computable* and *recursive* are used in some discussions, including this one, to refer just to functions that can be computed by a Turing machine that always halts. In some other discussions, they are used to refer to the class we have called the *partial recursive* or the *partially computable* functions.

Why are the computable functions traditionally called *recursive*? The word makes sense if you think of *recursive* as a synonym for *computable*. In this section, we will see why *recursive* is a reasonable synonym for *computable*. In the rest of this section, to be compatible with conventional treatments of this subject, we will use the term *recursive function* to mean *computable function*.

A ***recursive function*** is one that can be computed by a Turing machine that halts on all inputs. A ***partial recursive function*** is one that can be computed by some Turing machine (but one that may loop if there are any inputs on which the function is undefined). So we have definitions, stated in terms of a computational framework, for two important classes of functions. Let's now ask a different question: Are there definitions of the same classes of functions that do not appeal to any model of computation but that can instead be derived from standard mathematical tools, including the definition of a small set of primitive functions and the ability to construct new functions using operators such as composition and recursion? The answer is yes.

In the rest of this section we will develop such a definition for a class of functions that turns out to be exactly, given an appropriate encoding, the recursive functions. And we will develop a similar definition for the class of recursively enumerable functions. We will build a theory of functions, each of which has a domain that is an ordered n -tuple of natural numbers and a range that is the natural numbers. We have already shown that numbers can be represented as strings and strings can be represented as numbers, so there is no fundamental incompatibility between the theory we are about to describe and the one, based on Turing machines operating on strings, that we have already considered.

25.2.1 Primitive Recursive Functions

We begin by defining the ***primitive recursive functions*** to be the smallest class of functions from $\mathbb{N} \times \mathbb{N} \times \dots \times \mathbb{N}$ to \mathbb{N} that includes:

- the constant function 0,
- the successor function: $\text{succ}(n) = n + 1$, and
- a family of projection functions: for any $0 < j \leq k$, $p_{k,j}(n_1, n_2, \dots, n_k) = n_j$,

and that is closed under the operations:

- composition of g with h_1, h_2, \dots, h_k :

$$g(h_1(\), h_2(\), \dots, h_k(\)).$$

- primitive recursion of f in terms of g and h :

$f(n_1, n_2, \dots, n_k, 0) = g(n_1, n_2, \dots, n_k)$. This is the base case.

$f(n_1, n_2, \dots, n_k, m+1) = h(n_1, n_2, \dots, n_k, m, f(n_1, n_2, \dots, n_k, m))$. Note that in this, the recursive case, the function h takes a large number of arguments. It need not, however, use all of them, since the projection functions make it possible to select only those arguments that are needed.

Example 25.5 Primitive Recursive Functions Perform Arithmetic

To make these examples easier to read, we will define the constant $1 = \text{succ}(0)$.

All of the following functions are primitive recursive:

- The function *plus*, which adds two numbers:
 $\text{plus}(n, 0) = p_{1,1}(n) = n$.
 $\text{plus}(n, m+1) = \text{succ}(p_{3,3}(n, m, \text{plus}(n, m)))$.

For clarity, we will simplify our future definitions by omitting the explicit calls to the projection functions. Doing that here, we get:

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

- The function *times*:
 $\text{times}(n, 0) = 0$.
 $\text{times}(n, m+1) = \text{plus}(n, \text{times}(n, m))$.
- The function *factorial*, more usually written $n!$:
 $\text{factorial}(0) = 1$.
 $\text{factorial}(n+1) = \text{times}(\text{succ}(n), \text{factorial}(n))$.
- The function *exp*, more usually written n^m :
 $\text{exp}(n, 0) = 1$.
 $\text{exp}(n, m+1) = \text{times}(n, \text{exp}(n, m))$.
- The predecessor function *pred*, which is defined as follows:
 $\text{pred}(0) = 0$.
 $\text{pred}(n+1) = n$.

Many other straightforward functions are also primitive recursive. We may now wish to ask, “What is the relationship between the primitive recursive functions and the computable functions?” All of the primitive recursive functions that we have considered so far are computable. Are all primitive recursive functions computable? Are all computable functions primitive recursive? We will answer these questions one at a time.

Theorem 25.5 Every Primitive Recursive Function is Computable

Theorem: Every primitive recursive function is computable.

Proof: Each of the basic functions, as well as the two combining operations can be implemented in a straightforward fashion on a Turing machine or using a standard programming language. We omit the details. ■

Theorem 25.6 Not Every Computable Function is Primitive Recursive

Theorem: There exist computable functions that are not primitive recursive.

Proof: The proof is by diagonalization. We will consider only unary functions; we will show that there exists at least one unary computable function that is not primitive recursive.

We first observe that it is possible to create a lexicographic enumeration of the definitions of the unary primitive recursive functions. To do so, we first define an alphabet Σ that contains the symbols 0, 1, the letters of the alphabet (for use as function names), and the special characters (,), = and comma (,). Using the definition of the primitive recursive functions given above, we can build a Turing machine M that decides the language of syntactically legal unary primitive recursive functions. So, to produce the desired lexicographic enumeration of the primitive recursive function definitions, it suffices to enumerate lexicographically all strings over Σ^* and output only those that are accepted by M . We will choose to number the elements of this enumeration (the function definitions) starting with 0.

Using the lexicographic enumeration of the primitive recursive function definitions that we just described and a straightforward lexicographic enumeration of the natural numbers (the possible arguments to those functions), we can imagine the table T , shown in Table 25.4. $T[i, j]$ contains the value of f_i applied to j . Since every primitive recursive function is computable, there exists a Turing machine that can compute the value for any cell in T when it is required.

	0	1	2	3	4	5	...
f_0							
f_1							
f_2							
f_3							
f_4							
f_5							
...							

Table 25.4 Using diagonalization to prove that there are computable functions that are not primitive recursive

We now define the function $diagonal(n) = succ(T(n, n))$, which can be computed by the following Turing machine M :

$M(n) =$

1. Run the Turing machine that computes f_n on n . Let the value it produces be x .
2. Return $x+1$.

The function $diagonal$ is computable (by M) but it is not in the enumeration of primitive recursive functions since it differs from each of those in at least one place. So there exist computable functions that are not primitive recursive. ■

25.2.2 Ackermann's Function

Now we know that there exists at least one computable primitive recursive function that is not computable. But are there others? The answer is yes.

Consider Ackermann's function A , defined as follows on the domain $\mathbb{N} \times \mathbb{N}$:

$$\begin{aligned} A(0, y) &= y + 1. \\ A(x + 1, 0) &= A(x, 1). \\ A(x + 1, y + 1) &= A(x, A(x + 1, y)). \end{aligned}$$

Table 25.5 shows a few values for A . Table 25.6 comments on some of the values in the last row of Table 25.5.

x \ y	0	1	2	3	4
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	$2^{2^{2^{65536}}}-3$

Table 25.5 The first few values of Ackermann’s function

	Decimal digits required to express this value	To put that number in perspective
(4, 2)	19,729	There have been about $12 \cdot 10^9$ years or $3 \cdot 10^{17}$ seconds since the Big Bang.
(4, 3)	10^{5940}	There are about 10^{79} atoms in the causal universe.
(4, 4)	$10^{10^{5939}}$	

Table 25.6 Ackermann’s function grows very fast

So imagine that, at every second since the Big Bang, we had written one digit on every atom in the universe. By now we would have written approximately $3 \cdot 10^{96}$ digits, which is not enough to have written (much less computed) $A(4,3)$.

Ackermann’s function, unlike the busy beaver functions of Section 25.1.4, is recursive (computable). Ignoring memory and stack overflow, it is easy to write a program to compute it. But Ackermann’s function is not primitive recursive. While it does not grow as fast as the busy beaver functions, it does grow faster than many other fast-growing functions like *fermat*. It is possible to prove that A is not primitive recursive precisely because it grows so quickly and there is an upper bound on the rate at which primitive recursive functions can grow.

Yet A is computable, given an unbounded amount of time and memory. So it is another example of a computable function that is not primitive recursive.

25.2.3 Recursive (Computable) Functions

Since there are computable functions that are not primitive recursive, we are still looking for a way to define exactly the functions that Turing machines can compute.

We next define the class of μ -recursive functions using the same basic functions that we used to define the primitive recursive functions. We will again allow function composition and primitive recursion. But we will add one way of defining a new function.

We must first define a new notion: the *minimalization* f of a function g (of $k + 1$ arguments) is a function of k arguments defined as follows:

$$f(n_1, n_2, \dots, n_k) = \begin{cases} \text{the smallest } m \text{ such that } g(n_1, n_2, \dots, n_k, m) = 1, & \text{if there is such an } m, \\ 0, & \text{otherwise.} \end{cases}$$

Clearly, given any function g and any set of k arguments to it, there either is at least one value m such that $g(n_1, n_2, \dots, n_k, m) = 1$ or there isn’t. If there is at least one such value, then there is a smallest one (since we are considering only the natural numbers). So there always exists a function f that is the minimalization of g . If g is computable, then we can build a Turing machine T_{\min} that almost computes f as follows:

$T_{\min}(n_1, n_2, \dots, n_k) =$
 1. $m = 0.$
 2. While $g(n_1, n_2, \dots, n_k, m) \neq 1$ do:
 $m = m + 1.$
 3. Return $m.$

The problem is that T_{\min} will not halt if no value of m exists. There is no way for T_{\min} to discover that no such value exists and thus return 0.

Since we are trying to build a theory of computable functions (those for which there exists a Turing machine that always halts), we next define the class of minimalizable functions as follows: A function g is *minimalizable* iff, for every n_1, n_2, \dots, n_k , there is an m such that $g(n_1, n_2, \dots, n_k, m) = 1$. In other words, g is minimalizable if T_{\min} , as defined above, always halts.

We define the *μ -recursive functions* to be the smallest class of functions from $\mathbb{N} \times \mathbb{N} \times \dots \times \mathbb{N}$ to \mathbb{N} that includes:

- the constant function 0,
- the successor function: $\text{succ}(n) = n + 1$, and
- the family of projection functions: for any $k \geq j > 0$, $p_{k,j}(n_1, n_2, \dots, n_k) = n_j$,

and that is closed under the operations:

- composition of g with h_1, h_2, \dots, h_k :
 $g(h_1(\), h_2(\), \dots, h_k(\))$,
- primitive recursion of f in terms of g and h , and
- minimalization of minimalizable functions.

A good way to get an intuitive understanding of the difference between the primitive recursive functions and the μ -recursive functions is the following:

- In the computation of any primitive recursive function, iteration is always bounded; it can be implemented with a *for* loop that runs for n_k steps, where n_k is the value of the last argument to f . So, for example, computing *times*(2, 3) requires invoking *plus* three times.
- In the computation of a μ -recursive function, on the other hand, iteration may require the execution of a *while* loop like the one in T_{\min} . So it is not always possible to impose a bound, in advance, on the number of steps required by the computation.

Theorem 25.7 Equivalence of μ -Recursion and Computability

Theorem: A function is μ -recursive iff it is computable.

Proof: We must show both directions:

- Every μ -recursive function is computable. We show this by showing how to build a Turing machine for each of the basic functions and for each of the combining operations.
- Every computable function is μ -recursive. We show this by showing how to construct μ -recursive functions to perform each of the operations that a Turing machine can perform.

We will omit the details of both of these steps. They are straightforward but tedious. ■

We have now accomplished our first goal. We have a functional definition for the class of computable functions.

It is worth pointing out here why the same diagonalization argument that we used in the case of primitive recursive functions cannot be used again to show that there must exist some computable function that is not μ -recursive. The key to the argument in the case of the primitive recursive functions was that it was possible to create a lexicographic enumeration of exactly the primitive recursive function definitions. The reason it was possible is that a simple examination of the syntax of a proposed function tells us whether or not it is primitive recursive. But now consider trying to do the same thing to decide whether a function f is μ -recursive. If f is defined in terms of the minimalization of some other function g , we would first have to check to see whether g is minimalizable. To do that, we would need to know whether T_{\min} halts on all inputs. That problem is undecidable. So there exists no lexicographic enumeration of the μ -recursive functions.

Next we will attempt to find a functional definition for the class of partially computable functions.

We define the *partial μ -recursive functions* to be the smallest class of functions from $\mathbb{N} \times \mathbb{N} \times \dots \times \mathbb{N}$ to \mathbb{N} that includes:

- the constant function 0,
- the successor function: $\text{succ}(n) = n + 1$,
- the family of projection functions: for any $k \geq j > 0$, $p_{k,j}(n_1, n_2, \dots, n_k) = n_j$,

and that is closed under the operations:

- composition of g with h_1, h_2, \dots, h_k ,
- primitive recursion of f in terms of g and h , and
- minimalization.

The only difference between this definition and the one that we gave for the μ -recursive functions is that we now allow minimalization of any function, not just the minimalizable ones. A function that is defined in this way may, therefore, not be a total function. So it is possible that there exists no Turing machine that computes it and that always halts.

Theorem 25.8 Equivalence of Partial μ -Recursion and Partial Computability

Theorem: A function is a partial μ -recursive function iff it is partially computable.

Proof: We must show both directions:

- Every partial μ -recursive function is partially computable. We show this by showing how to build a Turing machine for each of the basic functions and for each of the combining operations. Note that the Turing machine that implements the minimalization of a function that is not minimalizable will not be guaranteed to halt on all inputs.
- Every partially computable function is partial μ -recursive. We show this by showing how to construct μ -recursive functions to perform each of the operations that a Turing machine can perform.

We will omit the details of both of these steps. They are straightforward but tedious. ■

25.3 The Recursion Theorem and its Use

In this section, we prove the existence of a very useful computable (recursive) function: *obtainSelf*. When called as a subroutine by any Turing machine M , *obtainSelf* writes onto M 's tape the string encoding of M .

We begin by asking whether there exists a Turing machine that implements the following specification:

virus() =

1. For each address in address book do:
 Write a copy of myself.
 Mail it to the address.
2. Do something fun and malicious like change one bit in every file on the machine.
3. Halt.

In particular, can we implement step 1.1 and build a program that writes a copy of itself? That seems simple until we try. A program that writes any literal string $s = "a_1 a_2 a_3 \dots a_n"$ is simply:

Write " $a_1 a_2 a_3 \dots a_n$ ".

But, using that simple string encoding of a program, this program is 8 characters longer than the string it writes. So if we imagined that our original code had length k then the program to write it would have length $k + 8$. But if that code contained the write statement, we would need to write:

Write "Write" || " $a_1 a_2 a_3 \dots a_n$ ".

But now we need to write that, and so forth. Perhaps this seems hopeless. But it is not.

First, let's rearrange *virus* a little bit:

virus() =

1. *copyme* = copy of myself.
2. For each address in address book do:
 - 2.1. Mail *copyme* to the address.
3. Do something fun and malicious like change one bit in every file on the machine.
4. Halt.

If *virus* can somehow get a single copy of itself onto its tape, a simple loop (of fixed length, independent of the length of the copy) can make additional copies, which can then be treated like any other string. The problem is for *virus* to get access to that first copy of itself. Here's how we can solve that problem.

First, we will define a family of printing functions, P_s . For any literal string s , P_s is the description of a Turing machine that writes the string s onto the tape. Think of s as being hardwired into P_s . For example, $P_{\text{abbb}} = \langle \text{aRbRbRbR} \rangle$. Notice that the length of the Turing machine P_s depends on the length of s .

Next we define a Turing machine, *createP*, that takes a string s as input on one tape and outputs the printing function P_s on a second tape:

createP(s) =

1. For each character c in s (on tape 1) do on tape 2:
 - 1.1. Write c .
 - 1.2. Write R.

Notice that the length of *createP* is fixed. It does not need separate code for each character of s . It has just one simple loop that reads the characters of s one at a time and outputs two characters for each.

Now let's break *virus* down into two parts:

- Step 1: We'll call this step *copy*. It writes on the tape a string that is the description of *virus*.
- Steps 2, 3, and 4, or whatever else *virus* wants to do: We'll call this part *work*. This part begins with *virus*'s description on the tape and does whatever it wants with it.

We will further break step 1, *copy*, down into two pieces that we will call *A* and *B*. *A* will execute first. Its job will be to write $\langle B, work \rangle$, the description of *B* and *work* onto the tape. The string $\langle B, work \rangle$ will be hardwired into *A*, so the length of *A* itself depends on $|\langle B, work \rangle|$. When *A* is done, the tape will be as shown in Figure 25.2 (a).

The job of *B* will be to write $\langle A \rangle$, the description of *A*, onto the tape immediately to the left of what *A* wrote. So, after *B* has finished, the job of copying *virus* will be complete and the tape will be as shown in Figure 25.2 (b).

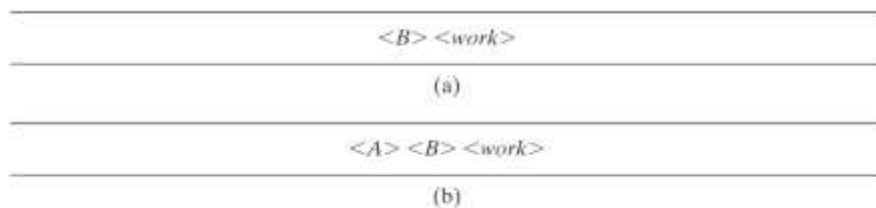


Figure 25.2 The result of running *A* and then *B*

Suppose that we knew exactly what *B* was. Then *A* would be $P_{\langle B \rangle \langle work \rangle}$. Assuming that we describe *A* in our macro language, $|\langle A \rangle|$ would then be $2 \cdot |\langle B \rangle \langle work \rangle|$, since for each character it must write and then move one square to the right. But what is *B*? It must be a machine that writes $\langle A \rangle$. And its length must be fixed. It cannot depend on the length of $\langle A \rangle$, since then the length of $\langle A \rangle$ would depend on the length of $\langle B \rangle$, which would depend on the length of $\langle A \rangle$ and so forth. So it cannot just be $P_{\langle A \rangle}$.

Fortunately, we know how to build *B* so that it writes $\langle A \rangle$ and does so with a fixed chunk of code, independent of the length of *A*. Given any string *s* on tape 1, *createP* writes, onto a second tape, the description of a Turing machine that writes *s*. And it does so with a fixed length program. *A* is a program that writes a string. So perhaps *B* could use *createP* to write a description of *A*. That will work if *B* has access to the string *s* that *A* wrote. But it does. *A* wrote $\langle B \rangle \langle work \rangle$, which is exactly what is on the tape when *B* gets control. So we have (expanding out the code for *createP*):

- B* =
1. /* Invoke *createP* to write onto tape 2 the code that writes the string that is currently on tape 1. For each character *c* in *s* (on tape 1) do on tape 2:
 - 1.1 Write *c*.
 - 1.2 Write R.
 2. /* Copy tape 2 to tape 1, moving right to left. Place this copy to the left of what is already on tape 1. Starting at the rightmost character *c* on tape 2 and the blank immediately to the left of the leftmost character on tape 1, loop until all characters have been processed:
 - 2.1 Copy *c* to tape 1.
 - 2.2 Move both read/write heads one square to the left.

So the code for *B* (unlike the code for *A*) is independent of the particular Turing machine of which we need to make a copy.

When *B* starts, the two tapes will be as shown in Figure 25.3 (a). After step 1, they will be as shown in Figure 25.3 (b). Remember that $\langle A \rangle$ is the description of a Turing machine that writes $\langle B \rangle \langle work \rangle$. Then, after step 2, they will be as shown in Figure 25.3 (c).

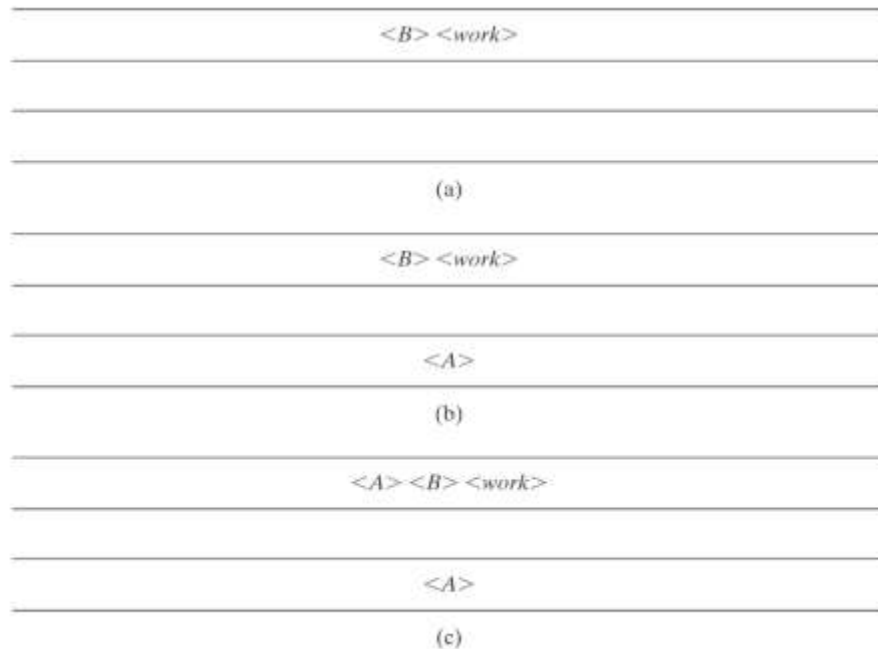


Figure 25.3 The tape before, during, and after the execution of B

Notice that the code for B is fixed. It first writes $\langle A \rangle$ onto tape 2 using a simple loop. Then, starting from the right, it copies $\langle A \rangle$ onto tape 2 just to the left of the string $\langle B \rangle \langle work \rangle$ that was already there. Again, it does this with a simple loop.

Now we can describe *virus* exactly as follows. Recall that $\langle M \rangle$ means the string description, written in the macro language described in Section 17.6, of the Turing machine M . So $\langle B \rangle$ is the description of the Turing machine labeled B here:

```

virus() =
  A: Write on tape 1  $\langle B \rangle \langle work \rangle$ .
  B: /* createP, which will write onto tape 2 the code that writes the string that is currently on tape 1.
      For each character  $c$  in  $s$  (on tape 1) do on tape 2:
          Write  $c$ .
          Write R.
      /* Copy tape 2 to tape 1, moving right to left. Place this copy to the left of what is already on tape 1.
      Starting at the rightmost character  $c$  on tape 2 and the blank immediately to the left of the leftmost character
      on tape 1, loop until all characters have been processed:
          Copy  $c$  to tape 1.
          Move both read/write heads one square to the left.
  work.

```

Or, more succinctly, using P_s and *createP*:

```

virus() =
  A:  $P_{\langle B \rangle \langle work \rangle}$ .
  B: createP.
  work.

```

The construction that we just did for *virus* is not unique to it. In fact, that construction enables us to describe the function *obtainSelf*, which we mentioned at the beginning of this section. Let M be a Turing machine composed of two steps:

1. *obtainSelf*.
2. *work* (which may exploit the description that *obtainSelf* produced).

Then we can define *obtainSelf*, which constructs $\langle M \rangle$:

obtainSelf(*work*) =
 A: $P_{\langle B \rangle \langle work \rangle}$.
 B: *createP*.

The Recursion Theorem, defined below, tells us that any Turing machine can obtain its own description and then use that description as it sees fit. There is one issue that we must confront in showing that, however. *Virus* ignored its input. But many Turing machines don't. So we need a way to write the description of a Turing machine M onto its tape without destroying its input. This is easy. If M is a k -tape Turing machine, we build a $k+2$ tape machine, where the extra two tapes are used, as we have just described, to create a description of M .

Theorem 25.9 The Recursion Theorem

Theorem: For every Turing machine T that computes a partially computable function t of two string arguments, there exists a Turing machine R that computes a partially computable function r of one string argument and:

$$\forall x (r(x) = t(\langle R \rangle, x)).$$

To understand the Recursion Theorem, it helps to see an example. Recall the Turing machine that we specified in our proof of Theorem 21.14 (that the language of descriptions of minimal Turing machines is not in SD):

$M\#(x) =$

1. Invoke *obtainSelf* to produce $\langle M\# \rangle$.
2. Run *ENUM* until it generates the description of some Turing machine M' whose description is longer than $|\langle M\# \rangle|$.
3. Invoke the universal Turing machine U on the string $\langle M', x \rangle$.

Steps 2 and 3 are the guts of $M\#$ and correspond to a Turing machine T that takes two arguments, $\langle M\# \rangle$ and x , and computes a function we can call t . $M\#$, on the other hand, takes a single argument, x . But $M\#(x)$ is exactly $T(\langle M\# \rangle, x)$ because in step 1, $M\#$ constructs $\langle M\# \rangle$, which it then hands to T (i.e., steps 2 and 3). So, given that we wish to compute $T(\langle M\# \rangle, x)$, $M\#$ is the Turing machine R that the Recursion Theorem says must exist. The only difference between R and T is that R constructs its own description and then passes that description, along with its own argument, on to T . Since, for any T , R must exist, it must always be possible for R to construct its own description and pass it to T .

Proof: The proof is by construction. The construction is identical to the one we showed above in our description of *virus* except that we substitute T for *work*. ■

The Recursion Theorem is sometimes stated in a different form, as a fixed-point theorem. We will state that version as a separate theorem whose proof follows from the Recursion Theorem as just stated and proved.

Theorem 25.10 The Fixed-Point Definition of the Recursion Theorem

Theorem: Let $f: \{\langle M \rangle : M \text{ is a Turing machine description}\} \rightarrow \{\langle M \rangle : M \text{ is a Turing machine description}\}$ be any computable function on the set of Turing machine descriptions. There exists some Turing machine F such that $f(\langle F \rangle)$ is the description of some Turing machine G and it is the case that F and G are equivalent (i.e., they behave identically on all inputs). We call F a **fixed point** of the function f , since it does not change when f is applied to it.

Proof: The Turing machine F that we claim must exist is:

$F(x) =$

1. Invoke *obtainSelf* to produce $\langle F \rangle$.
2. Since f is a computable function, there must be some Turing machine M_f that computes it. Invoke $M_f(\langle F \rangle)$, which produces the description of some Turing machine we can call G .
3. Run G on x .

Whatever f is, $f(\langle F \rangle) = \langle G \rangle$. F and G are equivalent since, on any input x , F halts exactly when G would halt and it leaves on its tape exactly what G leaves. ■

This theorem says something interesting and, at first glance perhaps, counterintuitive. Let's consider again the *virus* program that we described above. In its *work* section, it changes one bit in every file on its host machine. Consider the files that correspond to programs. Theorem 25.10 says that there exists at least one program whose behavior will not change when it is altered in that way. Of course, most programs will change. That is why *virus* can be so destructive. But there is not only one fixed point for *virus*, there are many, including:

- Every program that infinite loops on all inputs and where the bit that f changes comes after the section of code that went into the loop.
- Every program that has a chunk of redundant code, such as:
$$a = 5$$
$$a = 7$$
where the bit that gets changed is in the first value that is assigned and then overwritten.
- Every program that has a branch that can never be reached and where the bit that f changes is in the unreachable chunk of code.

We have stated and proved the Recursion Theorem in terms of the operation of Turing machines. It can also be stated and proved in the language of recursive functions. When done this way, its proof relies on another theorem that is interesting in its own right. We state and prove it next. To do so, we need to introduce a new technique for describing functions since, so far, we have described them as strings (i.e., the string encodings of the Turing machines that compute them). Yet the theory of recursive functions is a theory of functions on the natural numbers.

We define the following one-to-one function *Gödel* that maps from the set of Turing machines to the positive integers: Let M be a Turing machine that computes some partially computable function. Let $\langle M \rangle$ be the string description of M , using the encoding mechanism that we defined in Section 17.6.1. That encoding scheme used nine symbols, which can be encoded in binary using four bits. Rewrite $\langle M \rangle$ as a binary string. Now view that string as the number it encodes. We note that *Gödel* is a function (since each Turing machine is assigned a unique number); it is one-to-one (since no two Turing machines are assigned the same number); but it is not onto (since there are numbers that do not encode any Turing machine). A one-to-one function that assigns natural numbers to objects is called a ***Gödel numbering***, since the technique was introduced by Kurt Gödel. It played a key role in the proof of his Incompleteness Theorem.

We'll now create a second Gödel numbering, this time of the partial recursive functions. For each such function, assign to it the smallest number that has been assigned to some Turing machine that computes it. Now define:

φ_k to be the partially computable function with Gödel number k .

Notice that since functions are now represented as numbers, it is straightforward to talk about functions whose inputs and/or outputs are other functions. We'll take advantage of this and describe our next result. Suppose that $f(x_1, x_2, \dots, x_m, y_1, y_2, \dots, y_n)$ is an arbitrary function of $m+n$ arguments. Then we'll see that it is always possible, whenever we fix values for x_1, x_2, \dots, x_m , to create a new function f' of only n arguments. The new function f' will behave as though it were f with the fixed values supplied for the first m arguments. One way to think of f' is that it encapsulates f and a set of values v_1, v_2, \dots, v_m . We'll show that there exists a family of functions, one for each pair of values m and n , that, given f and v_1, v_2, \dots, v_m , creates f' as required.

Theorem 25.11 The *s-m-n* Theorem

Theorem: For all $m, n \geq 1$, there exists a computable function $s_{m,n}$ with the following property: Let k be the Gödel number of some partially computable function of $m+n$ arguments. Then:

For all $k, v_1, v_2, \dots, v_m, y_1, y_2, \dots, y_n$:

- $s_{m,n}(k, v_1, v_2, \dots, v_m)$ returns a number j that is the Gödel number of some partially computable function of n arguments, and
- $\varphi_j(y_1, y_2, \dots, y_n) = \varphi_k(v_1, v_2, \dots, v_m, y_1, y_2, \dots, y_n)$.

Proof: We will prove the theorem by defining a family of Turing machines $M_{m,n}$ that compute the $s_{m,n}$ family of functions. On input $(k, v_1, v_2, \dots, v_m)$, $M_{m,n}$ will construct a new Turing machine M_j that operates as follows on input w : Write v_1, v_2, \dots, v_m on the tape immediately to the left of w ; move the read/write head all the way to the left in front of v_1 ; and pass control to the Turing machine encoded by k . $M_{m,n}$ will then return j , the Gödel number of the function computed by M_j . ■

The *s-m-n* Theorem has important applications in the design of functional programming languages. § 671. In particular, it is the basis for **currying**, which implements the process we have just described. When a function of $k > 0$ arguments is curried, one or more of its arguments are fixed and a new function, of fewer arguments, is constructed.

25.4 Exercises

1) Define the function $pred(x)$ as follows:

$$pred: \mathbb{N} \rightarrow \mathbb{N},$$
$$pred(x) = x - 1.$$

- Is $pred$ a total function on \mathbb{N} ?
 - If not, is it a total function on some smaller, decidable domain?
 - Show that $pred$ is computable by defining an encoding of the elements of \mathbb{N} as strings over some alphabet Σ and then showing a Turing machine that halts on all inputs and that computes either $pred$ or $pred'$ (using the notion of a primed function as described in Section 25.1.2).
- Prove that every computable function is also partially computable.
 - Consider $f: A \rightarrow \mathbb{N}$, where $A \subseteq \mathbb{N}$. Prove that, if f is partially computable, then A is semidecidable (i.e., Turing enumerable).
 - Give an example, other than $steps$, of a function that is partially computable but not computable.
 - Define the function $countL(\langle M \rangle)$ as follows:

$$countL: \{\langle M \rangle : M \text{ is a Turing machine}\} \rightarrow \mathbb{N} \cup \{\aleph_0\},$$
$$countL(\langle M \rangle) = \text{the number of input strings that are accepted by Turing machine } M.$$

- Is $countL$ a total function on $\{\langle M \rangle : M \text{ is a Turing machine}\}$?
 - If not, is it a total function on some smaller, decidable domain?
 - Is $countL$ computable, partially computable, or neither? Prove your answer.
- Give an example, other than any mentioned in the book, of a function that is not partially computable.

- 7) Let g be some partially computable function that is not computable. Let h be some computable function and let $f(x) = g(h(x))$. Is it possible that f is a computable function?
- 8) Prove that the busy beaver function Σ is not computable.
- 9) Prove that each of the following functions is primitive recursive:
- The function $double(x) = 2x$.
 - The proper subtraction function $monus$, which is defined as follows:

$$monus(n, m) = \begin{cases} n - m & \text{if } n > m \\ 0 & \text{if } n \leq m \end{cases}$$

- The function $half$, which is defined as follows:

$$half(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ (n - 1)/2 & \text{if } n \text{ is odd} \end{cases}$$

- 10) Let A be Ackermann's function. Verify that $A(4, 1) = 65533$.

26 Summary and References

One way to think about what we have done in Part IV is to explore the limits of computation. We have considered many different models of “the computable”. All of them were described and studied by people who were trying to answer the question, “What can we compute?” Some of the models look similar. For example, Post production systems and unrestricted grammars both define languages by providing a start symbol and a set of production rules that rewrite one string into another. While there are differences (Post systems exploit variables and must match entire strings while unrestricted grammars use only constants and can match substrings), it turns out that the two formalisms are identical: they both define exactly the class of languages that we are calling SD. Similarly, Turing machines and tag systems look similar. One uses a tape with a moveable read/write head, the other uses a first-in, first-out queue. But that difference also turns out not to matter. A machine of either kind can be simulated by a machine of the other kind.

Some of the models look very different. Turing machines seem like sequential computers. Expressions in the lambda calculus read like mathematical function definitions. Unrestricted grammars are rewrite systems. One of the most important structural differences is between the models (such as Turing machines, tag systems, the lambda calculus, semi-Thue systems, and Markov algorithms) that accept inputs, and so compute functions, and those (such as unrestricted grammars, Post systems, and Lindenmayer systems) that include a start symbol and so generate languages. But all of these systems can be viewed as mechanisms for defining languages. The generating systems generate languages; the function-computation systems compute a language’s characteristic function. So even that difference doesn’t effect the bottom line of what is computable.

Another thing that we did in Part IV was to introduce three new classes of languages: D, SD, and the context-sensitive languages. Table 26.1 summarizes the properties of those languages and compares them to the regular and the context-free languages.

	<i>Regular</i>	<i>Context-Free</i>	<i>Context-Sensitive</i>	<i>D</i>	<i>SD</i>
<i>Automaton</i>	FSM	PDA	LBA		TM
<i>Grammar(s)</i>	Regular Regular expressions	Context-free	Context-sensitive		Unrestricted
<i>ND = D?</i>	Yes	No	unknown		Yes
<i>Closed under:</i>					
<i>Concatenation</i>	Yes	Yes	Yes	Yes	Yes
<i>Union</i>	Yes	Yes	Yes	Yes	Yes
<i>Kleene star</i>	Yes	Yes	Yes	Yes	Yes
<i>Complement</i>	Yes	No	Yes	Yes	No
<i>Intersection</i>	Yes	No	Yes	Yes	Yes
<i>∩ with Regular</i>	Yes	Yes	Yes	Yes	Yes
<i>Decidable:</i>					
<i>Membership</i>	Yes	Yes	Yes		No
<i>Emptiness</i>	Yes	Yes	No		No
<i>Finiteness</i>	Yes	Yes	No		No
<i>= Σ*</i>	Yes	No	No		No
<i>Equivalence</i>	Yes	No	No		No

Table 26.1 Comparing the classes of languages

References

Gödel's Completeness Theorem was presented in [Gödel 1929]. His Incompleteness Theorem was presented in [Gödel 1931].

The Entscheidungsproblem was articulated in [Hilbert and Ackermann 1928]. In [Church 1936], Alonzo Church defined the lambda calculus and proved that no solution to the Entscheidungsproblem exists. In [Turing 1936], Alan Turing defined the Turing Machine and also proved the unsolvability of the Entscheidungsproblem. Many of the early papers on computability have been reprinted in [Davis 1965]. The Turing machine description language defined in Chapter 17 is patterned closely after one described in [Lewis and Papadimitriou 1998].

Post published his work on tag systems in [Post 1943]. [Minsky 1961] showed that tag systems have the same computational power as Turing machines. As a result, that claim is sometimes called Minsky's Theorem. Post also described his production rule system in [Post 1943]. A good modern treatment can be found in [Taylor 1998].

Markov algorithms were first described (in Russian) in [Markov 1951]. A good treatment in English is [Markov and Nagorny 1988].

A description of Conway's Game of Life was first published in [Gardner 1970]. [Berlekamp, Conway, and Guy 1982] describe a proof of the equivalence of Turing machines and the Game of Life. [Rendell 2000] describes an implementation of a Turing machine in Life.


One dimensional cellular automata are described in detail in [Wolfram 2002].

The first experiment in DNA computing was described in [Adleman 1994] and [Adleman 1998]. A detailed mathematical treatment of the subject can be found in [Păun, Rozenberg and Salomaa 1998].

See [Lagarias 1985] for a comprehensive discussion of the $3x+1$ problem.

Hilbert's 10th problem was shown to be undecidable as a result of Matiyasevich's Theorem, published in [Matiyasevich 1970].

The undecidability of the Post Correspondence Problem was shown in [Post 1946]. The proof that we present in § 649 was modeled after the one in [Linz 2001]. The fact that the Post Correspondence Problem is decidable if limited to instances of size two was shown in [Ehrenfeucht, Karhumäki and Rozenberg 1982].

Wang tiles were first described in [Wang 1961]. Also in that paper, Wang articulated the hypothesis we called Wang's conjecture; he proved that, if the conjecture is true, then the tiling problem is decidable. [Berger 1966] showed that Wang's conjecture is false by demonstrating a set of 20,426 tiles that tile the plane only aperiodically. [Culik 1996] showed an aperiodic set of just 13 tiles .

Presburger arithmetic was defined in [Presburger 1929]. [Fischer and Rabin 1974] showed that any decision procedure for Presburger arithmetic requires time that is $\mathcal{O}(2^{2^{cn}})$.

See [Bar-Hillel, Perles and Shamir 1961], [Ginsburg and Rose 1963], and [Hartmanis and Hopcroft 1968] for fundamental results on the undecidability of questions involving context-free languages. The fact that it is undecidable whether a context-free grammar is ambiguous was published independently by [Cantor 1962], [Floyd 1962] and [Chomsky and Schutzenberger 1963].

Theorem 23.4, which tells us that the word problem for semi-Thue systems is undecidable, was proved in [Post 1947].

The Chomsky hierarchy was defined in [Chomsky 1959], with unrestricted grammars as the most powerful of the formalisms to occur in the hierarchy. Also in that paper Chomsky proved Theorem 23.1, which says that the set of languages that can be generated by an unrestricted grammar is equivalent to the set SD.

[Chomsky 1959] also defined the context-sensitive languages to be those that could be described with a context-sensitive grammar. It also proved Theorem 24.4, which says that the context-sensitive languages are a proper subset of D . The equivalence of the context-sensitive languages in that sense and the languages that can be accepted by a (nondeterministic) linear bounded automaton was shown in [Kuroda 1964]. The fact that the context-sensitive languages (unlike the context-free ones) are closed under intersection was proved in [Landweber 1963]. The proofs we give for the closure of the context-sensitive languages under union, concatenation, Kleene star, and intersection are from [Hopcroft and Ullman 1979]. The fact that the membership problem for context-sensitive languages is NP-hard was proved in [Karp 1972].

Attribute grammars as a way to define the semantics of context-free languages were introduced in [Knuth 1968]. For an introduction to the use of feature/unification grammars in natural language processing, see [Jurafsky and Martin 2000].

Lindenmayer systems (L-systems) were first described in [Lindenmayer 1968]. See [Prusinkiewicz and Lindenmayer 1990] for an excellent description of them and of their use as the basis for simulations of plant development. The L-system that generates the trees in Example 24.6 was taken from [Ochoa 1998].

The busy beaver functions were first described in [Rado 1962].

Primitive recursive functions were described in [Dedekind 1888]. See [Martin 2003] for a comprehensive discussion of primitive recursive functions with many examples.

[Ackermann 1928] showed the existence of a function that was computable but not primitive recursive. His original function was one of three variables. Rózsa Péter and Raphael Robinson created the simpler version, of two variables, that now bears Ackermann's name. It was described in [Péter 1967].

The μ -recursive functions are described in [Kleene 1936a]. The s - m - n Theorem and the Recursion Theorem are also due to Kleene. See [Kleene 1964]. The constructive proof that we present for Theorem 25.9 follows the one given in [Sipser 2006].