3 THE CHURCH-TURING THESIS

So far in our development of the theory of computation we have presented several models of computing devices. Finite automata are good models for devices that have a small amount of memory. Pushdown automata are good models for devices that have an unlimited memory that is usable only in the last in, first out manner of a stack. We have shown that some very simple tasks are beyond the capabilities of these models. Hence they are too restricted to serve as models of general purpose computers.

3.1 TURING MACHINES

We turn now to a much more powerful model, first proposed by Alan Turing in 1936, called the ***Turing machine***. Similar to a finite automaton but with an unlimited and unrestricted memory, a Turing machine is a much more accurate model of a general purpose computer. A Turing machine can do everything that a real computer can do. Nonetheless, even a Turing machine cannot solve certain problems. In a very real sense, these problems are beyond the theoretical limits of computation.

The Turing machine model uses an infinite tape as its unlimited memory. It has a tape head that can read and write symbols and move around on the tape.

Initially the tape contains only the input string and is blank everywhere else. If the machine needs to store information, it may write this information on the tape. To read the information that it has written, the machine can move its head back over it. The machine continues computing until it decides to produce an output. The outputs *accept* and *reject* are obtained by entering designated accepting and rejecting states. If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting.

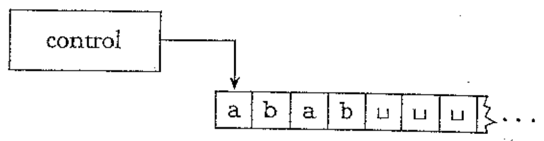

FIGURE 3.1
Schematic of a Turing machine

The following list summarizes the differences between finite automata and Turing machines.

1. A Turing machine can both write on the tape and read from it.
2. The read–write head can move both to the left and to the right.
3. The tape is infinite.
4. The special states for rejecting and accepting take immediate effect.

Let's consider a Turing machine M_1 for testing membership in the language $B = \{w\#w \mid w \in \{0,1\}^*\}$. That is, we want to design M_1 to accept if its input is a member of B. To understand M_1 better, put yourself in its place by imagining that you are standing on a mile-long input consisting of millions of characters. Your goal is to determine whether the input is a member of B, that is, whether the input comprises two identical strings separated by a # symbol. The input is too long for you to remember it all, but you are allowed to move back and forth over the input and make marks on it. Of course, the obvious strategy is to zig-zag to the corresponding places on the two sides of the # and determine whether they match. Use marks to keep track of which places correspond.

We design M_1 to work in the same way. It makes multiple passes over the input string with the read–write head. On each pass it matches one of the characters on each side of the # symbol. To keep track of which symbols have been checked already, M_1 crosses off each symbol as it is examined. If it crosses off all the symbols, that means that everything matched successfully, and M_1 goes into an accept state. If it discovers a mismatch, it enters a reject state. In summary, M_1's algorithm is as follows.

M_1 = "O
1.
2.
3.

The fo
puting in

FIGURE
Snapsho

This
not give
giving fo
automat
tic
never gi
big.

FORM

The hea
cause it
machin
chine is
a, and i

```
$M_1$ = "On input string $w$:
    1. Scan the input to be sure that it contains a single # symbol. If
       not, *reject*.
    2. Zig-zag across the tape to corresponding positions on either side
       of the # symbol to check on whether these positions contain the
       same symbol. If they do not, *reject*. Cross off symbols as they
       are checked to keep track of which symbols correspond.
    3. When all symbols to the left of the # have been crossed off, check
       for any remaining symbols to the right of the #. If any symbols
       remain, *reject*; otherwise *accept*."
```

The following figure contains several snapshots of M_1's tape while it is computing in stages 2 and 3 when started on input 011000#011000.

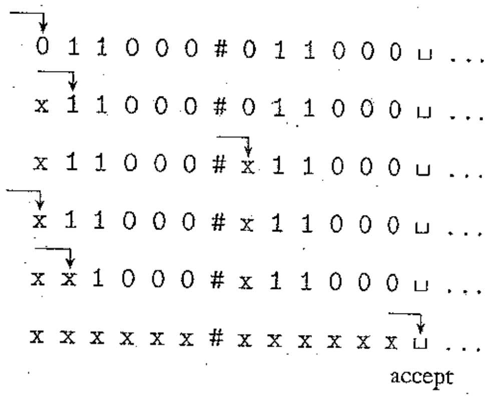
0 1 1 0 0 0 # 0 1 1 0 0 0 ␣ ...
x 1 1 0 0 0 # 0 1 1 0 0 0 ␣ ...
x 1 1 0 0 0 # x 1 1 0 0 0 ␣ ...
x 1 1 0 0 0 # x 1 1 0 0 0 ␣ ...
x x 1 0 0 0 # x 1 1 0 0 0 ␣ ...
x x x x x x # x x x x x x ␣ ...
accept

FIGURE 3.2
Snapshots of Turing machine M_1 computing on input 011000#011000

This description of Turing machine M_1 sketches the way it functions but does not give all its details. We can describe Turing machines in complete detail by giving formal descriptions analogous to those introduced for finite and pushdown automata. The formal description specifies each of the parts of the formal definition of the Turing machine model to be presented shortly. In actuality we almost never give formal descriptions of Turing machines because they tend to be very big.

FORMAL DEFINITION OF A TURING MACHINE

The heart of the definition of a Turing machine is the transition function δ because it tells us how the machine gets from one step to the next. For a Turing machine, δ takes the form: $Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$. That is, when the machine is in a certain state q and the head is over a tape square containing a symbol a, and if $\delta(q, a) = (r, b, \mathrm{L})$, the machine writes the symbol b replacing the a, and goes to state r. The third component is either L or R and indicates whether the head moves to the left or right after writing. In this case the L indicates a move to the left.

DEFINITION 3.1

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the special ***blank*** symbol $\sqcup$,
3. Γ is the tape alphabet, where $\{\sqcup\} \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ computes as follows. Initially M receives its input $w = w_1 w_2 \ldots w_n \in \Sigma^*$ on the leftmost n squares of the tape, and the rest of the tape is blank (i.e., filled with blank symbols). The head starts on the leftmost square of the tape. Note that Σ does not contain the blank symbol, so the first blank appearing on the tape marks the end of the input. Once M starts, the computation proceeds according to the rules described by the transition function. If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move, even though the transition function indicates L. The computation continues until it enters either the accept or reject states at which point it halts. If neither occurs, M goes on forever.

As a Turing machine computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a ***configuration*** of the Turing machine. Configurations often are represented in a special way. For a state q and two strings u and v over the tape alphabet Γ we write $u\,q\,v$ for the configuration where the current state is q, the current tape contents is uv, and the current head location is the first symbol of v. The tape contains only blanks following the last symbol of v. For example, $1011q_701111$ represents the configuration when the tape is 101101111, the current state is q_7, and the head is currently on the second 0. The following figure depicts a Turing machine with that configuration.

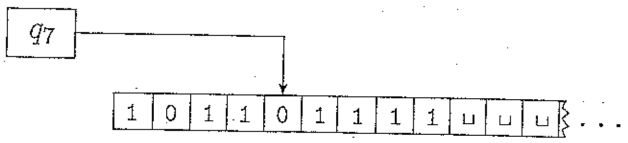

FIGURE **3.3**
A Turing machine with configuration $1011q_701111$

Here we formalize our intuitive understanding of the way that a Turing machine computes. Say that configuration C_1 ***yields*** configuration C_2 if the Turing machine can legally go from C_1 to C_2 in a single step. We define this notion formally as follows.

Suppose that we have a and b in Γ, as well as u and v in Γ^* and states q_i and q_j. In that case $ua\,q_i\,bv$ and $u\,q_j\,acv$ are two configurations. Say that

$$ua\,q_i\,bv \quad \text{yields} \quad u\,q_j\,acv$$

if in the transition function $\delta(q_i, b) = (q_j, c, \mathrm{L})$. That handles the case where the Turing machine moves leftward. For a rightward move, say that

$$ua\,q_i\,bv \quad \text{yields} \quad uac\,q_j\,v$$

if $\delta(q_i, b) = (q_j, c, \mathrm{R})$.

Special cases occur when the head is at one of the ends of the configuration. For the left-hand end, the configuration $q_i\,bv$ yields $q_j\,cv$ if the transition is left moving (because we prevent the machine from going off the left-hand end of the tape), and it yields $c\,q_jv$ for the right moving transition. For the right-hand end, the configuration $ua\,q_i$ is equivalent to $ua\,q_i\,\sqcup$ because we assume that blanks follow the part of the tape represented in the configuration. Thus we can handle this case as before, with the head no longer at the right-hand end.

The ***start configuration*** of M on input w is the configuration $q_0\,w$, which indicates that the machine is in the start state q_0 with its head at the leftmost position on the tape. In an ***accepting configuration*** the state of the configuration is q_{accept}. In a ***rejecting configuration*** the state of the configuration is q_{reject}. Accepting and rejecting configurations are ***halting configurations*** and accordingly do not yield further configurations. A Turing machine M ***accepts*** input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists where

1. C_1 is the start configuration of M on input w,
2. each C_i yields C_{i+1}, and
3. C_k is an accepting configuration.

The collection of strings that M accepts is ***the language of*** M, denoted $L(M)$.

DEFINITION 3.2

Call a language ***Turing-recognizable*** if some Turing machine recognizes it.[1]

When we start a TM on an input, three outcomes are possible. The machine may *accept*, *reject*, or *loop*. By ***loop*** we mean that the machine simply does not halt. It is not necessarily repeating the same steps in the same way forever as the connotation of looping may suggest. Looping may entail any simple or complex behavior that never leads to a halting state.

A Turing machine M can fail to accept an input by entering the q_{reject} state and rejecting, or by looping. Sometimes distinguishing a machine that is looping from one that is merely taking a long time is difficult. For this reason we prefer Turing machines that halt on all inputs; such machines never loop. These machines are called ***deciders*** because they always make a decision to accept or reject. A decider that recognizes some language also is said to ***decide*** that language.

DEFINITION 3.3

Call a language ***Turing-decidable*** or simply ***decidable*** if some Turing machine decides it.[2]

Every decidable language is Turing-recognizable but certain Turing-recognizable languages are not decidable. We now give some examples of decidable languages. We present examples of languages that are Turing-recognizable but not decidable after we develop a technique for proving undecidability in Chapter 4.

EXAMPLES OF TURING MACHINES

As we did for finite and pushdown automata, we can give a formal description of a particular Turing machine by specifying each of its seven parts. However, going to that level of detail for Turing machines can be cumbersome for all but the tiniest machines. Accordingly, we won't spend much time giving such descriptions. Mostly we will give only higher level descriptions because they are precise enough for our purposes and are much easier to understand. Nevertheless, it is important to remember that every higher level description is actually just shorthand for its formal counterpart. With patience and care we could describe any of the Turing machines in this book in complete formal detail.

To help you make the connection between the formal descriptions and the higher level descriptions, we give state diagrams in the next two examples. You may skip over them if you already feel comfortable with this connection.

[1]It is called a ***recursively enumerable*** language in some other textbooks.
[2]It is called a ***recursive*** language in some other textbooks.

EXAMPLE 3.4

Here we describe a TM M_2 that recognizes the language consisting of all strings of 0s whose length is a power of 2. It decides the language $A = \{0^{2^n} \mid n \geq 0\}$.

```
$M_2$ = "On input string $w$:
    1. Sweep left to right across the tape, crossing off every other 0.
    2. If in stage 1 the tape contained a single 0, accept.
    3. If in stage 1 the tape contained more than a single 0 and the
       number of 0s was odd, reject.
    4. Return the head to the left-hand end of the tape.
    5. Go to stage 1."
```

Each iteration of stage 1 cuts the number of 0s in half. As the machine sweeps across the tape in stage 1, it keeps track of whether the number of 0s seen is even or odd. If that number is odd and greater than 1, the original number of 0s in the input could not have been a power of 2. Therefore the machine rejects in this instance. However, if the number of 0s seen is 1, the original number must have been a power of 2. So in this case the machine accepts.

Now we give the formal description of $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$.

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0\}$, and
- $\Gamma = \{0, \text{x}, \sqcup\}$.
- We describe δ with a state diagram (see Figure 3.4).
- The start, accept, and reject states are q_1, q_{accept}, and q_{reject}.

In the state diagram in Figure 3.4 the label $0 \rightarrow \sqcup,\text{R}$ appears on the transition from q_1 to q_2. It signifies that, when in state q_1 with the head reading 0, the machine goes to state q_2, writes $\sqcup$, and moves the head to the right. In other words, $\delta(q_1,0) = (q_2,\sqcup,\text{R})$. For clarity we use the shorthand $0 \rightarrow \text{R}$ in the transition from q_3 to q_4, as meaning that the machine moves to the right when reading 0 in state q_4 but doesn't alter the tape, so $\delta(q_3,0) = (q_4,0,\text{R})$.

This machine begins by writing a blank symbol over the leftmost 0 on the tape so that it can find the left-hand end of the tape in stage 4. Whereas we would normally use a more suggestive symbol such as # for the left-hand end delimiter, we use a blank here to keep the tape alphabet, and hence the state diagram, small. Example 3.6 gives another method of finding the left-hand end of the tape.

We give a sample run of this machine on input 0000. The starting configuration is $q_1$0000. The sequence of configurations the machine enters appears following Figure 3.4. Read down the columns and left to right.

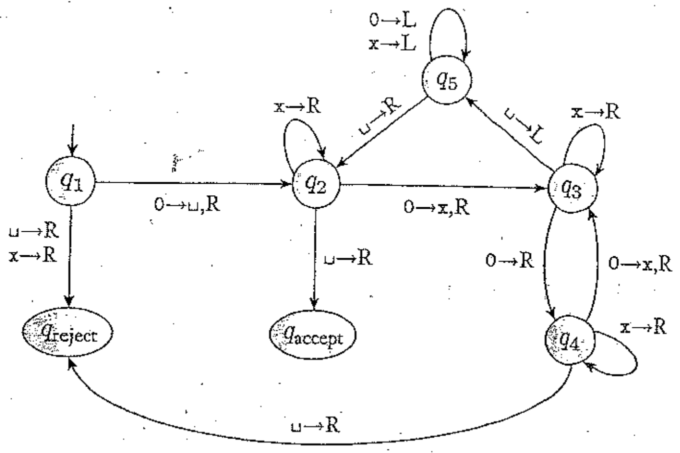

FIGURE 3.4
State diagram for Turing machine M_2

A sample run of M_2 on input 0000:

$q_1 0000$	$\sqcup q_5 \mathrm{x}0\mathrm{x}\sqcup$	$\sqcup \mathrm{x} q_5 \mathrm{xx}\sqcup$
$\sqcup q_2 000$	$q_5 \sqcup \mathrm{x}0\mathrm{x}\sqcup$	$\sqcup q_5 \mathrm{xxx}\sqcup$
$\sqcup \mathrm{x} q_3 00$	$\sqcup q_2 \mathrm{x}0\mathrm{x}\sqcup$	$q_5 \sqcup \mathrm{xxx}\sqcup$
$\sqcup \mathrm{x}0 q_4 0$	$\sqcup \mathrm{x} q_2 0\mathrm{x}\sqcup$	$\sqcup q_2 \mathrm{xxx}\sqcup$
$\sqcup \mathrm{x}0\mathrm{x} q_3 \sqcup$	$\sqcup \mathrm{xx} q_3 \mathrm{x}\sqcup$	$\sqcup \mathrm{x} q_2 \mathrm{xx}\sqcup$
$\sqcup \mathrm{x}0 q_5 \mathrm{x}\sqcup$	$\sqcup \mathrm{xxx} q_3 \sqcup$	$\sqcup \mathrm{xx} q_2 \mathrm{x}\sqcup$
$\sqcup \mathrm{x} q_5 0\mathrm{x}\sqcup$	$\sqcup \mathrm{xx} q_5 \mathrm{x}\sqcup$	$\sqcup \mathrm{xxx} q_2 \sqcup$
		$\sqcup \mathrm{xxx}\sqcup q_{\text{accept}}$

■

NONDETERMINISTIC TURING MACHINES

A nondeterministic Turing machine is defined in the expected way. At any point in a computation the machine may proceed according to several possibilities. The transition function for a nondeterministic Turing machine has the form

$$\delta : Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}).$$

The computation of a nondeterministic Turing machine is a tree whose branches correspond to different possibilities for the machine. If some branch of the computation leads to the accept state, the machine accepts its input. If you feel the need to review nondeterminism, turn to Section 1.2 on page 47. Now we show that nondeterminism does not affect the power of the Turing machine model.

THEOREM 3.10

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

PROOF IDEA We show that we can simulate any nondeterministic TM N with a deterministic TM D. The idea behind the simulation is to have D try all possible branches of N's nondeterministic computation. If D ever finds the accept state on one of these branches, D accepts. Otherwise, D's simulation will not terminate.

We view N's computation on an input w as a tree. Each branch of the tree represents one of the branches of the nondeterminism. Each node of the tree is a configuration of N. The root of the tree is the start configuration. The TM D searches this tree for an accepting configuration. Conducting this search carefully is crucial lest D fail to visit the entire tree. A tempting, though bad, idea is to have D explore the tree by using depth first search. The depth first search strategy goes all the way down one branch before backing up to explore other branches. If D were to explore the tree in this manner, D could go forever down one infinite branch and miss an accepting configuration on some other branch. Hence we design D to explore the tree by using breadth first search instead. This strategy explores all branches to the same depth before going on to explore any branch to the next depth. This method guarantees that D will visit every node in the tree until it encounters an accepting configuration.

FIG
Det

I
can
give
that
to t
that
the
in N
to a
add
over
add
add

1
2
3

4

PROOF The simulating deterministic TM D has three tapes. By Theorem 3.8 this arrangement is equivalent to having a single tape. The machine D uses its three tapes in a particular way, as illustrated in the following figure. Tape 1 always contains the input string and is never altered. Tape 2 maintains a copy of N's tape on some branch of its nondeterministic computation. Tape 3 keeps track of D's location in N's nondeterministic computation tree.

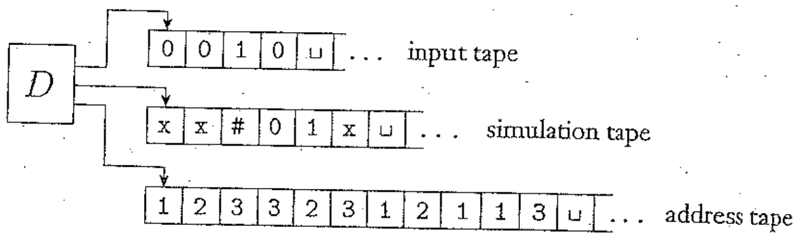
D
0 0 1 0 ␣ ... input tape
x x # 0 1 x ␣ ... simulation tape
1 2 3 3 2 3 1 2 1 1 3 ␣ ... address tape

FIGURE **3.7**
Deterministic TM D simulating nondeterministic TM N

Let's first consider the data representation on tape 3. Every node in the tree can have at most b children, where b is the size of the largest set of possible choices given by N's transition function. To every node in the tree we assign an address that is a string over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$. We assign the address 231 to the node we arrive at by starting at the root, going to its 2nd child, going to that node's 3rd child, and finally going to that node's 1st child. Each symbol in the string tells us which choice to make next when simulating a step in one branch in N's nondeterministic computation. Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration. In that case the address is invalid and doesn't correspond to any node. Tape 3 contains a string over Σ_b. It represents the branch of N's computation from the root to the node addressed by that string, unless the address is invalid. The empty string is the address of the root of the tree. Now we are ready to describe D.

1. Initially tape 1 contains the input w, and tapes 2 and 3 are empty.
2. Copy tape 1 to tape 2.
3. Use tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before each step of N consult the next symbol on tape 3 to determine which choice to make among those allowed by N's transition function. If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, *accept* the input.
4. Replace the string on tape 3 with the lexicographically next string. Simulate the next branch of N's computation by going to stage 2.

COROLLARY 3.11

A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.

PROOF Any deterministic TM is automatically a nondeterministic TM and so one direction of this theorem follows immediately. The other direction follows from Theorem 3.10.

We can modify the proof of Theorem 3.10 so that if N always halts on all branches of its computation, D will always halt. We call a nondeterministic Turing machine a ***decider*** if all branches halt on all inputs. Exercise 3.3 asks you to modify the proof in this way to obtain the following corollary to Theorem 3.10.

COROLLARY 3.12

A language is decidable if and only some nondeterministic Turing machine decides it.

ENUMERATORS

As we mentioned in an earlier footnote, some people use the term *recursively enumerable* language for Turing-recognizable language. That term originates from a type of Turing machine variant called an enumerator. Loosely defined, an enumerator is a Turing machine with an attached printer. The Turing machine can use that printer as an output device to print strings. Every time the Turing machine wants to add a string to the list, it sends the string to the printer. Exercise 3.4 asks you to give a formal definition an enumerator. The following figure depicts a schematic of this model.

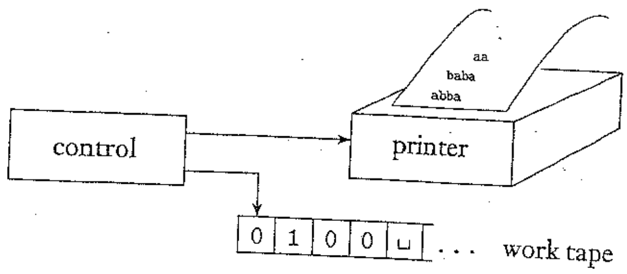

FIGURE 3.8
Schematic of an enumerator

An enumerator starts with a blank input tape. If the enumerator doesn't halt, it may print an infinite list of strings. The language enumerated by E is the collection of all the strings that it eventually prints out. Moreover, E may generate the strings of the language in any order, possibly with repetitions. Now we are ready to develop the connection between enumerators and Turing-recognizable languages.

THEOREM 3.13

A language is Turing-recognizable if and only if some enumerator enumerates it.

PROOF First we show that if we have an enumerator E that enumerates a language A, a TM M recognizes A. The TM M works in the following way.

```
M = "On input w:
    1. Run E. Every time that E outputs a string, compare it with w.
    2. If w ever appears in the output of E, accept."
```

Clearly, M accepts those strings that appear on E's list.

Now we do the other direction. If TM M recognizes a language A, we can construct the following enumerator E for A. Say that $s_1, s_2, s_3, \ldots$ is a list of all possible strings in Σ^*.

```
E = "Ignore the input.
    1. Repeat the following for $i = 1, 2, 3, \ldots$
    2.     Run M for $i$ steps on each input, $s_1, s_2, \ldots, s_i$.
    3.     If any computations accept, print out the corresponding $s_j$."
```

If M accepts a particular string s, eventually it will appear on the list generated by E. In fact, it will appear on the list infinitely many times because M runs from the beginning on each string for each repetition of step 1. This procedure gives the effect of running M in parallel on all possible input strings.

EQUIVALENCE WITH OTHER MODELS

So far we have presented several variants of the Turing machine model and have shown them to be equivalent in power. Many other models of general purpose computation have been proposed. Some of these models are very much like Turing machines, while others are quite different. All share the essential feature of Turing machines, namely, unrestricted access to unlimited memory, distinguishing them from weaker models such as finite automata and pushdown automata. Remarkably, *all* models with that feature turn out to be equivalent in power, so long as they satisfy certain reasonable requirements.[3]

[3]For example, one requirement is the ability to perform only a finite amount of work in a single step.

To understand this phenomenon consider the analogous situation for programming languages. Many, such as Pascal and LISP, look quite different from one another in style and structure. Can some algorithm be programmed in one of them and not the others? Of course not—we can compile LISP into Pascal and Pascal into LISP, which means that the two languages describe *exactly* the same class of algorithms. So do all other reasonable programming languages. The widespread equivalence of computational models holds for precisely the same reason. Any two computational models that satisfy certain reasonable requirements can simulate one another and hence are equivalent in power.

This equivalence phenomenon has an important philosophical corollary. Even though there are many different computational models, the class of algorithms that they describe is unique. Whereas each individual computational model has a certain arbitrariness to its definition, the underlying class of algorithms that it describes is natural because it is the same class that other models describe. This phenomenon also has had profound implications for mathematics, as we show in the next section.

3.3

THE DEFINITION OF ALGORITHM

Informally speaking, an ***algorithm*** is a collection of simple instructions for carrying out some task. Commonplace in everyday life, algorithms sometimes are called *procedures* or *recipes*. Algorithms also play an important role in mathematics. Ancient mathematical literature contains descriptions of algorithms for a variety of tasks, such as finding prime numbers and greatest common divisors. In contemporary mathematics algorithms abound.

Even though algorithms have had a long history in mathematics, the notion of algorithm itself was not defined precisely until the twentieth century. Before that, mathematicians had an intuitive notion of what algorithms were and relied upon that notion when using and describing them. But that intuitive notion was insufficient for gaining a deeper understanding of algorithms. The following story relates how the precise definition of algorithm was crucial to one important mathematical problem.

HILBERT'S PROBLEMS

In 1900, mathematician David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris. In his lecture, he identified twenty-three mathematical problems and posed them as a challenge for the coming century. The tenth problem on his list concerned algorithms.

Before describing that problem, let's briefly discuss polynomials. A ***polynomial*** is a sum of terms, where each ***term*** is a product of certain variables and a constant called a ***coefficient.*** For example,

$$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z = 6x^3yz^2$$

is a term with coefficient 6, and

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

is a polynomial with four terms over the variables x, y, and z. A ***root*** of a polynomial is an assignment of values to its variables so that the value of the polynomial is 0. This polynomial has a root at $x = 5$, $y = 3$, and $z = 0$. This root is an ***integral root*** because all the variables are assigned integer values. Some polynomials have an integral root and some do not.

Hilbert's tenth problem was to devise an algorithm that tests whether a polynomial has an integral root. He did not use the term *algorithm* but rather "a process according to which it can be determined by a finite number of operations."[4] Interestingly, in the way he phrased this problem, Hilbert explicitly asked that an algorithm be "devised." Thus he apparently assumed that such an algorithm must exist—someone need only find it.

As we now know, no algorithm exists for this task; it is algorithmically unsolvable. For mathematicians of that period to come to this conclusion with their intuitive concept of algorithm would have been virtually impossible. The intuitive concept may have been adequate for giving algorithms for certain tasks, but it was useless for showing that no algorithm exists for a particular task. Proving that an algorithm does not exist requires having a clear definition of algorithm. Progress on the tenth problem had to wait for that definition.

The definition came in the 1936 papers of Alonzo Church and Alan Turing. Church used a notational system called the λ-calculus to define algorithms. Turing did it with his "machines." These two definitions were shown to be equivalent. This connection between the informal notion of algorithm and the precise definition has come to be called the ***Church–Turing thesis.***

The Church–Turing thesis provides the definition of algorithm necessary to resolve Hilbert's tenth problem. In 1970, Yuri Matijasevič, building on work of Martin Davis, Hilary Putnam, and Julia Robinson, showed that no algorithm exists for testing whether a polynomial has integral roots. In Chapter 4 we develop the techniques that form the basis for proving that this and other problems are algorithmically unsolvable.

Intuitive notion of algorithms	equals	*Turing machine algorithms*

FIGURE **3.9**
The Church–Turing Thesis

[4]Translated from the original German.

Let's phrase Hilbert's tenth problem in our terminology. Doing so helps to introduce some themes that we explore in Chapters 4 and 5. Let

$$D = \{p \mid p \text{ is a polynomial with an integral root}\}.$$

Hilbert's tenth problem asks in essence whether the set D is decidable. The answer is negative. In contrast, we can show that D is Turing-recognizable. Before doing so, let's consider a simpler problem. It is an analog of Hilbert's tenth problem for polynomials that have only a single variable, such as $4x^3 - 2x^2 + x - 7$. Let

$$D_1 = \{p \mid p \text{ is a polynomial over } x \text{ with an integral root}\}.$$

Here is a Turing machine M_1 that recognizes D_1:

```
M_1 = "The input is a polynomial p over the variable x.
    1. Evaluate p with x set successively to the values 0, 1, −1, 2, −2,
       3, −3, . . . If at any point the polynomial evaluates to 0, accept."
```

If p has an integral root, M_1 eventually will find it and accept. If p does not have an integral root, M_1 will run forever. For the multivariable case, we can present a similar Turing machine M that recognizes D. Here, M goes through all possible settings of its variables to integral values.

Both M_1 and M are recognizers but not deciders. We can convert M_1 to be a decider for D_1 because we can calculate bounds within which the roots of a single variable polynomial must lie and restrict the search to these bounds. In Problem 3.18 you are asked to show that the roots of such a polynomial must lie between the values

$$\pm k \frac{c_{\max}}{c_1},$$

where k is the number of terms in the polynomial, $c_{\max}$ is the coefficient with largest absolute value, and c_1 is the coefficient of the highest order term. If a root is not found within these bounds, the machine *rejects*. Matijasevič's theorem shows that calculating such bounds for multivariable polynomials is impossible.

such a
ing so
first is
sition
The s
in whi
its hea
tails o
use Er
At thi
head.
In t
variou
descri
them.
We
put to
than a
easily
of tho
sentati
the en
severa
$\langle O_1, C$
does n
late or
In c
ment c
volvin
the bl
the alg
ply w,
of an c