The correspondence between the generation of languages by grammars and their recognition by machines extends to the languages of Turing machines. If Turing machines represent the ultimate in string recognition machines, it seems reasonable to expect the associated family of grammars to be the most general string transformation systems. This is indeed the case; the grammars that correspond to Turing machines are called unrestricted grammars because there are no restrictions on the form or the applicability of their rules. To establish the correspondence between recognition by a Turing machine and generation by an unrestricted grammar, we show that a computation of a Turing machine can be simulated by a derivation in an unrestricted grammar.

With the acceptance of the Church-Turing Thesis, the extent of algorithmic problem solving can be identified with the capabilities of Turing machine computations. Consequently, to prove a problem to be unsolvable, it suffices to show that there is no Turing machine solution to the problem. Using this approach, we show that the Halting Problem for Turing machines is undecidable. That is, there is no algorithm that can determine, for an arbitrary Turing machine M and string w, whether M will halt when run with w. We will then use problem reduction to establish undecidability of additional questions about the results of Turing machine computations, of the existence of derivations using the rules of a grammar, and of properties of context-free languages.

CHAPTER 8

Turing Machines

The Turing machine, introduced by Alan Turing in 1936, represents another step in the development of finite-state computing machines. Turing machines were originally proposed for the study of effective computation and exhibit many of the features commonly associated with a modern computer. This is no accident; the Turing machine provided a model for the design and development of the stored-program computer. Utilizing a sequence of elementary operations, a Turing machine may access and alter any memory position. A Turing machine, unlike a computer, has no limitation on the amount of time or memory available for a computation.

The Church-Turing Thesis, which will be discussed in detail in Chapter 11, asserts that any effective procedure can be realized by a suitably designed Turing machine. The variations of Turing machine architectures and applications presented in the next two chapters indicate the robustness and the versatility of Turing machine computation.

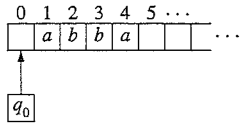
8.1 The Standard Turing Machine

A Turing machine is a finite-state machine in which a transition prints a symbol on the tape. The tape head may move in either direction, allowing the machine to read and manipulate the input as many times as desired. The structure of a Turing machine is similar to that of a finite automaton, with the transition function incorporating these additional features.

Definition 8.1.1

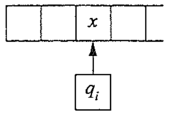
A **Turing machine** is a quintuple M = (Q, Σ, Γ, δ, q_0) where Q is a finite set of states, Γ is a finite set called the *tape alphabet,* Γ contains a special symbol B that represents a blank, Σ is a subset of $\Gamma - \{B\}$ called the *input alphabet,* δ is a partial function from Q $\times$ Γ to Q $\times$ $\Gamma \times \{L, R\}$ called the *transition function,* and $q_0 \in$ Q is a distinguished state called the *start state.*

The tape of a Turing machine has a left boundary and extends indefinitely to the right. Tape positions are numbered by the natural numbers, with the leftmost position numbered zero. Each tape position contains one element from the tape alphabet.

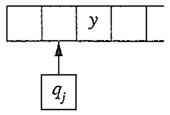


A computation begins with the machine in state q_0 and the tape head scanning the leftmost position. The input, a string from Σ^*, is written on the tape beginning at position one. Position zero and the remainder of the tape are blank. The diagram shows the initial configuration of a Turing machine with input *abba*. The tape alphabet provides additional symbols that may be used during a computation.

A transition consists of three actions: changing the state, writing a symbol on the square scanned by the tape head, and moving the tape head. The direction of the movement is specified by the final component of the transition. An L indicates a move of one tape position to the left and R one position to the right. The machine configuration



and transition $\delta(q_i, x) = [q_j, y, L]$ combine to produce the new configuration



The transition changed the state from q_i to q_j, replaced the tape symbol x with y, and moved the tape head one square to the left. The ability of the machine to move in both directions and process blanks introduces the possibility of a computation continuing indefinitely.

A computation halts when it encounters a state, symbol pair for which no transition is defined. A transition from tape position zero may specify a move to the left of the boundary of the tape. When this occurs, the computation is said to *terminate abnormally*. When we say that a computation halts, we mean that it terminates in a normal fashion.

The Turing machine presented in Definition 8.1.1 is deterministic, that is, at most one transition is specified for every combination of state and tape symbol. The one-tape deterministic Turing machine, with initial conditions as described above, is referred to as the **standard Turing machine**. The first two examples demonstrate the use of Turing machines to manipulate strings. After developing a facility with Turing machine computations, we will use Turing machines to accept languages and to compute functions.

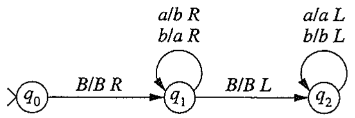
Example 8.1.1

The tabular representation of the transition function of a standard Turing machine with input alphabet $\{a, b\}$ is given in the table below.

δ	B	a	b
q_0	q_1, B, R		
q_1	q_2, B, L	q_1, b, R	q_1, a, R
q_2		q_2, a, L	q_2, b, L

The transition from state q_0 moves the tape head to position one to read the input. The transitions in state q_1 read the input string and interchange the symbols a and b. The transitions in q_2 return the machine to the initial position.

A Turing machine can be graphically represented by a state diagram. The transition $\delta(q_i, x) = [q_j, y, d]$, $d \in \{L, R\}$ is depicted by an arc from q_i to q_j labeled $x/y\ d$. The state diagram



represents the Turing machine defined in the preceding transition table. □

A machine configuration consists of the state, the tape, and the position of the tape head. At any step in a computation of a standard Turing machine, only a finite segment of the tape is nonblank. A configuration is denoted uq_ivB, where all tape positions to the right of the B are blank and uv is the string spelled by the symbols on the tape from the left-hand boundary to the B. Blanks may occur in the string uv; the only requirement is that the entire nonblank portion of the tape be included in uv. The notation uq_ivB indicates that the machine is in state q_i scanning the first symbol of v and the entire tape to the right of uvB is blank.

This representation of machine configurations can be used to trace the computations of a Turing machine. The notation $uq_ivB \underset{M}{\vdash} xq_jyB$ indicates that the configuration xq_jyB is obtained from uq_ivB by a single transition of M. Following the standard conventions, $uq_ivB \overset{*}{\underset{M}{\vdash}} xq_jyB$ signifies that xq_jyB can be obtained from uq_ivB by a finite number, possibly zero, of transitions. The reference to the machine is omitted when there is no possible ambiguity.

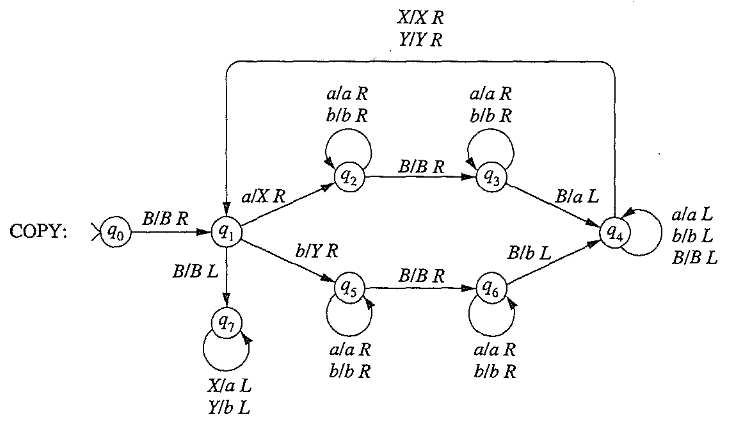
The Turing machine in Example 8.1.1 interchanges the a's and b's in the input string. Tracing the computation generated by the input string $abab$ yields

$$\begin{aligned}
&q_0BababB\\
&\vdash Bq_1ababB\\
&\vdash Bbq_1babB\\
&\vdash Bbaq_1abB\\
&\vdash Bbabq_1bB\\
&\vdash Bbabaq_1B\\
&\vdash Bbabq_2aB\\
&\vdash Bbaq_2baB\\
&\vdash Bbq_2abaB\\
&\vdash Bq_2babaB\\
&\vdash q_2BbabaB.
\end{aligned}$$

The Turing machine from Example 8.1.1 made two passes through the input string. Moving left to right, the first pass interchanged the a's and b's. The second pass, going right to left, simply returned the tape head to the leftmost tape position. The next example shows how Turing machine transitions can be used to make a copy of a string. The ability to copy data is an important component in many algorithmic processes. When copies are needed, the strategy employed be this machine can by modified to suit the type of data considered in the particular problem.

Example 8.1.2

The Turing machine COPY with input alphabet $\{a, b\}$ produces a copy of the input string. That is, a computation that begins with the tape having the form BuB terminates with tape $BuBuB$.



The computation copies the input string one symbol at a time beginning with the leftmost symbol in the input. Tape symbols X and Y record the portion of the input that has been copied. The first unmarked symbol in the string specifies the arc to be taken from state q_1. The cycle q_1, q_2, q_3, q_4, q_1 replaces an a with X and adds an a to the string being constructed. Similarly, the lower branch copies a b using Y to mark the input string. After the entire string has been copied, the transitions in state q_7 change the X's and Y's to a's and b's and return the tape head to the initial position. □

8.2 Turing Machines as Language Acceptors

Turing machines have been introduced as a paradigm for effective computation. A Turing machine computation consists of a sequence of elementary operations determined from the machine state and the symbol being read by the tape head. The machines constructed in the previous section were designed to illustrate the features of Turing machine computations. The computations read and manipulated the symbols on the tape; no interpretation was given to the result of a computation. Turing machines can be designed to accept languages and to compute functions. The result of a computation can be defined in terms of the state in which the computation terminates or the configuration of the tape at the end of the computation.

In this section we consider the use of Turing machines as language acceptors; a computation accepts or rejects the input string. Initially, acceptance is defined by the final state of the computation. This is similar to the technique used by finite-state and pushdown automata to accept strings. Unlike finite-state and pushdown automata, a Turing machine need not read the entire input string to accept the string. A Turing machine augmented with final states is a sextuple (Q, Σ, Γ, δ, q_0, F), where F ⊆ Q is the set of final states.

Definition 8.2.1

Let M = (Q, Σ, Γ, δ, q_0, F) be a Turing machine. A string $u \in \Sigma^*$ is **accepted by final state** if the computation of M with input u halts in a final state. A computation that terminates abnormally rejects the input regardless of the state in which the machine halts. The language of M, denoted L(M), is the set of all strings accepted by M.

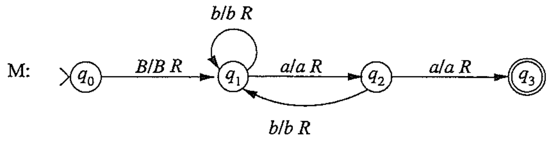
A language accepted by a Turing machine is called a **recursively enumerable language**. The ability of a Turing machine to move in both directions and process blanks introduces the possibility that the machine may not halt for a particular input. Thus there are three possible outcomes for a Turing machine computation: It may halt and accept the input string; halt and reject the string; or it may not halt at all. Because of the last possibility, we will sometimes say that a machine M *recognizes* L if it accepts L but does not necessarily halt for all input strings. The computations of M identify the strings L but may not provide answers for strings not in L.

A language accepted by a Turing machine that halts for all input strings is said to be **recursive**. Membership in a recursive language is decidable; the computations of a Turing machine that halts for all inputs provide a procedure for determining whether a string is in the language. A Turing machine of this type is sometimes said to *decide* the language. Being recursive is a property of a language, not of a Turing machine that accepts it. There are multiple Turing machines that accept a particular language; some may halt for all input, whereas others may not. The existence of one Turing machine that halts for all inputs is sufficient to show that the membership in the language is decidable and the language is recursive.

In Chapter 12 we will show that there are languages that are recognized by a Turing machine but are not decided by any Turing machine. It follows that the set of recursive languages is a proper subset of the recursively enumerable languages. The terms *recursive* and *recursively enumerable* have their origins in the functional interpretation of Turing computability that will be presented in Chapter 13.

Example 8.2.1

The Turing machine M



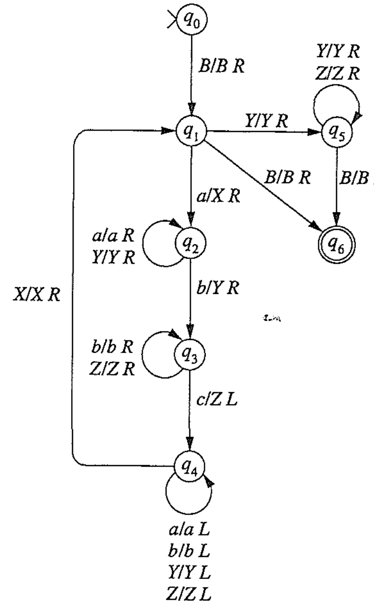
accepts the language $(a \cup b)^*aa(a \cup b)^*$. The computation

$$\begin{aligned}&q_0BaabbB\\ \vdash\ &Bq_1aabbB\\ \vdash\ &Baq_2abbB\\ \vdash\ &Baaq_3bbB\end{aligned}$$

examines only the first half of the input before accepting the string $aabb$. The language $(a \cup b)^*aa(a \cup b)^*$ is recursive; the computations of M halt for every input string. A successful computation terminates when a substring aa is encountered. All other computations halt upon reading the first blank following the input. □

Example 8.2.2

The language $L = \{a^ib^ic^i \mid i \geq 0\}$ is accepted by the Turing machine



The tape symbols X, Y, and Z mark the a's, b's, and c's as they are matched. A computation successfully terminates when all the symbols in the input string have been transformed to the appropriate tape symbol. The transition from q_1 to q_6 accepts the null string.

The Turing machine M shows that L is recursive. The computations for strings in L halt in q_6. For strings not in L, the computations halt in a nonaccepting state as soon as it is discovered that the input string does not match the pattern $a^ib^ic^i$. For example, the computation with input bca halts in q_1 and with input abb in q_3. □

8.3 Alternative Acceptance Criteria

Using Definition 8.2.1, the acceptance of a string by a Turing machine is determined by the state of the machine when the computation halts. Alternative approaches to defining acceptance are presented in this section.

The first alternative is acceptance by halting. In a Turing machine that is designed to accept by halting, an input string is accepted if the computation initiated with the string halts. Computations for which the machine terminates abnormally reject the string. When acceptance is defined by halting, the machine is defined by the quintuple $(Q, \Sigma, \Gamma, \delta, q_0)$. The final states are omitted since they play no role in the determination of the language of the machine.

Definition 8.3.1

Let $M = (Q, \Sigma, \Gamma, \delta, q_0)$ be a Turing machine. A string $u \in \Sigma^*$ is **accepted by halting** if the computation of M with input u halts (normally).

Turing machines designed for acceptance by halting are used for language recognition. The computation for any input not in the language will not terminate. Theorem 8.3.2 shows that any language recognized by a machine that accepts by halting is also accepted by a machine that accepts by final state.

Theorem 8.3.2

The following statements are equivalent:

i) The language L is accepted by a Turing machine that accepts by final state.
ii) The language L is accepted by a Turing machine that accepts by halting.

Proof. Let $M = (Q, \Sigma, \Gamma, \delta, q_0)$ be a Turing machine that accepts L by halting. The machine $M' = (Q, \Sigma, \Gamma, \delta, q_0, Q)$, in which every state is a final state, accepts L by final state.

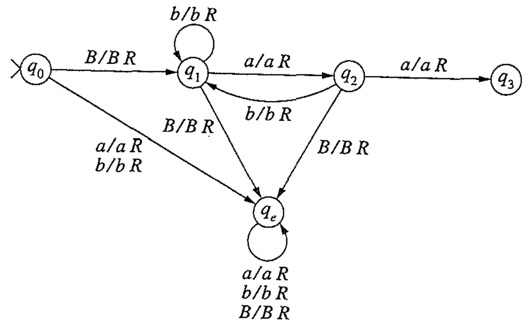
Conversely, let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a Turing machine that accepts the language L by final state. Define the machine $M' = (Q \cup \{q_e\}, \Sigma, \Gamma, \delta', q_0)$ that accepts by halting as follows:

i) If $\delta(q_i, x)$ is defined, then $\delta'(q_i, x) = \delta(q_i, x)$.
ii) For each state $q_i \in Q - F$, if $\delta(q_i, x)$ is undefined, then $\delta'(q_i, x) = [q_e, x, R]$.
iii) For each $x \in \Gamma$, $\delta'(q_e, x) = [q_e, x, R]$.

Computations that accept strings in M and M′ are identical. An unsuccessful computation in M may halt in a rejecting state, terminate abnormally, or fail to terminate. When an unsuccessful computation in M halts, the computation in M′ enters the state q_e. Upon entering q_e, the machine moves indefinitely to the right. The only computations that halt in M′ are those that are generated by computations of M that halt in an accepting state. Thus $L(M') = L(M)$. ■

Example 8.3.1

The Turing machine from Example 8.2.1 is altered to accept $(a \cup b)^*aa(a \cup b)^*$ by halting. The machine below is constructed as specified by Theorem 8.3.2. A computation enters q_e when the entire input string has been read and no aa has been encountered.



The machine obtained by deleting the arcs from q_0 to q_e and those from q_e to q_e labeled $a/a\ R$ and $b/b\ R$ also accepts $(a \cup b)^*aa(a \cup b)^*$ by halting. □

In Exercise 7 a type of acceptance, referred to as *acceptance by entering*, is introduced that uses final states but does not require the accepting computations to terminate. A string is accepted if the computation ever enters a final state; after entering a final state, the remainder of the computation is irrelevant to the acceptance of the string. As with acceptance by halting, any Turing machine designed to accept by entering can be transformed into a machine that accepts the same language by final state.

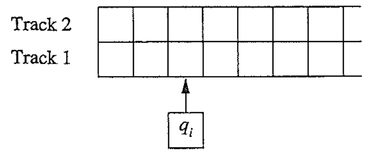
Unless noted otherwise, Turing machines will accept by final state as in Definition 8.2.1. The alternative definitions are equivalent in the sense that machines designed in this manner accept the same family of languages as those accepted by standard Turing machines.

8.4 Multitrack Machines

The remainder of the chapter is dedicated to examining variations of the standard Turing machine model. Each of the variations appears to increase the capability of the machine.

We prove that the languages accepted by these generalized machines are precisely those accepted by the standard Turing machines. Additional variations will be presented in the exercises.

A multitrack tape is one in which the tape is divided into tracks. A tape position in an n-track tape contains n symbols from the tape alphabet. The diagram depicts a two-track tape with the tape head scanning the second position.



The machine reads an entire tape position. Multiple tracks increase the amount of information that can be considered when determining the appropriate transition. A tape position in a two-track machine is represented by the ordered pair $[x, y]$, where x is the symbol in track 1 and y is in track 2.

The states, input alphabet, tape alphabet, initial state, and final states of a two-track machine are the same as in the standard Turing machine. A two-track transition reads and rewrites the entire tape position. A transition of a two-track machine is written $\delta(q_i, [x, y]) = [q_j, [z, w], d]$, where $d \in \{L, R\}$.

The input to a two-track machine is placed in the standard input position in track 1. All the positions in track 2 are initially blank. Acceptance in multitrack machines is by final state.

We will show that the languages accepted by two-track machines are precisely the recursively enumerable languages. The argument easily generalizes to n-track machines.

Theorem 8.4.1

A language L is accepted by a two-track Turing machine if, and only if, it is accepted by a standard Turing machine.

Proof. Clearly, if L is accepted by a standard Turing machine, it is accepted by a two-track machine. The equivalent two-track machine simply ignores the presence of the second track.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a two-track machine. A one-track machine will be constructed in which a single tape square contains the same information as a tape position in the two-track tape. The representation of a two-track tape position as an ordered pair indicates how this can be accomplished. The tape alphabet of the equivalent one-track machine M′ consists of ordered pairs of tape elements of M. The input to the two-track machine consists of ordered pairs whose second component is blank. The input symbol a of M is identified with the ordered pair $[a, B]$ of M′. The one-track machine

$$M' = (Q, \Sigma \times \{B\}, \Gamma \times \Gamma, \delta', q_0, F)$$

with transition function

$$\delta'(q_i, [x, y]) = \delta(q_i, [x, y])$$

accepts L(M). ■

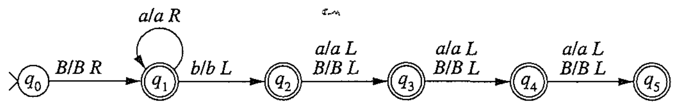
8.5 Two-Way Tape Machines

A Turing machine with a two-way tape is identical to the standard model except that the tape extends indefinitely in both directions. Since a two-way tape has no left boundary, the input can be placed anywhere on the tape. All other tape positions are assumed to be blank. The tape head is initially positioned on the blank to the immediate left of the input string. The advantage of a two-way tape is that the Turing machine designer need not worry about crossing the left boundary of the tape.

A machine with a two-way tape can be constructed to simulate the actions of a standard machine by placing a special symbol on the tape to represent the left boundary of the one-way tape. The symbol #, which is assumed not to be an element of the tape alphabet of the standard machine, is used to simulate the boundary of the tape. A computation in the equivalent machine with two-way tape begins by writing # to the immediate left of the initial tape head position. The remainder of a computation in the two-way machine is identical to that of the one-way machine except when the computation of the one-way machine terminates abnormally. When the one-way computation attempts to move to the left of the tape boundary, the two-way machine reads the symbol # and enters a nonaccepting state that terminates the computation.

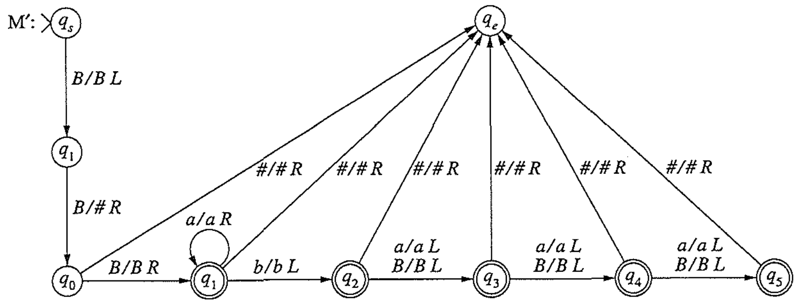
The standard Turing machine M

M:

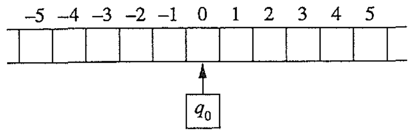


will be used to demonstrate the conversion of a machine with a one-way tape to an equivalent two-way machine. All the states of M other than q_0 are accepting. When the first b is encountered, the tape head moves four positions to the left, if possible. Acceptance is completely determined by the boundary of the tape. A string is rejected by M whenever the tape head attempts to cross the left-hand boundary. All computations that remain within the bounds of the tape accept the input. Thus the language of M consists of all strings over $\{a, b\}$ in which the first b, if present, is preceded by at least three a's.

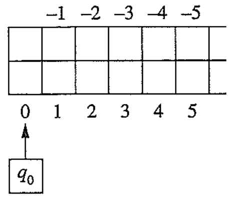
A machine M′ with a two-way tape can be obtained from M by the addition of three states q_s, q_t, and q_e. The transitions from states q_s and q_t insert the simulated endmarker to the left of the initial position of the tape head of M′, the two-way machine that accepts L(M). After writing the simulated boundary, the computation enters a copy of the one-way machine M. The error state q_e is entered in M′ when a computation in M attempts to move to the left of the tape boundary.



We will now show that a language accepted by a machine with a two-way tape is accepted by a standard Turing machine. The argument utilizes Theorem 8.4.1, which establishes the interdefinability of two-track and standard machines. The tape positions of the two-way tape can be numbered by the complete set of integers. The initial position of the tape head is numbered zero, and the input begins at position one.



Imagine taking the two-way infinite tape and folding it so that position $-i$ sits directly above position i. Adding an unnumbered tape square over position zero produces a two-track tape. The symbol in tape position i of the two-way tape is stored in the corresponding position of the one-way, two-track tape. A computation on a two-way infinite tape can be simulated on this one-way, two-track tape.



Let M = (Q, Σ, Γ, δ, q_0, F) be a Turing machine with a two-way tape. Using the correspondence between a two-way tape and a two-track tape, we construct a Turing machine M′ with a two-track, one-way tape to accept L(M). A transition of M is specified by the state and the symbol scanned. M′, scanning a two-track tape, reads two symbols at each

tape position. Symbols U (up) and D (down) are included in the states of M′ to designate which of the two tracks should be used to determine the transition.

The components of M′ are constructed from those of M and the symbols U and D:

$$Q' = (Q \cup \{q_s, q_t\}) \times \{U, D\}$$
$$\Sigma' = \Sigma$$
$$\Gamma' = \Gamma \cup \{\#\}$$
$$F' = \{[q_i, U],\ [q_i, D] \mid q_i \in F\}.$$

The initial state of M′ is a pair $[q_s, D]$. The transition from this state writes the marker # on the upper track in the leftmost tape position.

A transition from $[q_t, D]$ returns the tape head to its original position to begin the simulation of a computation of M. During the remainder of a computation, the # on track 2 is used to indicate when the tape head is reading position zero and to trigger changes from U to D in the state. The transitions of M′ are defined as follows:

1. $\delta'([q_s, D],\ [B, B]) = [[q_t, D],\ [B, \#],\ R]$.
2. For every $x \in \Gamma$, $\delta'([q_t, D],\ [x, B]) = [[q_0, D],\ [x, B],\ L]$.
3. For every $z \in \Gamma - \{\#\}$ and $d \in \{L, R\}$, $\delta'([q_i, D],\ [x, z]) = [[q_j, D],\ [y, z],\ d]$ whenever $\delta(q_i, x) = [q_j, y, d]$ is a transition of M.
4. For every $x \in \Gamma - \{\#\}$ and $d \in \{L, R\}$, $\delta'([q_i, U],\ [z, x]) = [[q_j, U],\ [z, y],\ d']$ whenever $\delta(q_i, x) = [q_j, y, d]$ is a transition of M, where d' is the opposite direction of d.
5. $\delta'([q_i, D],\ [x, \#]) = [[q_j, U],\ [y, \#],\ R]$ whenever $\delta(q_i, x) = [q_j, y, L]$ is a transition of M.
6. $\delta'([q_i, D],\ [x, \#]) = [[q_j, D],\ [y, \#],\ R]$ whenever $\delta(q_i, x) = [q_j, y, R]$ is a transition of M.
7. $\delta'([q_i, U],\ [x, \#]) = [[q_j, D],\ [y, \#],\ R]$ whenever $\delta(q_i, x) = [q_j, y, R]$ is a transition of M.
8. $\delta'([q_i, U],\ [x, \#]) = [[q_j, U],\ [y, \#],\ R]$ whenever $\delta(q_i, x) = [q_j, y, L]$ is a transition of M.

A transition generated by schema 3 simulates a transition of M in which the tape head begins and ends in positions labeled with nonnegative values. In the simulation, this is represented by writing on the lower track of the tape. Transitions defined in schema 4 use only the upper track of the two-track tape. These correspond to transitions of M that occur to the left of position zero on the two-way infinite tape.

The remaining transitions simulate the transitions of M from position zero on the two-way tape. Regardless of the U or D in the state, transitions from position zero are determined by the tape symbol on track 1. When the track is specified by D, the transition is defined by schema 5 or 6. Transitions defined in 7 and 8 are applied when the state is $[q_i, U]$.

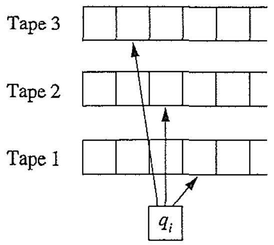
The preceding informal arguments outline the proof of the equivalence of one-way and two-way Turing machines.

Theorem 8.5.1

A language L is accepted by a Turing machine with a two-way tape if, and only if, it is accepted by a standard Turing machine.

8.6 Multitape Machines

A k-tape machine has k tapes and k independent tape heads. The states and alphabets of a multitape machine are the same as in a standard Turing machine. The machine reads the tapes simultaneously but has only one state. This is depicted by connecting each of the independent tape heads to a single control indicating the current state.



A transition is determined by the state and the symbols scanned by each of the tape heads. A transition in a multitape machine may

i) change the state,
ii) write a symbol on each of the tapes,
iii) independently reposition each of the tape heads.

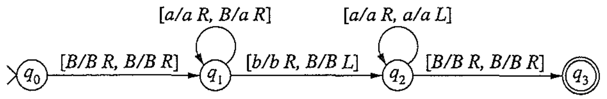
The repositioning consists of moving the tape head one square to the left or one square to the right or leaving it at its current position. A transition of a two-tape machine scanning x_1 on tape 1 and x_2 on tape 2 is written $\delta(q_i, x_1, x_2) = [q_j; y_1, d_1; y_2, d_2]$, where $x_i, y_i \in \Gamma$ and $d_i \in \{L, R, S\}$. This transition causes the machine to write y_i on tape i. The symbol d_i specifies the direction of the movement of tape head i: L signifies a move to the left, R a move to the right, and S means the head remains stationary. Any tape head attempting to move to the left of the boundary of its tape terminates the computation abnormally.

The input to a multitape machine is placed in the standard position on tape 1. All the other tapes are assumed to be blank. The tape heads originally scan the leftmost position of each tape. A multitape machine can be represented by a state diagram in which the label on an arc specifies the action for each tape. For example, the transition $\delta(q_i, x_1, x_2) = [q_j; y_1, d_1; y_2, d_2]$ will be represented by an arc from q_i to q_j labeled $[x_1/y_1\ d_1, x_2/y_2\ d_2]$.

Two advantages of multitape machines are the ability to copy data between tapes and to compare strings on different tapes. Both of these features will be demonstrated in the following example.

Example 8.6.1

The machine



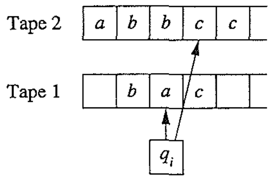
accepts the language $\{a^iba^i \mid i \geq 0\}$. A computation with input string a^iba^i copies the leading a's to tape 2 in state q_1. When the b is read on tape 1, the computation enters state q_2 to compare the a's on tape 2 with the a's after the b on tape 1. If the same number of a's precede and follow the b, the computation halts in q_3 and accepts the input. The computation for strings without a b halt in q_1 and strings with more than one b in q_2. The computations for strings with with one b and an unequal number of leading and trailing a's also halt in q_2. Since every computation halts, M provides a decision procedure for membership in $\{a^iba^i \mid i \geq 0\}$ and consequently the language is recursive. □

A standard Turing machine is a multitape Turing machine with a single tape. Consequently, every recursively enumerable language is accepted by a multitape machine. We will show that the computations of a two-tape machine can be simulated by computations of a five-track machine. The argument can be generalized to show that any language accepted by a k-tape machine is accepted by a $2k + 1$-track machine. The equivalence of acceptance by multitrack and standard machines then allows us to conclude the following.

Theorem 8.6.1

A language L is accepted by a multitape Turing machine if, and only if, it is accepted by a standard Turing machine.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a two-tape machine. During a computation, the tape heads of a multitape machine are independently positioned on the two tapes.



The single tape head of a multitrack machine reads all the tracks of a fixed position. The five-track machine M′ is constructed to simulate the computations of M. Tracks 1 and 3 maintain the information stored on tapes 1 and 2 of the two-tape machine. Tracks 2 and 4 have a single nonblank square indicating the position of the tape heads of the multitape machine.

Track 5	#						
Track 4				X			
Track 3	a	b	b	c	c		
Track 2			X				
Track 1		b	a	c			

The initial action of the simulation in the multitrack machine is to write # in the leftmost position of track 5 and X in the leftmost positions of tracks 2 and 4. The remainder of the computation of the multitrack machine consists of a sequence of actions that simulate the transitions of the two-tape machine.

A transition of the two-tape machine is determined by the two symbols being scanned and the machine state. The simulation in the five-track machine records the symbols marked by each of the X's. The states are 8-tuples of the form $[s, q_i, x_1, x_2, y_1, y_2, d_1, d_2]$, where $q_i \in Q$; $x_i, y_i \in \Sigma \cup \{U\}$; and $d_i \in \{L, R, S, U\}$. The element s represents the status of the simulation of the transition of M. The symbol U, added to the tape alphabet and the set of directions, indicates that this item is unknown.

Let $\delta(q_i, x_1, x_2) = [q_j; y_1, d_1; y_2, d_2]$ be the applicable two-tape transition of M. M′ begins the simulation of the transition in the state $[f1, q_i, U, U, U, U, U, U]$. The following five actions simulate the transition of M in the multitrack machine.

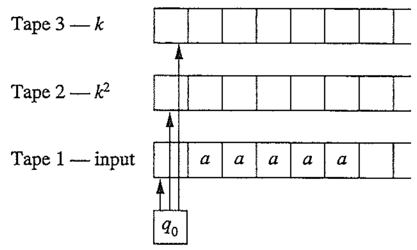
1. $f1$ (find first symbol): M′ moves to the right until it reads the X on track 2. State $[f1, q_i, x_1, U, U, U, U, U]$ is entered, where x_1 is the symbol in track 1 under the X. After recording the symbol on track 1 in the state, M′ returns to the initial position. The # on track 5 is used to reposition the tape head.
2. $f2$ (find second symbol): The same sequence of actions records the symbol beneath the X on track 4. M′ enters state $[f2, q_i, x_1, x_2, U, U, U, U]$, where x_2 is the symbol in track 3 under the X. The tape head is then returned to the initial position.
3. M′ enters the state $[p1, q_j, x_1, x_2, y_1, y_2, d_1, d_2]$, where the values q_j, y_1, y_2, d_1, and d_2 are obtained from the transition $\delta(q_i, x_1, x_2)$. This state contains the information needed to simulate the transition of the M.
4. $p1$ (print first symbol): M′ moves to the right to the X in track 2 and writes the symbol y_1 on track 1. The X on track 2 is moved in the direction designated by d_1. The machine then returns to the initial position.
5. $p2$ (print second symbol): M′ moves to the right to the X in track 4 and writes the symbol y_2 on track 3. The X on track 4 is moved in the direction designated by d_2.
6. The simulation of the transition $\delta(q_i, x_1, x_2) = [q_j; y_1, d_1; y_2, d_2]$ terminates by returning the tape head to the initial position to process the subsequent transition.

If $\delta(q_i, x_1, x_2)$ is undefined in the two-tape machine, the simulation halts after returning to the initial position following step 2. A state $[f2, q_i, x_1, y_1, U, U, U, U]$ is an accepting state of the multitrack machine M′ whenever q_i is an accepting state of M.

The next two examples illustrate the use of the additional tapes to store and manipulate data in a computation.

Example 8.6.2

The set $\{a^k \mid k \text{ is a perfect square}\}$ is a recursively enumerable language. The design of a three-tape machine that accepts this language is presented. Tape 1 contains the input string. The input is compared with a string of X's on tape 2 whose length is a perfect square. Tape 3 holds a string whose length is the square root of the string on tape 2. The initial configuration for a computation with input $aaaaa$ is



The values of k and k^2 are incremented until the length of the string on tape 2 is greater than or equal to the length of the input. A machine to perform these comparisons consists of the following actions.

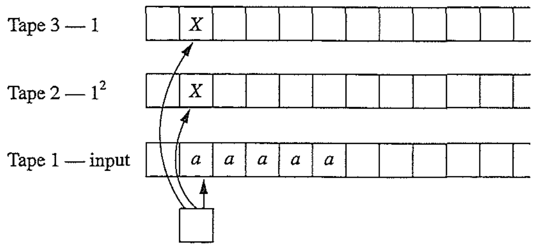
1. If the input is the null string, the computation halts in an accepting state. If not, tapes 2 and 3 are initialized by writing X in position one. The three tape heads are then moved to position one.
2. Tape 3 now contains a sequence of k X's and tape 2 contains k^2 X's. Simultaneously, the heads on tapes 1 and 2 move to the right while both heads scan nonblank squares. The head reading tape 3 remains at position one.
   a) If both heads simultaneously read a blank, the computation halts and the string is accepted.
   b) If tape head 1 reads a blank and tape head 2 an X, the computation halts and the string is rejected.
3. If neither of the halting conditions occur, the tapes are reconfigured for comparison with the next perfect square.
   a) An X is added to the right end of the string of X's on tape 2.
   b) Two copies of the string on tape 3 are added to the right end of the string on tape 2. This constructs a sequence of $(k + 1)^2$ X's on tape 2.

c) An X is added to the right end of the string of X's on tape 3. This constructs a sequence of $k+1$ X's on tape 3.

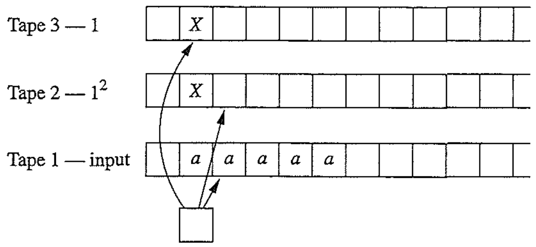
d) The tape heads are then repositioned at position one of their respective tapes.

4. The computation continues with step 2.

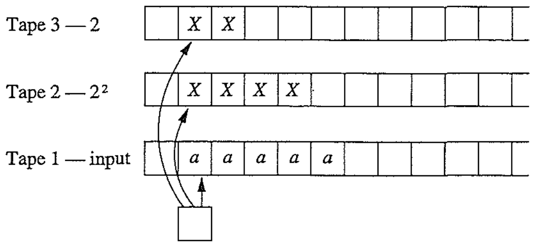
Tracing the computation for the input string $aaaaa$, step 1 produces the configuration



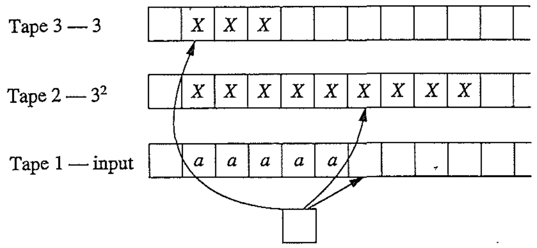
The simultaneous left-to-right movement of tape heads 1 and 2 halts when tape head 2 scans the blank in position two.



Part (c) of step 3 reformats tapes 2 and 3 so that the input string can be compared with the next perfect square.



Another iteration of step 2 halts and rejects the input.



A machine that performs the preceding computation is defined by the following transitions:

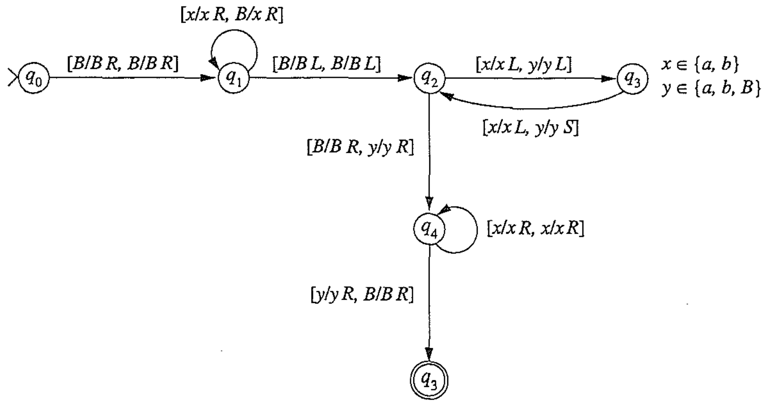
$$
\begin{array}{lll}
\delta(q_0, B, B, B) = [q_1; B, R; B, R; B, R] & & \text{(initialize the tape)} \\
\delta(q_1, a, B, B) = [q_2; a, S; X, S; X, S] & & \\
\\
\delta(q_2, a, X, X) = [q_2; a, R; X, R; X, S] & & \text{(compare strings on tapes 1 and 2)} \\
\delta(q_2, B, B, X) = [q_3; B, S; B, S; X, S] & & \text{(accept)} \\
\delta(q_2, a, B, X) = [q_4; a, S; X, R; X, S] & & \\
\\
\delta(q_4, a, B, X) = [q_5; a, S; X, R; X, S] & & \text{(rewrite tapes 2 and 3)} \\
\delta(q_4, a, B, B) = [q_6; a, L; B, L; X, L] & & \\
\delta(q_5, a, B, X) = [q_4; a, S; X, R; X, R] & & \\
\\
\delta(q_6, a, X, X) = [q_6; a, L; X, L; X, L] & & \text{(reposition tape heads)} \\
\delta(q_6, a, X, B) = [q_6; a, L; X, L; B, S] & & \\
\delta(q_6, a, B, B) = [q_6; a, L; B, S; B, S] & & \\
\delta(q_6, B, X, B) = [q_6; B, S; X, L; B, S] & & \\
\delta(q_6, B, B, B) = [q_2; B, R; B, R; B, R]. & & \text{(repeat comparison cycle)}
\end{array}
$$

The accepting states are q_1 and q_3. The null string is accepted in q_1, and strings a^k, where k is a perfect square greater than zero, are accepted in q_3.

Since the machine designed above halts for all input strings, we have shown that the language $\{a^k \mid k \text{ is a perfect square}\}$ is not only recursively enumerable but also recursive. □

Example 8.6.3

The two-tape Turing machine



accepts the language $\{uu \mid u \in \{a, b\}^*\}$. The symbols x and y on the labels of the arcs represent an arbitrary input symbol.

The computation begins by making a copy of the input on tape 2. When this is complete, both tape heads are to the immediate right of the input. The tape heads now move back to the left, with tape head 1 moving two squares for every one square that tape head 2 moves. If the computation halts in q_3, the input string has odd length and is rejected. The loop in q_4 compares the first half of the input with the second; if they match, the string is accepted in state q_5. □

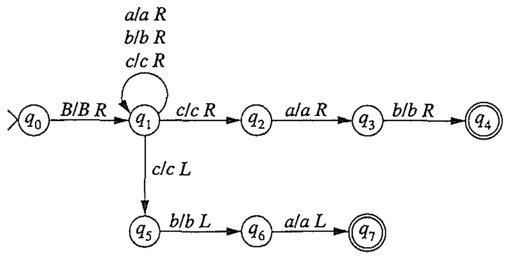
8.7 Nondeterministic Turing Machines

A nondeterministic Turing machine may specify any finite number of transitions for a given configuration. The components of a nondeterministic machine, with the exception of the transition function, are identical to those of the standard Turing machine. Transitions in a nondeterministic machine are defined by a function from $Q \times \Gamma$ to subsets of $Q \times \Gamma \times \{L, R\}$.

Whenever the transition function indicates that more than one action is possible, a computation arbitrarily chooses one of the transitions. An input string is accepted by a nondeterministic machine if there is at least one computation that terminates in an accepting state. The existence of other computations that halt in nonaccepting states or fail to halt altogether is irrelevant. As usual, the language of a machine is the set of strings accepted by the machine.

Example 8.7.1

The nondeterministic Turing machine



accepts strings containing a c preceded or followed by ab. The machine processes the input in state q_1 until a c is encountered. When this occurs, the computation may continue in state q_1, enter state q_2 to determine if the c is followed by ab, or enter q_5 to determine if the c is preceded by ab. In the language of nondeterminism, the computation chooses a c and then chooses one of the conditions to check. □

The machine constructed in Example 8.7.1 accepts strings by final state. As with standard machines, acceptance in nondeterministic Turing machines can be defined by final state or by halting alone. A nondeterministic machine accepts a string u by halting if there is at least one computation that halts normally when run with u. Exercise 24 establishes that these alternative approaches accept the same languages.

Nondeterminism does not increase the capabilities of Turing computation; the languages accepted by nondeterministic machines are precisely those accepted by deterministic machines. To accomplish the transformation of a nondeterministic Turing machine to an equivalent deterministic machine, we show that the multiple computations for a single input string can by sequentially generated and examined.

A nondeterministic Turing machine may produce multiple computations for a single input string. The computations can be systematically produced by ordering the alternative transitions for a state, symbol pair. Let n be the maximum number of transitions defined for any combination of state and tape symbol. The numbering assumes that $\delta(q_i, x)$ defines n, not necessarily distinct, transitions for every state q_i and tape symbol x with $\delta(q_i, x) \neq \emptyset$. If the transition function defines fewer than n transitions, one transition is assigned several numbers to complete the ordering.

A sequence $(m_1, \ldots, m_i, \ldots, m_k)$, where each m_i is a number from 1 to n, defines a unique computation in the nondeterministic machine. The computation associated with this sequence consists of k or fewer transitions. The jth transition is determined by the state, the tape symbol scanned, and m_j, the jth number in the sequence. Assume the $j-1$st transition leaves the machine in state q_i scanning x. If $\delta(q_i, x) = \emptyset$, the computation halts. Otherwise, the machine executes the transition in $\delta(q_i, x)$ numbered m_j.

TABLE 8.1 Ordering of Transitions

State	Symbol	Transition	State	Symbol	Transition
q_0	B	$1q_1, B, R$	q_2	a	$1q_3, a, R$
		$2q_1, B, R$			$2q_3, a, R$
		$3q_1, B, R$			$3q_3, a, R$
q_1	a	$1q_1, a, R$	q_3	b	$1q_4, b, R$
		$2q_1, a, R$			$2q_4, b, R$
		$3q_1, a, R$			$3q_4, b, R$
q_1	b	$1q_1, b, R$	q_5	b	$1q_6, b, L$
		$2q_1, b, R$			$2q_6, b, L$
		$3q_1, b, R$			$3q_6, b, L$
q_1	c	$1q_1, c, R$	q_6	a	$1q_7, a, L$
		$2q_2, c, R$			$2q_7, a, L$
		$3q_5, c, L$			$3q_7, a, L$

The transitions of the nondeterministic machine in Example 8.7.1 can be ordered as shown in Table 8.7.1. The computations defined by the input string $acab$ and the sequences (1, 1, 1, 1, 1), (1, 1, 2, 1, 1), and (2, 2, 3, 3, 1) are

$$\begin{array}{lll}
q_0BacabB\ 1 & q_0BacabB\ 1 & q_0BacabB\ 2 \\
\vdash Bq_1acabB\ 1 & \vdash Bq_1acabB\ 1 & \vdash Bq_1acabB\ 2 \\
\vdash Baq_1cabB\ 1 & \vdash Baq_1cabB\ 2 & \vdash Baq_1cabB\ 3 \\
\vdash Bacq_1abB\ 1 & \vdash Bacq_2abB\ 1 & \vdash Bq_5acabB. \\
\vdash Bacaq_1bB\ 1 & \vdash Bacaq_3bB\ 1 & \\
\vdash Bacabq_1B & \vdash Bacabq_4B &
\end{array}$$

The number on the right designates the transition used to obtain the subsequent configuration. The third computation terminates prematurely since no transition is defined when the machine is in state q_5 scanning an a. The string $acab$ is accepted since the computation defined by (1, 1, 2, 1, 1) terminates in state q_4.

Using the ability to sequentially produce the computations of a nondeterministic machine, we will now show that every nondeterministic Turing machine can be transformed into an equivalent deterministic machine. Let M = (Q, Σ, Γ, δ, q_0) be a nondeterministic machine that accepts strings by halting. We choose acceptance by halting because this reduces the number of potential outcomes of a computation from three to two—a

computation halts (and accepts) or does not halt. Thus we have fewer cases to consider in the proof. Assume that the transitions of M have been numbered according to the previous scheme, with n the maximum number of transitions for a state, symbol pair. A deterministic three-tape machine M′ is constructed to accept the language of M. Acceptance in M′ is also defined by halting.

The machine M′ is built to simulate the computations of M. The correspondence between sequences $(m_1, \ldots, m_k)$ and computations of M′ ensures that all possible computations are examined. The role of the three tapes of M′ are

Tape 1: stores the input string;

Tape 2: simulates the tape of M;

Tape 3: holds sequences of the form $(m_1, \ldots, m_k)$ to guide the simulation.

A computation in M′ consists of the actions:

1. A sequence of integers $(m_1, \ldots, m_k)$ from 1 to n is written on tape 3.
2. The input string on tape 1 is copied to the standard input position on tape 2.
3. The computation of M defined by the sequence on tape 3 is simulated on tape 2.
4. If the simulation halts prior to executing k transitions, the computation of M′ halts and accepts the input.
5. If the computation did not halt in step 3, the next sequence is generated on tape 3 and the computation continues at step 2.

The simulation is guided by the sequence of values on tape 3. The deterministic Turing machine in Figure 8.1 generates all finite-length sequences of integers from 1 to n, where the symbols *1, 2, . . . , n* are individual tape symbols. Sequences of length 1 are generated in numeric order, followed by sequences of length 2, length 3, and so on. A computation begins in state q_0 at position zero. When the tape head returns to position zero the tape contains the next sequence of values. The notation i/i abbreviates *1/1, 2/2, . . . , n/n*.

Using the exhaustive generation of numeric sequences, we now construct a deterministic three-tape machine M′ that accepts L(M). A computation of the machine M′ interweaves the generation of the sequences on tape 3 with the simulation of M on tape 2. M′ halts when the sequence on tape 3 defines a computation that halts in M. Recall that both M and M′ accept by halting.

Let Σ and Γ be the input and tape alphabets of M. The alphabets of M′ are

$$\Sigma_{M'} = \Sigma$$
$$\Gamma_{M'} = \{x, \#x \mid x \in \Gamma\} \cup \{1, \ldots, n\}.$$

Symbols of the form $\#x$ represent tape symbol x and are used to mark the leftmost square on tape 2 during the simulation of the computation of M. The transitions of M′ are naturally grouped by their function. States labeled $q_{s,j}$ are used in the generation of a sequence on tape

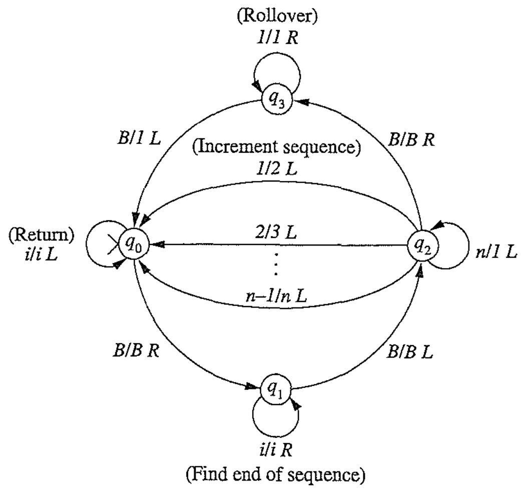


FIGURE 8.1 Turing machine generating $\{1, 2, \ldots, n\}^+$.

3. These transitions are obtained from the machine in Figure 8.1. The tape heads reading tapes 1 and 2 remain stationary during this operation.

$$\delta(q_{s,0}, B, B, B) = [q_{s,1}; B, S; B, S; B, R]$$
$$\delta(q_{s,1}, B, B, t) = [q_{s,1}; B, S; B, S; i, R] \qquad t = 1, \ldots, n$$
$$\delta(q_{s,1}, B, B, B) = [q_{s,2}; B, S; B, S; B, L]$$
$$\delta(q_{s,2}, B, B, n) = [q_{s,2}; B, S; B, S; 1, L]$$
$$\delta(q_{s,2}, B, B, t-1) = [q_{s,4}; B, S; B, S; t, L] \qquad t = 1, \ldots, n-1$$
$$\delta(q_{s,2}, B, B, B) = [q_{s,3}; B, S; B, S; B, R]$$
$$\delta(q_{s,3}, B, B, 1) = [q_{s,3}; B, S; B, S; 1, R]$$
$$\delta(q_{s,3}, B, B, B) = [q_{s,4}; B, S; B, S; 1, L]$$
$$\delta(q_{s,4}, B, B, t) = [q_{s,4}; B, S; B, S; t, L] \qquad t = 1, \ldots, n$$
$$\delta(q_{s,4}, B, B, B) = [q_{c,0}; B, S; B, S; B, S]$$

The next step is to make a copy of the input on tape 2. The symbol $\#B$ is written in position zero to designate the left boundary of the tape.

$$\delta(q_{c,0}, B, B, B) = [q_{c,1}; B, R; \#B, R; B, S]$$
$$\delta(q_{c,1}, x, B, B) = [q_{c,1}; x, R; x, R; B, S] \qquad \text{for all } x \in \Gamma - \{B\}$$
$$\delta(q_{c,1}, B, B, B) = [q_{c,2}; B, L; B, L; B, S]$$
$$\delta(q_{c,2}, x, x, B) = [q_{c,2}; x, L; x, L; B, S] \qquad \text{for all } x \in \Gamma$$
$$\delta(q_{c,2}, B, \#B, B) = [q_0; B, S; \#B, S; B, R]$$

The transitions that simulate the computation of M on tape 2 of M′ are obtained directly from the transitions of M. If $\delta(q_i, x) = [q_j, y, d]$ is a transition of M assigned the number t in the ordering, then

$$\delta(q_i, B, x, t) = [q_j; B, S; y, d; t, R]$$
$$\delta(q_i, B, \#x, t) = [q_j; B, S; \#y, d; t, R]$$

are the corresponding transitions of M′.

If the sequence on tape 3 consists of k numbers, the simulation processes at most k transitions. The computation of M′ halts if the computation of M specified by the sequence on tape 3 halts. When a blank is read on tape 3, the simulation has processed all of the transitions designated by the current sequence. Before the next sequence is processed, the result of the simulated computation must be erased from tape 2. To accomplish this, the tape heads on tapes 2 and 3 are repositioned at the leftmost position in state $q_{e,0}$ and $q_{e,1}$, respectively. The head on tape 2 then moves to the right, erasing the tape.

$$\delta(q_i, B, x, B) = [q_{e,0}; B, S; x, S; B, S] \qquad \text{for all } x \in \Gamma$$
$$\delta(q_i, B, \#x, B) = [q_{e,0}; B, S; \#x, S; B, S] \qquad \text{for all } x \in \Gamma$$
$$\delta(q_{e,0}, B, x, B) = [q_{e,0}; B, S; x, L; B, S] \qquad \text{for all } x \in \Gamma$$
$$\delta(q_{e,0}, B, \#x, B) = [q_{e,1}; B, S; B, S; B, L] \qquad \text{for all} x \in \Gamma$$
$$\delta(q_{e,1}, B, B, t) = [q_{e,1}; B, S; B, S; t, L] \qquad t = 1, \ldots, n$$
$$\delta(q_{e,1}, B, B, B) = [q_{e,2}; B, S; B, R; B, R]$$
$$\delta(q_{e,2}, B, x, i) = [q_{e,2}; B, S; B, R; i, R] \qquad \text{for all } x \in \Gamma \text{ and } i = 1, \ldots, n$$
$$\delta(q_{e,2}, B, B, B) = [q_{e,3}; B, S; B, L; B, L]$$
$$\delta(q_{e,3}, B, B, t) = [q_{e,3}; B, S; B, L; t, L] \qquad t = 1, \ldots, n$$
$$\delta(q_{e,3}, B, B, B) = [q_{s,0}; B, S; B, S; B, S]$$

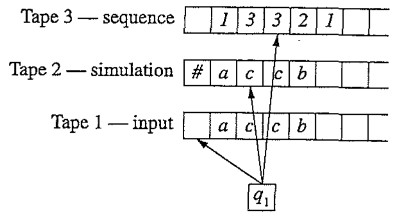
When a blank is read on tape 3, the entire segment of the tape that may have been accessed during the simulated computation has been erased. M′ then returns the tape heads to their initial position and enters $q_{s,0}$ to generate the next sequence and continue the simulation of computations.

The process of simulating computations of M, steps 2 through 5 of the algorithm, continues until a sequence of numbers is generated on tape 3 that defines a halting computation. The simulation of this computation causes M′ to halt, accepting the input. If the input string is not in L(M), the cycle of sequence generation and computation simulation in M′ will continue indefinitely.

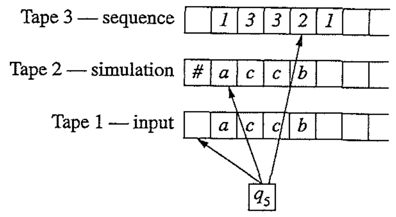
The actions of the deterministic machine constructed following the preceding strategy are illustrated using the nondeterministic machine from Example 8.7.1 and the numbering of the transitions in Table 8.7.1. The first three transitions of the computation M defined by the sequence (1, 3, 3, 2, 1) and input string *accb* are

$$\begin{aligned} & q_0BaccbB \quad 1 \\ \vdash\ & Bq_1accbB \quad 3 \\ \vdash\ & Baq_1ccbB \quad 3 \\ \vdash\ & Bq_5accbB. \end{aligned}$$

The sequence *1, 3, 3, 2, 1* that designates the particular computation of M is written on tape 3 of M′. The configuration of the three-tape machine M′ prior to the execution of the third transition of M is



Transition 3 from state q_1 with M scanning a c causes the machine to print c, enter state q_5, and move to the left. This transition is simulated in M′ by the transition $\delta'(q_1, B, c, 3) = [q_5; B, S; c, L; 3, R]$. The transition of M′ alters tape 2 as prescribed by the transition of M and moves the head on tape 3 to designate the number of the subsequent transition.

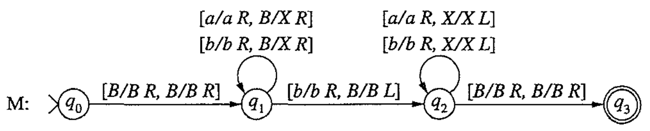


Nondeterministic Turing machines can be defined with a multitrack tape, two-way tape, or multiple tapes. Machines defined using these alternative configurations can also be shown to accept precisely the recursively enumerable languages.

Like their deterministic counterparts, nondeterministic machines that accept by final state can be used to show that a language is recursive. If every computation in the nondeterministic machine halts, so will every computation in the equivalent deterministic machine (Exerecise 23).

Example 8.7.2

The two-tape nondeterministic machine



accepts the set of strings over $\{a, b\}$ with a b in the middle. The transition from state q_1 to q_2 on reading a b on tape 1 represents a guess that the b is in the middle of the input. The loop in state q_2 compares the number of symbols following the b to the number preceding it. If a string is in L(M), one computation will enter q_3 upon reading the middle b and accept the input. The computations for strings with no b's halt in q_1, and strings that do not have a b in the middle halt in either q_1 or q_2. Since M halts for all inputs, L(M) is recursive. □

The next example illustrates the flexibility afforded by the combination of multitape machines and the guess and check strategy of nondeterminism.

Example 8.7.3

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a standard Turing machine that accepts a language L. We will design a two-tape nondeterministic machine M′ that accepts strings over Σ^* that have a substring of length two or more in L. That is, $L(M') = \{u \mid u = xyz,\ length(y) \geq 2 \text{ and } y \in L\}$. A computation of M′ with input u consists of the following steps:

1. Reading the input on tape 1 and nondeterministically choosing a position in the string to begin copying to tape 2;
2. Copying from tape 1 to tape 2 and nondeterministically choosing a position to stop copying;
3. Simulating the computation of M on tape 2.

The first two steps constitute the nondeterministic guess of a substring of u and the third checks whether the substring is in L.

The states of M′ are $Q \cup \{q_s, q_b, q_c, q_d, q_e\}$ with start state q_s. The alphabets and final states are the same as those of M. The transitions for steps 1 and 2 use states q_s, q_b, q_c, q_d, and q_e.

$$\delta'(q_s, B, B) = \{ [q_b; B, R; B, R]\}$$
$$\delta'(q_b, x, B) = \{ [q_b; x, R; B, S], [q_c; x, R; x, R]\} \quad \text{for all } x \in \Sigma$$
$$\delta'(q_c, x, B) = \{ [q_c; x, R; x, R], [q_d; x, R; x, R]\} \quad \text{for all } x \in \Sigma$$
$$\delta'(q_d, x, B) = \{ [q_d; x, R; B, S]\} \quad \text{for all } x \in \Sigma$$
$$\delta'(q_d, B, B) = \{ [q_e; B, S; B, L]\}$$
$$\delta'(q_e, B, x) = \{ [q_e; B, S; x, L]\} \quad \text{for all } x \in \Sigma$$
$$\delta'(q_e, B, B) = \{ [q_0; B, S; B, S]\}$$

The transition from q_b to q_c initiates the copying of a substring of u onto tape 2. The second transition in q_c completes the selection of the substring. The tape head on tape 1 is moved to the blank following the input in q_d, and the head on tape 2 is returned to position zero in q_e.

After the nondeterministic selection of a substring, the transitions of M are run on tape 2 to check whether the "guessed" substring is in L. The transitions for this part of the computation are obtained directly from δ, the transition function of M:

$$\delta'(q_i, B, x) = \{ [q_j; B, S; y, d]\} \quad \text{whenever } \delta(q_i, x) = [q_j, y, d] \text{ is a transition of M.}$$

The tape head reading tape 1 remains stationary while the computation of M is run on tape 2. □

8.8 Turing Machines as Language Enumerators

In the preceding sections Turing machines have been formulated as language acceptors: A machine is provided with an input string, and the result of the computation indicates the acceptability of the input. Turing machines may also be designed to enumerate a language. The computation of such a machine sequentially produces an exhaustive listing of the elements of the language. An enumerating machine has no input; its computation continues until it has generated every string in the language.

Like Turing machines that accept languages, there are a number of equivalent ways to define an enumerating machine. We will use a k-tape deterministic machine, $k \geq 2$, as the underlying Turing machine model in the definition of enumerating machines. The first tape is the output tape and the remaining tapes are work tapes. A special tape symbol # is used on the output tape to separate the elements of the language that are generated during the computation.

The machines considered in this section perform two distinct tasks, acceptance and enumeration. To distinguish them, a machine that accepts a language will be denoted M while an enumerating machine will be denoted E.

Definition 8.8.1

A k-tape Turing machine E = (Q, Σ, Γ, δ, q_0) **enumerates** the language L if

i) the computation begins with all tapes blank;

ii) with each transition, the tape head on tape 1 (the output tape) remains stationary or moves to the right;

iii) at any point in the computation, the nonblank portion of tape 1 has the form

$$B\#u_1\#u_2\# \ldots \#u_k\# \qquad \text{or} \qquad B\#u_1\#u_2\# \ldots \#u_k\#v,$$

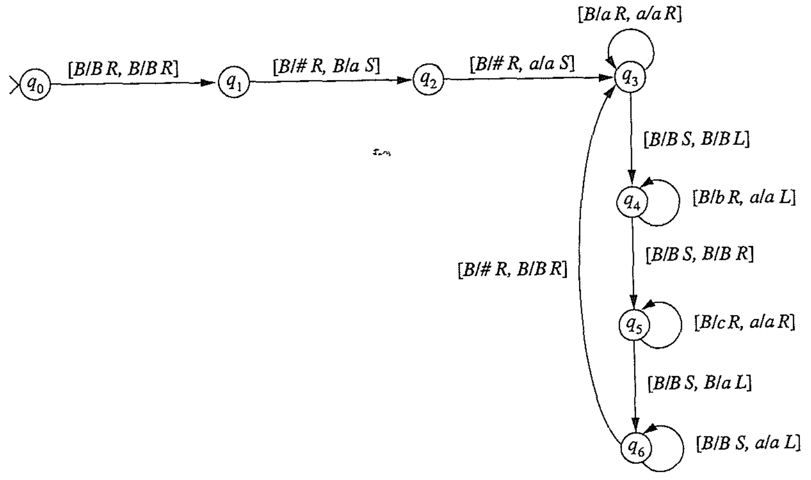
where $u_i \in$ L and $v \in \Sigma^*$;

iv) a string u will be written on tape 1 preceded and followed by # if, and only if, $u \in L$.

The last condition indicates that the computation of a machine E that enumerates L eventually writes every string in L on the output tape. Since all of the elements of a language must be produced, a computation enumerating an infinite language will never halt. The definition does not require a machine to halt even if it is enumerating a finite language. Such a machine may continue indefinitely after writing the last element on the output tape.

Example 8.8.1

The machine E enumerates the language L = $\{a^i b^i c^i \mid i \geq 0\}$. A Turing machine accepting this language was given in Example 8.2.2.



The computation of E begins by writing ## on the output tape, indicating that λ ∈ L. Simultaneously, an a is written in position one of tape 2, with the head returning to tape position zero. At this point, E enters the nonterminating loop described by the following actions.

1. The tape heads move to the right, writing an a on the output tape for every a on the work tape.
2. The head on the work tape then moves right to left through the a's and a b is written on the output tape for each a.
3. The tape heads move to the right, writing a c on the output tape for every a on the work tape.
4. An a is added to the end of the work tape and the head is moved to position one.
5. A # is written on the output tape.

After a string is completed on the output tape, the work tape contains the information required to construct the next string in the enumeration. □

The definition of enumeration requires that each string in the language appear on the output tape but permits a string to appear multiple times. Theorem 8.8.2 shows that any language that is enumerated by a Turing machine can be enumerated by one in which each string is written only once on the output tape.

Theorem 8.8.2

Let L be a language enumerated by a Turing machine E. Then there is a Turing machine E′ that enumerates L and each string in L appears only once on the output tape of E′.

Proof. Assume E is a k-tape machine enumerating L. A $(k + 1)$-tape machine E′ that satisfies the "single output" requirement can be built from the enumerating machine E. Intuitively, E is a submachine of E′ that produces strings to be considered for output by E′. The output tape of E′ is the additional tape added to E, while the output tape of E becomes a work tape for E′. For convenience, we call tape 1 the output tape of E′. Tapes 2, 3, . . . , $k + 1$ are used to simulate E, with tape 2 being the output tape of the simulation. The actions of E′ consist of the following sequence of steps:

1. The computation begins by simulating the actions of E on tapes 2, 3, . . . , $k + 1$.
2. When the simulation of E writes #u# on tape 2, E′ initiates a search procedure to see if u already occurs on tape 2.
3. If u is not on tape 2, it is added to the output tape of E′.
4. The simulation of E is restarted to produce the next string.

Searching for another occurrence of u requires the tape head to examine the entire nonblank portion of tape 2. Since tape 2 is not the output tape of E′, the restriction that the tape head on the output tape never move to the left is not violated. ■

Theorem 8.8.2 justifies the selection of the term *enumerate* to describe this type of computation. The computation sequentially and exhaustively lists the strings in the

language. The order in which the strings are produced defines a mapping from an initial sequence of the natural numbers onto L. Thus we can talk about the zeroth string in L, the first string in L, and so on. This ordering is machine-specific; another enumerating machine may produce a completely different ordering.

Turing machine computations now have two distinct ways of defining a language: by acceptance and by enumeration. We show that these two approaches produce the same languages.

Lemma 8.8.3

If L is enumerated by a Turing machine, then L is recursively enumerable.

Proof. Assume that L is enumerated by a k-tape Turing machine E. A $(k + 1)$-tape machine M accepting L can be constructed from E. The additional tape of M is the input tape; the remaining k tapes allow M to simulate the computation of E. The computation of M begins with a string u on its input tape. Next M simulates the computation of E. When the simulation of E writes #, a string $w \in$ L has been generated. M then compares u with w and accepts u if $u = w$. Otherwise, the simulation of E is used to generate another string from L and the comparison cycle is repeated. If $u \in$ L, it will eventually be produced by E and consequently accepted by M. ■

The proof that any recursively enumerable language L can be enumerated is complicated by the fact that a Turing machine M that accepts L need not halt for every input string. A straightforward approach to enumerating L would be to build an enumerating machine that simulates the computations of M to determine whether a string should be written on the output tape. The actions of such a machine would be to

1. Generate a string $u \in \Sigma^*$.
2. Simulate the computation of M with input u.
3. If M accepts, write u on the output tape.
4. Continue at step 1 until all strings in Σ^* have been tested.

The generate-and-test approach requires the ability to generate the entire set of strings over Σ for testing. This presents no difficulty, as we will see later. However, step 2 of this naive approach causes it to fail. It is possible to produce a string u for which the computation of M does not terminate. In this case, no strings after u will be generated and tested for membership in L.

To construct an enumerating machine, we first introduce the lexicographical ordering of the input strings and provide a strategy to ensure that the enumerating machine E will check every string in Σ^*. The lexicographical ordering of the set of strings over a nonempty alphabet Σ defines a one-to-one correspondence between the natural numbers and the strings in Σ^*.

Definition 8.8.4

Let $\Sigma = \{a_1, \ldots, a_n\}$ be an alphabet. The lexicographical ordering lo of Σ^* is defined recursively as follows:

i) Basis: $lo(\lambda) = 0$, $lo(a_i) = i$ for $i = 1, 2, \ldots, n$.

ii) Recursive step: $lo(a_i u) = lo(u) + i \cdot n^{length(u)}$.

The values assigned by the function lo define a total ordering on the set Σ^*. Strings u and v are said to satisfy $u < v$, $u = v$, and $u > v$ if $lo(u) < lo(v)$, $lo(u) = lo(v)$, and $lo(u) > lo(v)$, respectively.

Example 8.8.2

Let $\Sigma = \{a, b, c\}$ and let a, b, and c be assigned the values 1, 2, and 3, respectively. The lexicographical ordering produces

$lo(\lambda) = 0$	$lo(a) = 1$	$lo(aa) = 4$	$lo(ba) = 7$	$lo(ca) = 10$	$lo(aaa) = 13$
	$lo(b) = 2$	$lo(ab) = 5$	$lo(bb) = 8$	$lo(cb) = 11$	$lo(aab) = 14$
	$lo(c) = 3$	$lo(ac) = 6$	$lo(bc) = 9$	$lo(cc) = 12$	$lo(aac) = 15.$

□

Lemma 8.8.5

For any alphabet Σ, there is a Turing machine E_{Σ^*} that enumerates Σ^* in lexicographical order.

The construction of a machine that enumerates the set of strings over the alphabet $\{0, 1\}$ is left as an exercise.

The lexicographical ordering and a dovetailing technique are used to show that a recursively enumerable language L can be enumerated by a Turing machine. Let M be a Turing machine that accepts L. Recall that M need not halt for all input strings. The lexicographical ordering produces a listing $u_0 = \lambda, u_1, u_2, u_3, \ldots$ of the strings of Σ^*. A two-dimensional table is constructed whose columns are labeled by the strings of Σ^* and rows by the natural numbers.

$$\begin{array}{c|cccc}
\vdots & \vdots & \vdots & \vdots & \\
3 & [\lambda,3] & [u_1,3] & [u_2,3] & \ldots \\
2 & [\lambda,2] & [u_1,2] & [u_2,2] & \ldots \\
1 & [\lambda,1] & [u_1,1] & [u_2,1] & \ldots \\
0 & [\lambda,0] & [u_1,0] & [u_2,0] & \ldots \\
\hline
 & \lambda & u_1 & u_2 & \ldots
\end{array}$$

The $[i, j]$ entry in this table is interpreted to mean "run machine M on input u_i for j steps." Using the technique presented in Example 1.4.2, the ordered pairs in the table can be enumerated in a "diagonal by diagonal" manner (Exercise 33).

The machine E built to enumerate L interleaves the enumeration of the ordered pairs with the computations of M. The computation of E is a loop that consists of the following steps:

1. Generate an ordered pair $[i, j]$.
2. Run a simulation of M with input u_i for j transitions or until the simulation halts.
3. If M accepts, write u_i on the output tape.
4. Continue with step 1.

If $u_i \in$ L, then the computation of M with input u_i halts and accepts after k transitions, for some number k. Thus u_i will be written to the output tape of E when the ordered pair $[i, k]$ is processed. The second element in an ordered pair $[i, j]$ ensures that the simulation M is terminated after j steps. Consequently, no nonterminating computations are allowed and each string in Σ^* is examined.

Combining the preceding argument with Lemma 8.8.3 yields

Theorem 8.8.6

A language is recursively enumerable if, and only if, it can be enumerated by a Turing machine.

A Turing machine that accepts a recursively enumerable language halts and accepts every string in the language but is not required to halt when an input is a string that is not in the language. A language L is recursive if it is accepted by a machine that halts for all input. Since every computation halts, such a machine provides a decision procedure for determining membership in L. The family of recursive languages can also be defined by enumerating Turing machines.

The definition of an enumerating Turing machine does not impose any restrictions on the order in which the strings of the language are generated. Requiring the strings to be generated in a predetermined computable order provides the additional information needed to obtain negative answers to the membership question. Intuitively, the strategy to determine whether a string u is in the language is to begin the enumerating machine and compare u with each string that is produced. Eventually either u is output, in which case it is accepted, or some string beyond u in the ordering is generated. Since the output strings are produced according to the ordering, u has been passed and is not in the language. Thus we are able to decide membership, and the language is recursive. Theorem 8.8.7 shows that recursive languages may be characterized as the family of languages whose elements can be enumerated in order.

Theorem 8.8.7

L is recursive if, and only if, L can be enumerated in lexicographical order.

Proof. We first show that every recursive language can be enumerated in lexicographical order. Let L be a recursive language over an alphabet Σ. Then it is accepted by some machine M that halts for all input strings. A machine E that enumerates L in lexicographical order can be constructed from M and the machine E_{Σ^*} that enumerates Σ^* in lexicographical order. The machine E is a hybrid, interleaving the computations of M and E_{Σ^*}. The computation of E consists of the following loop:

1. The machine E_{Σ^*} is run, producing a string $u \in \Sigma^*$.
2. M is run with input u.
3. If M accepts u, u is written on the output tape of E.
4. The generate-and-test loop continues with step 1.

Since M halts for all inputs, E cannot enter a nonterminating computation in step 2. Thus each string $u \in \Sigma^*$ will be generated and tested for membership in L.

Now we show that any language L that can be enumerated in lexicographical order is recursive. This proof is divided into two cases based on the cardinality of L.

Case 1: L is finite. Then L is recursive since every finite language is recursive.

Case 2: L is infinite. The argument is similar to that given in Theorem 8.8.2 except that the ordering is used to terminate the computation. As before, a $(k+1)$-tape machine M accepting L can be constructed from a k-tape machine E that enumerates L in lexicographical order. The additional tape of M is the input tape; the remaining k tapes allow M to simulate the computations of E. The ordering of the strings produced by E provides the information needed to halt M when the input is not in the language. The computation of M begins with a string u on its input tape. Next M simulates the computation of E. When the simulation produces a string w, M compares u with w. If $u = w$, then M halts and accepts. If w is greater than u in the ordering, M halts rejecting the input. Finally, if w is less than u in the ordering, then the simulation of E is restarted to produce another element of L and the comparison cycle is repeated. ■

Exercises

1. Let M be the Turing machine defined by

δ	B	a	b	c
q_0	q_1, B, R			
q_1	q_2, B, L	q_1, a, R	q_1, c, R	q_1, c, R
q_2		q_2, c, L		q_2, b, L

a) Trace the computation for the input string $aabca$.
b) Trace the computation for the input string $bcbc$.

c) Give the state diagram of M.
d) Describe the result of a computation in M.

2. Let M be the Turing machine defined by

δ	B	a	b	c
q_0	q_1, B, R			
q_1	q_1, B, R	q_1, a, R	q_1, b, R	q_2, c, L
q_2		q_2, b, L	q_2, a, L	

a) Trace the computation for the input string $abcab$.
b) Trace the first six transitions of the computation for the input string $abab$.
c) Give the state diagram of M.
d) Describe the result of a computation in M.

3. Construct a Turing machine with input alphabet $\{a, b\}$ to perform each of the following operations. Note that the tape head is scanning position zero in state q_f whenever a computation terminates.
a) Move the input one space to the right. Input configuration q_0BuB, result q_fBBuB.
b) Concatenate a copy of the reversed input string to the input. Input configuration q_0BuB, result q_fBuu^RB.
* c) Insert a blank between each of the input symbols. For example, input configuration q_0BabaB, result $q_fBaBbBaB$.
d) Erase the b's from the input. For example, input configuration $q_0BbabaababB$, result $q_fBaaaaB$.

4. Construct a Turing machine with input alphabet $\{a, b, c\}$ that accepts strings in which the first c is preceded by the substring aaa. A string must contain a c to be accepted by the machine.

5. Construct a Turing machine with input alphabet $\{a, b\}$ to accept each of the following languages by final state.
a) $\{a^ib^j \mid i \geq 0,\ j \geq i\}$
b) $\{a^ib^ja^ib^j \mid i,\ j > 0\}$
c) Strings with the same number of a's and b's
d) $\{uu^R \mid u \in \{a, b\}^*\}$
e) $\{uu \mid u \in \{a, b\}^*\}$

6. Modify your solution to Exercise 5(a) to obtain a Turing machine that accepts the language $\{a^ib^j \mid i \geq 0,\ j \geq i\}$ by halting.

7. An alternative method of acceptance by final state can be defined as follows: A string u is accepted by a Turing machine M if the computation of M with input u enters

(but does not necessarily terminate in) a final state. With this definition, a string may be accepted even though the computation of the machine does not terminate. Prove that the languages accepted by this definition are precisely the recursively enumerable languages.

8. The transitions of a one-tape deterministic Turing machine may be defined by a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R, S\}$, where S indicates that the tape head remains stationary. Prove that machines defined in this manner accept precisely the recursively enumerable languages.

9. An **atomic** Turing machine is one in which every transition consists of a change of state and one other action. The transition may write on the tape or move the tape head, but not both. Prove that the atomic Turing machines accept precisely the recursively enumerable languages.

* 10. A **context-sensitive** Turing machine is one in which the applicability of a transition is determined not only by the symbol scanned but also by the symbol in the tape square to the right of the tape head. A transition has the form

$$\delta(q_i, xy) = [q_j, z, d] \qquad x, y, z \in \Gamma;\ d \in \{L, R\}.$$

When the machine is in state q_i scanning an x, the transition may be applied only when the tape position to the immediate right of the tape head contains a y. In this case the x is replaced by z, the machine enters state q_j, and the tape head moves in direction d.

a) Let M be a standard Turing machine. Define a context-sensitive Turing machine M′ that accepts L(M). *Hint:* Define the transition function of M′ from that of M.

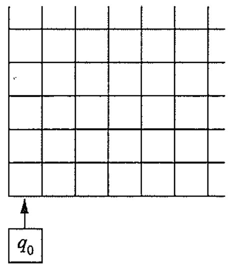
b) Let $\delta(q_i, xy) = [q_j, z, d]$ be a context-sensitive transition. Show that the result of the application of this transition can be obtained by a sequence of standard Turing machine transitions. You must consider the case both when transition $\delta(q_i, xy)$ is applicable and when it isn't.

c) Use parts (a) and (b) to conclude that context-sensitive machines accept precisely the recursively enumerable languages.

11. Prove that every recursively enumerable language is accepted by a Turing machine with a single accepting state.

12. Construct a Turing machine with two-way tape and input alphabet $\{a\}$ that halts if the tape contains a nonblank square. The symbol a may be anywhere on the tape, not necessarily to the immediate right of the tape head.

13. A **two-dimensional** Turing machine is one in which the tape consists of a two-dimensional array of tape squares.



A transition consists of rewriting a square and moving the head to any one of the four adjacent squares. A computation begins with the tape head reading the corner position. The transitions of the two-dimensional machine are written $\delta(q_i, x) = [q_j, y, d]$, where d is U (up), D (down), L (left), or R (right). Design a two-dimensional Turing machine with input alphabet $\{a\}$ that halts if the tape contains a nonblank square.

14. Let L be the set of palindromes over $\{a, b\}$.

a) Build a standard Turing machine that accepts L.

b) Build a two-tape machine that accepts L in which the computation with input u should take no more than 3 $length(u) + 4$ transitions.

15. Construct a two-tape Turing machine with input alphabet $\{a, b\}$ that accepts the language $\{a^i b^{2i} \mid i \geq 0\}$ in which the tape head on the input tape only moves from left to right.

16. Construct a two-tape Turing machine with input alphabet $\{a, b, c\}$ that accepts the language $\{a^i b^i c^i \mid i \geq 0\}$.

17. Construct a two-tape Turing machine with input alphabet $\{a, b\}$ that accepts strings with the same number of a's and b's. The computation with input u should take no more than 2 $length(u) + 3$ transitions.

18. Construct a two-tape Turing machine that accepts strings in which each a is followed by an increasing number of b's; that is, the strings are of the form

$$ab^{n_1}ab^{n_2}\ldots ab^{n_k},\ k > 0,$$

where $n_1 < n_2 < \cdots < n_k$.

19. Construct a nondeterministic Turing machine whose language is the set of strings over $\{a, b\}$ that contain a substring u satisfying the following two properties:

i) $length(u) \geq 3$;

ii) u contains the same number of a's and b's.

20. Construct a two-tape nondeterministic Turing machine that accepts $L = \{uvuw \mid u \in \{a, b\}^5,\ v, w \in \{a, b\}^*\}$. A string is in L if it contains two nonoverlapping identical

substrings of length 5. Every computation with input w should terminate after at most $2\ length(w) + 2$ transitions.

21. Construct a two-tape nondeterministic Turing machine that accepts $L = \{uu \mid u \in \{a, b\}^*\}$. Every computation with input w should terminate after at most $2\ length(w) + 2$ transitions. Using the deterministic machine from Example 8.6.2 that accepts L, what is the maximum number of transitions required for a computation with an input of length n?

22. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a standard Turing machine that accepts a language L. Design a Turing machine M′ (of any variety) that accepts a string $w \in \Sigma^*$ if, and only if, there is a substring of w in L.

23. Let L be a language accepted by a nondeterministic Turing machine in which every computation terminates. Prove that L is recursive.

24. Prove the equivalent of Theorem 8.3.2 for nondeterministic Turing machines.

25. Prove that every finite language is recursive.

26. Prove that a language L is recursive if, and only if, L and $\overline{L}$ are recursively enumerable.

27. Prove that the recursive languages are closed under union, intersection, and complement.

28. A machine that generates all sequences made up of integers from 1 to n was given in Figure 8.1. Trace the first seven cycles of the machine for $n = 3$. A cycle consists of the tape head returning to the initial position in state q_0.

29. Build a Turing machine that enumerates the set of even length strings over $\{a\}$.

30. Build a Turing machine that enumerates the set $\{a^i b^j \mid 0 \leq i \leq j\}$.

31. Build a Turing machine that enumerates the set $\{a^{2^n} \mid n \geq 0\}$.

32. Build a Turing machine E_{Σ^*} that enumerates Σ^* where $\Sigma = \{0, 1\}$. *Note:* This machine may be thought of as enumerating all finite-length bit strings.

*33. Build a machine that enumerates the ordered pairs N × N. Represent a number n by a string of $n + 1$ *1*'s. The output for ordered pair $[i, j]$ should consist of the representation of the number i followed by a blank followed by the representation of j. The markers # should surround the entire ordered pair.

34. In Theorem 8.8.7, the proof that every recursive language can be enumerated in lexicographical order considered the cases of finite and infinite languages separately. The argument for an infinite language may not be sufficient for a finite language. Why?

35. Define the components of a two-track nondeterministic Turing machine. Prove that these machines accept precisely the recursively enumerable languages.

36. Prove that every context-free language is recursive. *Hint:* Construct a two-tape nondeterministic Turing machine that simulates the computation of a pushdown automaton.

Bibliographic Notes

The Turing machine was introduced by Turing [1936] as a model for algorithmic computation. Turing's original machine was deterministic, consisting of a two-way tape and a single tape head. Independently, Post [1936] introduced a family of abstract machines with the same computational capabilities as Turing machines.

The use of Turing machines for the computation of functions is presented in Chapter 9. The capabilities and limitations of Turing machines as language acceptors are examined in Chapters 10 and 11. The books by Kleene [1952], Minsky [1967], Brainerd and Landweber [1974], and Hennie [1977] give an introduction to computability and Turing machines.

CHAPTER 9

Turing Computable Functions

In the preceding chapter Turing machines provided the computational framework for accepting languages. The result of a computation was determined by final state or by halting. In either case there are only two possible outcomes: accept or reject. The result of a Turing machine computation can also be defined in terms of the symbols written on the tape when the computation terminates. Defining the result in terms of the halting tape configuration permits an infinite number of possible outcomes. In this manner, the computations of a Turing machine produce a mapping between input strings and output strings; that is, the Turing machine computes a function. When the strings are interpreted as natural numbers, Turing machines can be used to compute number-theoretic functions. We will show that several important number-theoretic functions are Turing computable and that computability is closed under the composition of functions. In Chapter 13 we will categorize the entire family of functions that can be computed by Turing machines.

The current chapter ends by outlining how a high-level programming language could be defined using the Turing machine architecture. This brings Turing machine computations closer to the computational paradigm with which we are most familiar—the modern-day computer.

9.1 Computation of Functions

A function $f : X \to Y$ is a mapping that assigns at most one value from the set Y to each element of the domain X. Adopting a computational viewpoint, we refer to the variables of f as the input of the function. The definition of a function does not specify how to obtain

$f(x)$, the value assigned to x by the function f, from the input x. Turing machines will be designed to compute the values of functions. The domain and range of a function computed by a Turing machine consist of strings over the input alphabet of the machine.

A Turing machine that computes a function has two distinguished states: the initial state q_0 and the halting state q_f. A computation begins with a transition from state q_0 that positions the tape head at the beginning of the input string. The state q_0 is never reentered; its sole purpose is to initiate the computation. All computations that terminate do so in state q_f with the value of the function written on the tape beginning at position one. These conditions are formalized in Definition 9.1.1.

Definition 9.1.1

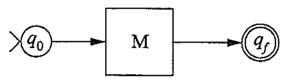
A deterministic one-tape Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$ computes the unary function $f : \Sigma^* \to \Sigma^*$ if

i) there is only one transition from the state q_0 and it has the form $\delta(q_0, B) = [q_i, B, R]$;
ii) there are no transitions of the form $\delta(q_i, x) = [q_0, y, d]$ for any $q_i \in Q$, $x, y \in \Gamma$, and $d \in \{L, R\}$;
iii) there are no transitions of the form $\delta(q_f, B)$;
iv) the computation with input u halts in the configuration $q_f BvB$ whenever $f(u) = v$; and
v) the computation continues indefinitely whenever $f(u)\uparrow$.

A function is said to be **Turing computable** if there is a Turing machine that computes it. A Turing machine that computes a function f may fail to halt for an input string u. In this case, f is undefined for u. Thus Turing machines can compute both total and partial functions.

An arbitrary function need not have the same domain and range. Turing machines can be designed to compute functions from Σ^* to a specific set R by designating an input alphabet Σ and a range R. Condition (iv) is then interpreted as requiring the string v to be an element of R.

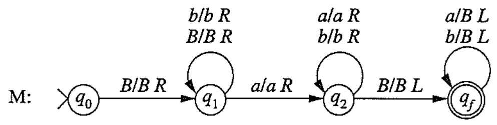
To highlight the distinguished states q_0 and q_f, a Turing machine M that computes a function is depicted by the diagram



Intuitively, the computation remains inside the box labeled M until termination. This diagram is somewhat simplistic since Definition 9.1.1 permits multiple transitions to state q_f and transitions from q_f. However, condition (iii) ensures that there are no transitions from q_f when the machine is scanning a blank. When this occurs, the computation terminates with the result written on the tape.

Example 9.1.1

The Turing machine



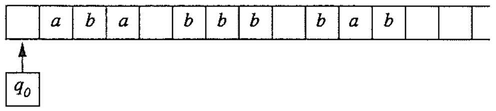
computes the partial function f from $\{a, b\}^*$ to $\{a, b\}^*$ defined by

$$f(u) = \begin{cases} \lambda & \text{if } u \text{ contains an } a \\ \uparrow & \text{otherwise.} \end{cases}$$

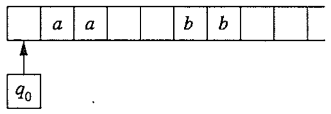
The function f is undefined if the input does not contain an a. In this case the machine moves indefinitely to the right in state q_1. When an a is encountered, the machine enters state q_2 and reads the remainder of the input. The computation is completed by erasing the input while returning to the initial position. A computation that terminates produces the configuration $q_f BB$ designating the null string as the result. □

The machine M in Example 9.1.1 was designed to compute the unary function f. It should be neither surprising nor alarming that computations of M do not satisfy the requirements of Definition 9.1.1 when the input does not have the anticipated form. A computation of M initiated with input $BbBbBaB$ terminates in the configuration $BbBbq_fB$. In this halting configuration, the tape does not contain a single value and the tape head is not in the correct position. This is just another manifestation of the time-honored "garbage in, garbage out" principle of computer science.

Functions with more than one argument are computed in a similar manner. The input is placed on the tape with the arguments separated by blanks. The initial configuration of a computation of a ternary function f with input aba, bbb, and bab is



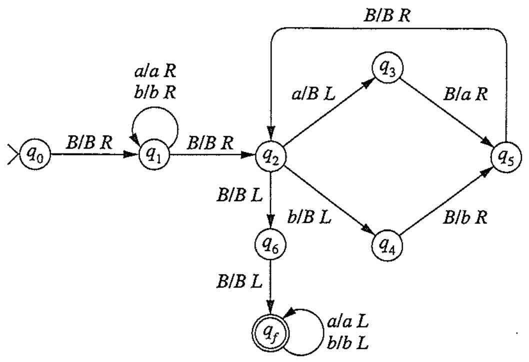
If $f(aba, bbb, bab)$ is defined, the computation terminates with the configuration $q_f Bf(aba, bbb, bab)B$. The initial configuration for the computation of $f(aa, \lambda, bb)$ is



The consecutive blanks in tape positions three and four indicate that the second argument is the null string.

Example 9.1.2

The Turing machine



computes the binary function of concatenation of strings over $\{a, b\}$. The initial configuration of a computation with input strings u and v has the form q_0BuBvB. Either or both of the input strings may be null.

The initial string is read in state q_1. The cycle formed by states q_2, q_3, q_5, q_2 translates an a one position to the left. Similarly, q_2, q_4, q_5, q_2 shift a b to the left. These cycles are repeated until the entire second argument has been translated one position to the left, producing the configuration q_fBuvB. □

Turing machines that compute functions can also be used to accept languages. The **characteristic function** of a language L is a function $\chi_L : \Sigma^* \to \{0, 1\}$ defined by

$$\chi_L(u) = \begin{cases} 1 & \text{if } u \in L \\ 0 & \text{if } u \notin L. \end{cases}$$

A language L is recursive if there is a Turing machine M that computes the characteristic function χ_L. The results of the computations of M indicate the acceptability of strings. A machine that computes the partial characteristic function

$$\hat{\chi}_L(u) = \begin{cases} 1 & \text{if } u \in L \\ 0 \text{ or } \uparrow & \text{if } u \notin L \end{cases}$$

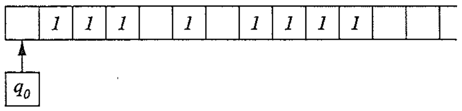
shows that L is recursively enumerable. Exercises 2, 3, and 4 establish the equivalence between acceptance of a language by a Turing machine and the computability of its characteristic function.

9.2 Numeric Computation

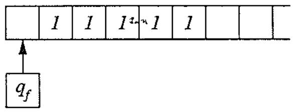
We have seen that Turing machines can be used to compute the values of functions whose domain and range consist of strings over the input alphabet. In this section we turn our attention to numeric computation, in particular the computation of number-theoretic functions. A **number-theoretic function** is a function of the form $f : \mathrm{N} \times \mathrm{N} \times \cdots \times \mathrm{N} \to \mathrm{N}$. The domain consists of natural numbers or n-tuples of natural numbers. The function $sq : \mathrm{N} \to \mathrm{N}$ defined by $sq(n) = n^2$ is a unary number-theoretic function. The standard operations of addition and multiplication are binary number-theoretic functions.

The transition from symbolic to numeric computation requires only a change of perspective since numbers are represented by strings of symbols. The input alphabet of the Turing machine is determined by the representation of the natural numbers used in the computation. We will represent the natural number n by the string 1^{n+1}. The number zero is represented by the string 1, the number one by 11, and so on. This notational scheme is known as the *unary representation* of the natural numbers. The unary representation of a natural number n is denoted $\overline{n}$. When numbers are encoded using the unary representation, the input alphabet for a machine that computes a number-theoretic function is the singleton set $\{1\}$.

The computation of $f(2, 0, 3)$ in a Turing machine that computes a ternary number-theoretic function f begins with the machine configuration



If $f(2, 0, 3) = 4$, the computation terminates with the configuration



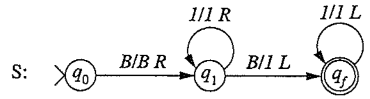
A k-variable total number-theoretic function $r : \mathrm{N} \times \mathrm{N} \times \cdots \times \mathrm{N} \to \{0, 1\}$ defines a k-ary relation R on the domain of the function. The relation is defined by

$$[n_1, n_2, \ldots, n_k] \in \mathrm{R} \text{ if } r(n_1, n_2, \ldots, n_k) = 1$$
$$[n_1, n_2, \ldots, n_k] \notin \mathrm{R} \text{ if } r(n_1, n_2, \ldots, n_k) = 0.$$

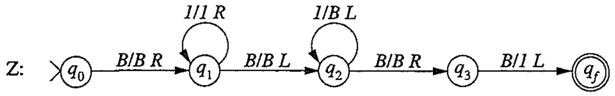
The function r is called the **characteristic function** of the relation R. A relation is Turing computable if its characteristic function is Turing computable.

We will now construct Turing machines that compute several simple, but important, number-theoretic functions. The functions are denoted by lowercase letters and the corresponding machines by capital letters.

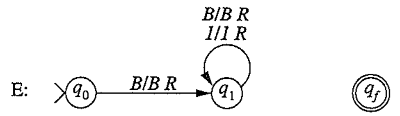
The successor function: $s(n) = n + 1$



The zero function: $z(n) = 0$

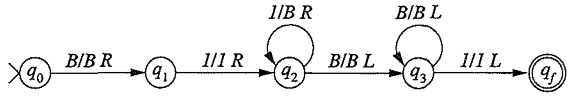


The empty function: $e(n) \uparrow$



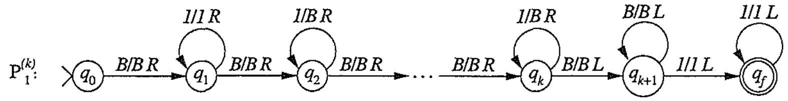
The machine that computes the successor simply adds a 1 to the right end of the input string. The zero function is computed by erasing the input and writing 1 in tape position one. The empty function is undefined for all arguments; the machine moves indefinitely to the right in state q_1.

The zero function is also computed by the machine



That two machines compute the same function illustrates the difference between functions and algorithms. A function is a mapping from elements in the domain to elements in the range. A Turing machine mechanically computes the value of the function whenever the function is defined. The difference is that of definition and computation. In Section 9.5 we will see that there are number-theoretic functions that cannot be computed by any Turing machine.

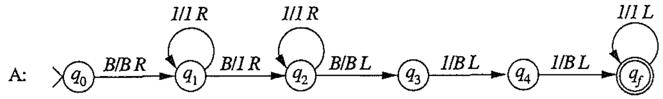
The value of the k-variable projection function $p_i^{(k)}$ is defined as the ith argument of the input, $p_i^{(k)}(n_1, n_2, \ldots, n_i, \ldots, n_k) = n_i$. The superscript k specifies the number of arguments and the subscript designates the argument that defines the result of the projection. The superscript is placed in parentheses so that it is not mistaken for an exponent. The machine that computes $p_1^{(k)}$ leaves the first argument unchanged and erases the remaining arguments.



The function $p_1^{(1)}$ maps a single input to itself. This function is also called the *identity function* and is denoted id. Machines $P_i^{(k)}$ that compute $p_i^{(k)}$ will be designed in Example 9.3.1.

Example 9.2.1

The Turing machine A computes the binary function defined by the addition of natural numbers.



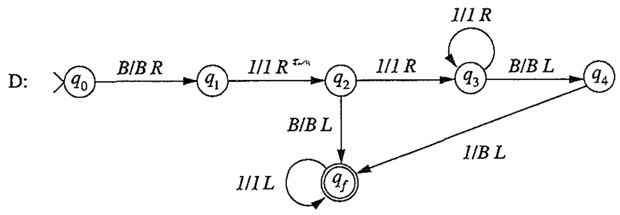
The unary representations of natural numbers n and m are 1^{n+1} and 1^{m+1}. The sum of these numbers is represented by 1^{n+m+1}. This string is generated by replacing the blank between the arguments with a 1 and erasing two 1's from the right end of the second argument. □

Example 9.2.2

The predecessor function

$$pred(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$$

is computed by the machine D (decrement):

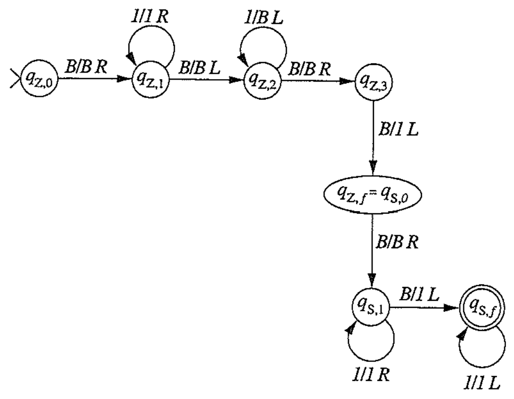


For input greater than zero, the computation erases the rightmost 1 on the tape. □

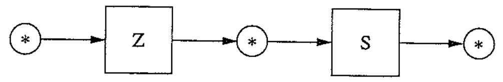
9.3 Sequential Operation of Turing Machines

Turing machines designed to accomplish a single task can be combined to construct machines that perform complex computations. Intuitively, the combination is obtained by running the machines sequentially. The result of one computation becomes the input for the succeeding machine. A machine that computes the constant function $c(n) = 1$ can be constructed by combining the machines that compute the zero and the successor functions. Regardless of the input, a computation of the machine Z terminates with the value zero on the tape. Running the machine S on this tape configuration produces the number one.

The computation of Z terminates with the tape head in position zero scanning a blank. These are precisely the input conditions for the machine S. The initiation and termination conditions of Definition 9.1.1 were introduced to facilitate this coupling of machines. The handoff between machines is accomplished by identifying the final state of Z with the initial state of S. Except for this handoff, the states of the two machines are assumed to be distinct. This can be ensured by subscripting each state of the composite machine with the name of the original machine.



The sequential combination of two machines is represented by the diagram



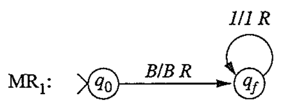
The state names are omitted from the initial and final nodes in the diagram since they may be inferred from the constituent machines.

There are certain sequences of actions that frequently occur in a computation of a Turing machine. Machines can be constructed to perform these recurring tasks. These machines are designed in a manner that allows them to be used as components in more complicated machines. Borrowing terminology from assembly language programming, we call a machine constructed to perform a single simple task a **macro**.

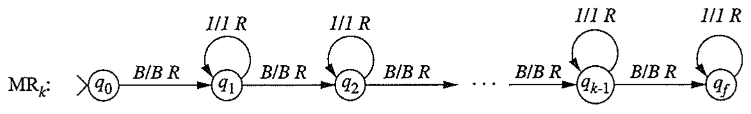
The computations of a macro adhere to several of the restrictions introduced in Definition 9.1.1. The initial state q_0 is used strictly to initiate the computation. Since these machines are combined to construct more complex machines, we do not assume that a computation must begin with the tape head at position zero. We do assume, however, that each computation begins with the machine scanning a blank. Depending upon the operation, the

segment of the tape to the immediate right or left of the tape head will be examined by the computation. A macro may contain several states in which a computation may terminate. As with machines that compute functions, a macro is not permitted to contain a transition of the form $\delta(q_f, B)$ from any halting state q_f.

A family of macros is often described by a schema. The macro MR_i moves the tape head to the right through i consecutive natural members (sequences of I's) on the tape. MR_1 is defined by the machine



MR_k is constructed by adding states to move the tape head through the sequence of k natural numbers.



The move macros do not affect the tape to the left of the initial position of the tape head. A computation of MR_2 that begins with the configuration $B\overline{n}_1 q_0 B\overline{n}_2 B\overline{n}_3 B\overline{n}_4 B$ terminates in the configuration $B\overline{n}_1 B\overline{n}_2 B\overline{n}_3 q_f B\overline{n}_4 B$.

Macros, like Turing machines that compute functions, expect to be run with the input having a specified form. The move right macro MR_i requires a sequence of at least i natural numbers to the immediate right of the tape at the initiation of a computation. The design of a composite machine must ensure that the appropriate input configuration is provided to each macro.

Several families of macros are defined by describing the results of a computation of the machine. The computation of each macro remains within the segment of the tape indicated by the initial and final blank in the description. The application of the macro will neither access nor alter any portion of tape outside of these bounds. The location of the tape head is indicated by the underscore. The double arrows indicate identical tape positions in the before and after configurations.

ML_k (move left):

$$\begin{array}{ll} B\overline{n}_1 B\overline{n}_2 B \ldots B\overline{n}_k \underline{B} & \quad k \geq 0 \\ \updownarrow \qquad\qquad\qquad \updownarrow & \\ \underline{B}\overline{n}_1 B\overline{n}_2 B \ldots B\overline{n}_k B & \end{array}$$

FR (find right):

$$\begin{array}{ll} \underline{B}B^i\bar{n}B & i \geq 0 \\ \updownarrow \quad \updownarrow & \\ B^i\underline{B}\bar{n}B & \end{array}$$

FL (find left):

$$\begin{array}{ll} B\bar{n}B^i\underline{B} & i \geq 0 \\ \updownarrow \quad \updownarrow & \\ \underline{B}\bar{n}B^iB & \end{array}$$

E_k (erase):

$$\begin{array}{ll} \underline{B}\bar{n}_1B\bar{n}_2B \ldots B\bar{n}_kB & k \geq 1 \\ \updownarrow \qquad\qquad \updownarrow & \\ \underline{B}B \quad \ldots \quad BB & \end{array}$$

CPY_k (copy):

$$\begin{array}{ll} \underline{B}\bar{n}_1B\bar{n}_2B \ldots B\bar{n}_kBBB \quad \ldots \quad BB & k \geq 1 \\ \updownarrow \qquad\qquad \updownarrow \qquad\qquad \updownarrow & \\ \underline{B}\bar{n}_1B\bar{n}_2B \ldots B\bar{n}_kB\bar{n}_1B\bar{n}_2B \ldots B\bar{n}_kB & \end{array}$$

$\mathrm{CPY}_{k,i}$ (copy through i numbers):

$$\begin{array}{ll} \underline{B}\bar{n}_1B\bar{n}_2B \ldots B\bar{n}_kB\bar{n}_{k+1} \ldots B\bar{n}_{k+i}BB \quad \ldots \quad BB & k \geq 1 \\ \updownarrow \qquad\qquad \updownarrow \qquad\qquad \updownarrow \qquad\qquad \updownarrow & \\ \underline{B}\bar{n}_1B\bar{n}_2B \ldots B\bar{n}_kB\bar{n}_{k+1} \ldots B\bar{n}_{k+i}B\bar{n}_1B\bar{n}_2B \ldots B\bar{n}_kB & \end{array}$$

T (translate):

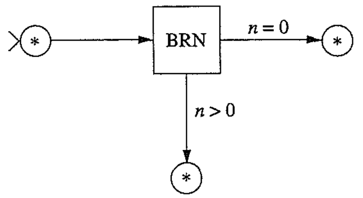
$$\begin{array}{ll} \underline{B}B^i\bar{n}B & i \geq 0 \\ \updownarrow \quad \updownarrow & \\ \underline{B}\bar{n}B^iB & \end{array}$$

The find macros move the tape head into a position to process the first natural number to the right or left of the current position. E_k erases a sequence of k natural numbers and halts with the tape head in its original position.

The copy machines produce a copy of the designated number of integers. The segment of the tape on which the copy is produced is assumed to be blank. $\mathrm{CPY}_{k,i}$ expects a sequence

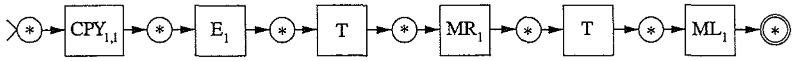
of $k+i$ numbers followed by a blank segment large enough to hold a copy of the first k numbers. The translate macro changes the location of the first natural number to the right of the tape head. A computation terminates with the head in the position it occupied at the beginning of the computation with the translated string to its immediate right.

The BRN (branch on zero) macro has two possible terminating states. The input to the macro BRN, a single natural number, is used to select the halting state of the macro. The branch macro is depicted



The computation of BRN does not alter the tape nor change the position of the tape head. Consequently, it may be run in any configuration $\underline{B}\bar{n}B$. The branch macro is often used in the construction of loops in composite machines and in the selection of alternative computations.

Additional macros can be created using those already defined. The machine



interchanges the order of two numbers. The tape configurations for this macro are

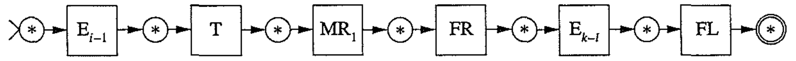
INT (interchange):

$$\begin{array}{l} \underline{B}\bar{n}B\bar{m}BB^{n+1}B \\ \updownarrow \qquad\quad \updownarrow \\ \underline{B}\bar{m}B\bar{n}BB^{n+1}B \end{array}$$

In Exercise 6, you are asked to construct a Turing machine for the macro INT that does not leave the tape segment $B\bar{n}B\bar{m}B$.

Example 9.3.1

The computation of a machine that evaluates the projection function $p_i^{(k)}$ consists of three distinct actions: erasing the initial $i-1$ arguments, translating the ith argument to tape position one, and erasing the remainder of the input. A machine to compute $p_i^{(k)}$ can be designed using the macros FR, FL, E_i, MR_1, and T.

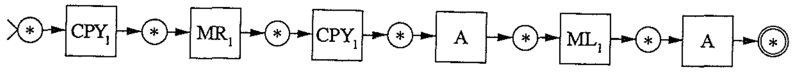


□

Turing machines defined to compute functions can be used like macros in the design of composite machines. Unlike the computations of the macros, there is no a priori bound on the amount of tape required by a computation of such a machine. Consequently, these machines should be run only when the input is followed by a completely blank tape.

Example 9.3.2

The macros and previously constructed machines can be used to design a Turing machine that computes the function $f(n) = 3n$.



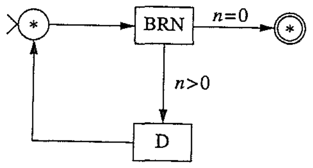
The machine A, constructed in Example 9.2.1, adds two natural numbers. The computation of $f(n)$ combines the copy macro with A to add three copies of n. A computation with input $\overline{n}$ generates the following sequence of tape configurations.

Machine	Configuration
	$\underline{B}\overline{n}B$
CPY_1	$\underline{B}\overline{n}B\overline{n}B$
MR_1	$B\overline{n}\underline{B}\overline{n}B$
CPY_1	$B\overline{n}\underline{B}\overline{n}B\overline{n}B$
A	$B\overline{n}\underline{B}\overline{n+n}B$
ML_1	$\underline{B}\overline{n}B\overline{n+n}B$
A	$\underline{B}\overline{n+n+n}B$

Note that the addition machine A is run only when its arguments are the two rightmost encoded numbers on the tape. □

Example 9.3.3

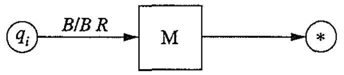
The one-variable constant function zero defined by $z(n) = 0$, for all $n \in \mathbf{N}$, can be built from the BRN macro and the machine D that computes the predecessor function.



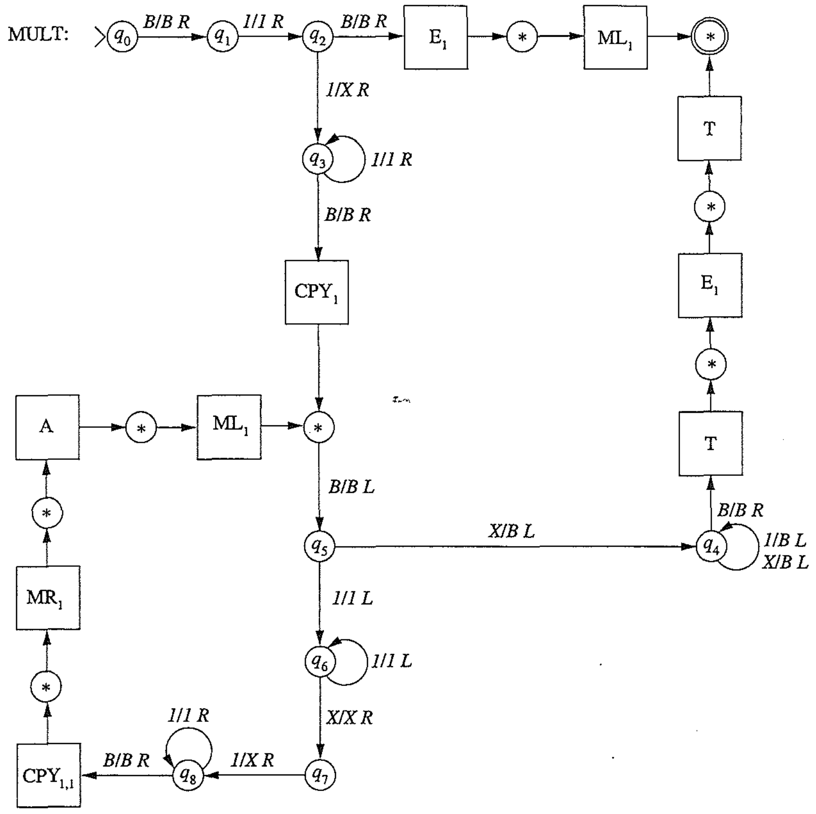
□

Example 9.3.4

A Turing machine MULT is constructed to compute the multiplication of natural numbers. Macros can be mixed with standard Turing machine transitions when designing a composite machine. The conditions on the initial state of a macro permit the submachine to be entered upon the processing of a blank from any state. The identification of the start state of a macro with a state q_i is depicted



Since the macro is entered only upon the processing of a blank, transitions may also be defined for state q_i with the tape head scanning nonblank tape symbols.



If the first argument is zero, the computation erases the second argument, returns to the initial position, and halts. Otherwise, a computation of MULT adds m to itself n times. The addition is performed by copying $\overline{m}$ and then adding the copy to the previous total. The number of iterations is recorded by replacing a *1* in the first argument with an X when a copy is made. □

9.4 Composition of Functions

Using the interpretation of a function as a mapping from its domain to its range, we can represent the unary number-theoretic functions g and h by the diagrams

$$\begin{tikzcd} \text{N} \arrow[r, bend left, "g"] & \text{N} \end{tikzcd} \qquad \begin{tikzcd} \text{N} \arrow[r, bend left, "h"] & \text{N} \end{tikzcd}$$

A mapping from N to N can be obtained by identifying the range of g with the domain of h and sequentially traversing the arrows in the diagrams.

$$\begin{tikzcd} \text{N} \arrow[r, bend left, "g"] & \text{N} \arrow[r, bend left, "h"] & \text{N} \end{tikzcd}$$

The function obtained by this combination is called the composition of h with g. The composition of unary functions is formally defined in Definition 9.4.1. Definition 9.4.2 extends the notion to n-variable functions.

Definition 9.4.1

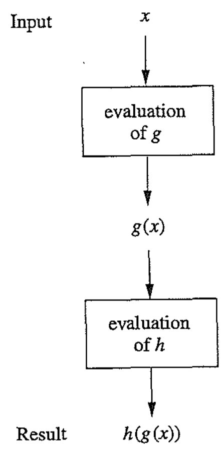
Let g and h be unary number-theoretic functions. The **composition** of h with g is the unary function $f : \text{N} \to \text{N}$ defined by

$$f(x) = \begin{cases} \uparrow & \text{if } g(x) \uparrow \\ \uparrow & \text{if } g(x) = y \text{ and } h(y) \uparrow \\ h(y) & \text{if } g(x) = y \text{ and } h(y) \downarrow. \end{cases}$$

The composite function is denoted $f = h \circ g$.

The value of the composite function $f = h \circ g$ for input x is written $f(x) = h(g(x))$. The latter expression is read "h of g of x." The value $h(g(x))$ is defined whenever $g(x)$ is defined and h is defined for the value $g(x)$. Consequently, the composition of total functions produces a total function.

From a computational viewpoint, the composition $h \circ g$ consists of the sequential evaluation of functions g and h. The computation of g provides the input for the computation of h:



The composite function is defined only when the preceding sequence of computations can be successfully completed.

Definition 9.4.2

Let $g_1, g_2, \ldots, g_n$ be k-variable number-theoretic functions and let h be an n-variable number-theoretic function. The k-variable function f defined by

$$f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$$

is called the **composition** of h with $g_1, g_2, \ldots, g_n$ and written $f = h \circ (g_1, \ldots, g_n)$. The function $f(x_1, \ldots, x_k)$ is undefined if either

i) $g_i(x_1, \ldots, x_k) \uparrow$ for some $1 \le i \le n$, or
ii) $g_i(x_1, \ldots, x_k) = y_i$ for $1 \le i \le n$ and $h(y_1, \ldots, y_n) \uparrow$.

The general definition of composition of functions also admits a computational interpretation. The input is provided to each of the functions g_i. These functions generate the arguments of h.

Example 9.4.1

Consider the mapping defined by the composite function

$$add \circ (c_2^{(3)}, add \circ (p_1^{(3)}, p_3^{(3)})),$$

where $add(n, m) = n + m$ and $c_2^{(3)}$ is the three-variable constant function defined by

$c_2^{(3)}(n_1, n_2, n_3) = 2$. The composite is a three-variable function since the innermost functions of the composition, the functions that directly utilize the input, require three arguments. The function adds the sum of the first and third arguments to the constant 2. The result for input 1, 0, 3 is

$$\begin{aligned}
&add \circ (c_2^{(3)}, add \circ (p_1^{(3)}, p_3^{(3)}))(1, 0, 3)\\
&= add \circ (c_2^{(3)}(1, 0, 3), add \circ (p_1^{(3)}, p_3^{(3)})(1, 0, 3))\\
&= add(2,\ add(p_1^{(3)}(1, 0, 3), p_3^{(3)}(1, 0, 3)))\\
&= add(2,\ add(1, 3))\\
&= add(2, 4)\\
&= 6.
\end{aligned}$$

□

A function obtained by composing Turing computable functions is itself Turing computable. The argument is constructive; a machine can be designed to compute the composite function by combining the machines that compute the constituent functions and the macros developed in the previous section.

Let g_1 and g_2 be three-variable Turing computable functions and let h be a Turing computable two-variable function. Since g_1, g_2, and h are computable, there are machines G_1, G_2, and H that compute them. The actions of a machine that computes the composite function $h \circ (g_1, g_2)$ are traced for input n_1, n_2, and n_3.

Machine	Configuration
	$\underline{B}\overline{n}_1B\overline{n}_2B\overline{n}_3B$
CPY_3	$\underline{B}\overline{n}_1B\overline{n}_2B\overline{n}_3B\overline{n}_1B\overline{n}_2B\overline{n}_3B$
MR_3	$B\overline{n}_1B\overline{n}_2B\overline{n}_3\underline{B}\overline{n}_1B\overline{n}_2B\overline{n}_3B$
G_1	$B\overline{n}_1B\overline{n}_2B\overline{n}_3\underline{B}\overline{g_1(n_1, n_2, n_3)}B$
ML_3	$\underline{B}\overline{n}_1B\overline{n}_2B\overline{n}_3B\overline{g_1(n_1, n_2, n_3)}B$
$\mathrm{CPY}_{3,1}$	$\underline{B}\overline{n}_1B\overline{n}_2B\overline{n}_3B\overline{g_1(n_1, n_2, n_3)}B\overline{n}_1B\overline{n}_2B\overline{n}_3B$
MR_4	$B\overline{n}_1B\overline{n}_2B\overline{n}_3B\overline{g_1(n_1, n_2, n_3)}\underline{B}\overline{n}_1B\overline{n}_2B\overline{n}_3B$
G_2	$B\overline{n}_1B\overline{n}_2B\overline{n}_3B\overline{g_1(n_1, n_2, n_3)}\underline{B}\overline{g_2(n_1, n_2, n_3)}B$
ML_1	$B\overline{n}_1B\overline{n}_2B\overline{n}_3\underline{B}\overline{g_1(n_1, n_2, n_3)}B\overline{g_2(n_1, n_2, n_3)}B$
H	$B\overline{n}_1B\overline{n}_2B\overline{n}_3\underline{B}\overline{h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))}B$
ML_3	$\underline{B}\overline{n}_1B\overline{n}_2B\overline{n}_3B\overline{h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))}B$
E_3	$\underline{B}B \quad \ldots \quad B\overline{h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))}B$
T	$\underline{B}\overline{h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))}B$

The computation copies the input and computes the value of g_1 using the newly created copy as the arguments. Since the machine G_1 does not move to the left of its starting position, the original input remains unchanged. If $g_1(n_1, n_2, n_3)$ is undefined, the computation of G_1 continues indefinitely. In this case the entire computation fails to terminate, correctly indicating that $h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))$ is undefined. Upon the termination of G_1, the input is copied and G_2 is run on the new copy.

If both $g_1(n_1, n_2, n_3)$ and $g_2(n_1, n_2, n_3)$ are defined, G_2 terminates with the input for H on the tape preceded by the original input. The machine H is run computing $h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))$. When the computation of H terminates, the result is translated to the correct position.

The preceding construction easily generalizes to the composition of functions of any number of variables, yielding Theorem 9.4.3.

Theorem 9.4.3

The Turing computable functions are closed under the operation of composition.

Theorem 9.4.3 can be used to show that a function f is Turing computable without explicitly constructing a machine that computes it. If f can be defined as the composition of Turing computable functions then, by Theorem 9.4.3, f is also Turing computable.

Example 9.4.2

The k-variable constant functions $c_i^{(k)}$ whose values are given by $c_i^{(k)}(n_1, \ldots, n_k) = i$ are Turing computable. The function $c_i^{(k)}$ can be defined by

$$c_i^{(k)} = \underbrace{s \circ s \circ \cdots \circ s}_{i \text{ times}} \circ z \circ p_1^{(k)}.$$

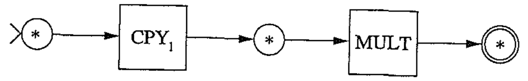
The projection function accepts the k-variable input and passes the first value to the zero function. The composition of i successor functions produces the desired value. Since each of the functions in the composition is Turing computable, the function $c_i^{(k)}$ is Turing computable by Theorem 9.4.3. □

Example 9.4.3

The binary function $smsq(n, m) = n^2 + m^2$ is Turing computable. The sum-of-squares function can be written as the composition of functions

$$smsq = add \circ (sq \circ p_1^{(2)},\ sq \circ p_2^{(2)}),$$

where sq is defined by $sq(n) = n^2$. The function add is computed by the machine constructed in Example 9.2.1 and sq by

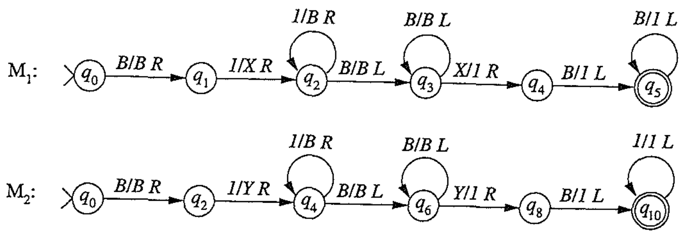


□

9.5 Uncomputable Functions

A function is Turing computable only if there is a Turing machine that computes it. The existence of number-theoretic functions that are not Turing computable can be demonstrated by a simple counting argument. We begin by showing that the set of computable functions is countably infinite.

A Turing machine is completely defined by its transition function. The states and tape alphabet used in computations of the machine can be extracted from the transitions. Consider the machines M_1 and M_2 defined by

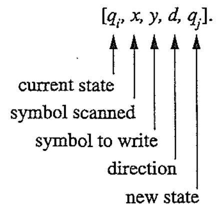


Both M_1 and M_2 compute the unary constant function $c_1^{(1)}$. The two machines differ only in the names given to the states and the markers used during the computation. These symbols have no effect on the result of a computation and hence the function computed by the machine.

Since the names of the states and tape symbols other than B and 1 are immaterial, we adopt the following conventions concerning the naming of the components of a Turing machine:

i) The set of states is a finite subset of $Q_0 = \{q_i \mid i \geq 0\}$.
ii) The input alphabet is $\{1\}$.
iii) The tape alphabet is a finite subset of the set $\Gamma_0 = \{B, 1, X_i \mid i \geq 0\}$.
iv) The initial state is q_0.

The transitions of a Turing machine have been specified using functional notation; the transition defined for state q_i and tape symbol x is represented by $\delta(q_i, x) = [q_j, y, d]$. This information can also be represented by the quintuple



With the preceding naming conventions, a transition of a Turing machine is an element of the set $T = Q_0 \times \Gamma_0 \times \Gamma_0 \times \{L, R\} \times Q_0$. The set T is countable since it is the Cartesian product of countable sets.

The transitions of a deterministic Turing machine form a finite subset of T in which the first two components of every element are distinct. There are only a countable number of such subsets. It follows that the number of Turing computable functions is at most countably infinite. On the other hand, the number of Turing computable functions is at least countably infinite since there are countably many constant functions, all of which are Turing computable by Example 9.4.2. These observations yield

Theorem 9.5.1

The set of Turing computable number-theoretic functions is countably infinite.

In Section 1.4, the diagonalization technique was used to prove that there are uncountably many total unary number-theoretic functions. Combining this with Theorem 9.5.1, we obtain Corollary 9.5.2.

Corollary 9.5.2

There is a total unary number-theoretic function that is not Turing computable.

Corollary 9.5.2 vastly understates the relationship between computable and uncomputable functions. The former constitute a countable set and the latter an uncountable set.

9.6 Toward a Programming Language

High-level programming languages are the most commonly employed type of computational system. A program defines a mechanistic and deterministic process, the hallmark of algorithmic computation. The intuitive argument that the computation of a program written in a programming language and executed on a computer can be simulated by a Turing machine rests in the fact that a machine (computer) instruction simply changes the bits in some location of memory. This is precisely the type of action performed by a Turing machine, writing 0's and 1's in memory. Although it may take a large number of Turing machine transitions to accomplish the task, it is not at all difficult to envision a sequence of transitions that will access the correct position and rewrite the memory.

Exercises

1. Construct Turing machines with input alphabet $\{a, b\}$ that compute the specified functions. The symbols u and v represent arbitrary strings over $\{a, b\}^*$.
   a) $f(u) = aaa$
   b) $f(u) = \begin{cases} a & \text{if } length(u) \text{ is even} \\ b & \text{otherwise} \end{cases}$
   c) $f(u) = u^R$
   d) $f(u, v) = \begin{cases} u & \text{if } length(u) > length(v) \\ v & \text{otherwise} \end{cases}$
2. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$ be a Turing machine that computes the partial characteristic function of the language L. Use M to build a standard Turing machine that accepts L.
3. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a standard Turing machine that accepts a language L. Construct a machine M′ that computes the partial characteristic function of L. Recall that the tape of M′ must have the form $q_f B0B$ or $q_f B1B$ upon the completion of a computation of $\hat{\chi}_L$.
4. Let L be a language over Σ and let

$$\chi_L(w) = \begin{cases} 1 & \text{if } w \in L \\ 0 & \text{otherwise} \end{cases}$$

   be the characteristic function of L.
   a) If χ_L is Turing computable, prove that L is recursive.
   b) If L is recursive, prove that there is a Turing machine that computes χ_L.
5. Construct Turing machines that compute the following number-theoretic functions and relations. Do not use macros in the design of these machines.
   a) $f(n) = 2n + 3$
   b) $half(n) = \lfloor n/2 \rfloor$ where $\lfloor x \rfloor$ is the greatest integer less than or equal to x
   c) $f(n_1, n_2, n_3) = n_1 + n_2 + n_3$
   d) $even(n) = \begin{cases} 1 & \text{if } n \text{ is even} \\ 0 & \text{otherwise} \end{cases}$
   e) $eq(n, m) = \begin{cases} 1 & \text{if } n = m \\ 0 & \text{otherwise} \end{cases}$
   f) $lt(n, m) = \begin{cases} 1 & \text{if } n < m \\ 0 & \text{otherwise} \end{cases}$

   g) $n \dot{-} m = \begin{cases} n - m & \text{if } n \geq m \\ 0 & \text{otherwise} \end{cases}$
6. Construct Turing machines that perform the actions specified by the following macros. The computation should not leave the segment of the tape specified in the input configuration.
   a) ZR; input $\underline{B}BB$, output $\underline{B}\overline{0}B$
   b) FL; input $B\overline{n}B^i\underline{B}$, output $\underline{B}\overline{n}B^iB$
   c) E_2; input $\underline{B}\overline{n}B\overline{m}B$, output $\underline{B}B^{n+m+3}B$
   d) T; input $\underline{B}B^i\overline{n}B$, output $\underline{B}\overline{n}B^iB$
   e) BRN; input $\underline{B}\overline{n}B$, output $\underline{B}\overline{n}B$
   f) INT; input $\underline{B}\overline{n}B\overline{m}B$, output $\underline{B}\overline{m}B\overline{n}B$
7. Use the macros and machines constructed in Sections 9.2 through 9.4 to design machines that compute the following functions:
   a) $f(n) = 2n + 3$
   b) $f(n) = n^2 + 2n + 2$
   c) $f(n_1, n_2, n_3) = n_1 + n_2 + n_3$
   d) $f(n, m) = m^3$
   e) $f(n_1, n_2, n_3) = n_2 + 2n_3$
8. Design machines that compute the following relations. You may use the macros and machines constructed in Sections 9.2 through 9.4 and the machines constructed in Exercise 5.
   a) $gt(n, m) = \begin{cases} 1 & \text{if } n > m \\ 0 & \text{otherwise} \end{cases}$
   b) $persq(n) = \begin{cases} 1 & \text{if } n \text{ is a perfect square} \\ 0 & \text{otherwise} \end{cases}$
   c) $divides(n, m) = \begin{cases} 1 & \text{if } n > 0,\ m > 0, \text{ and } m \text{ divides } n \\ 0 & \text{otherwise} \end{cases}$
9. Trace the actions of the machine MULT for computations with input
   a) $n = 0, m = 4$
   b) $n = 1, m = 0$
   c) $n = 2, m = 2$.
10. Describe the mapping defined by each of the following composite functions:
   a) $add \circ (mult \circ (id, id), add \circ (id, id))$
   b) $p_1^{(2)} \circ (s \circ p_1^{(2)}, e \circ p_2^{(2)})$
   c) $mult \circ (c_2^{(3)}, add \circ (p_1^{(3)}, s \circ p_2^{(3)}))$
   d) $mult \circ (mult \circ (p_1^{(1)}, p_1^{(1)}), p_1^{(1)})$.

11. Give examples of total unary number-theoretic functions that satisfy the following conditions:
    a) g is not id and h is not id but $g \circ h = id$.
    b) g is not a constant function and h is not a constant function but $g \circ h$ is a constant function.
12. Give examples of unary number-theoretic functions that satisfy the following conditions:
    a) g is not one-to-one, h is not total, $h \circ g$ is total.
    b) $g \neq e$, $h \neq e$, $h \circ g = e$, where e is the empty function.
    c) $g \neq id$, $h \neq id$, $h \circ g = id$, where id is the identity function.
    d) g is total, h is not one-to-one, $h \circ g = id$.

* 13. Let F be a Turing machine that computes a total unary number-theoretic function f. Design a machine that returns the first natural number n such that $f(n) = 0$. A computation should continue indefinitely if no such n exists. What will happen if the function computed by F is not total?

14. Let F be a Turing machine that computes a total unary number-theoretic function f. Design a machine that computes the function

$$g(n) = \sum_{i=0}^{n} f(i).$$

15. Let F and G be Turing machines that compute total unary number-theoretic functions f and g, respectively. Design a Turing machine that computes the function

$$h(n) = \sum_{i=0}^{n} eq(f(i), g(i)).$$

That is, $h(n)$ is the number of values in the range 0 to n for which the functions f and g assume the same value.

16. A unary relation R over N is Turing computable if its characteristic function is computable. Prove that every computable unary relation over N defines a recursive language. *Hint:* Construct a machine that accepts R from the machine that computes its characteristic function.

* 17. Let $R \subseteq \{I\}^+$ be a recursive language. Prove that R defines a computable unary relation over N.

18. Prove that there are unary relations over N that are not Turing computable.
19. Let F be the set consisting of all total unary number-theoretic functions that satisfy $f(i) = i$ for every even natural number i. Prove that there are functions in F that are not Turing computable.

20. Let v_1, v_2, v_3, v_4 be a listing of the variables used in a TM program and assume register 1 contains a value. Trace the action of the instruction STOR v_2,1. To trace the actions, use the technique in Example 9.3.2.
21. Give a TM program that computes the function $f(v_1, v_2) = v_1 \dot{-} v_2$.

Bibliographic Notes

The Turing machine assembly language provides an architecture that resembles another family of abstract computing devices known as random access machines [Cook and Reckhow, 1973]. Random access machines consist of an infinite number of memory locations and a finite number of registers, each of which is capable of storing a single integer. The instructions of a random access machine manipulate the registers and memory and perform arithmetic operations. These machines provide an abstraction of the standard von Neumann computer architecture. An introduction to random access machines and their equivalence to Turing machines can be found in Aho, Hopcroft, and Ullman [1974].

CHAPTER 11

Decision Problems and the Church-Turing Thesis

In the preceding chapters Turing machines were used to detect patterns in strings, to recognize languages, and to compute functions. Many interesting problems, however, are posed at a higher level than string recognition or manipulation. For example, we may be interested in determining answers to questions of the form: "Is a natural number a perfect square?" Or "Does a graph have a cycle?" Or "Does the computation of a Turing machine halt before the 20th transition?" Each of these general questions describes a decision problem.

Formally, a **decision problem P** is a set of related questions each of which has a yes or no answer. The decision problem of determining if a natural number is a perfect square consists of the following questions:

$$\begin{array}{ll} p_0: & \text{Is 0 a perfect square?} \\ p_1: & \text{Is 1 a perfect square?} \\ p_2: & \text{Is 2 a perfect square?} \\ \vdots & \qquad \vdots \end{array}$$

Each individual question is referred to as an instance of the problem. A solution to a decision problem **P** is an algorithm that determines the appropriate answer to every question $p \in \mathbf{P}$. A decision problem is said to be **decidable** if it has a solution.

Since the solution to a decision problem is an algorithm, a review of our intuitive notion of algorithmic computation may be beneficial. We have not defined, and probably cannot precisely define, the term *algorithm*. This notion falls into the category of "I can't describe it but I know one when I see one." We can, however, list several properties that seem fundamental to the concept of algorithm. An algorithm that solves a decision problem should be

- Complete: It produces the correct answer for each problem instance.
- Mechanistic: It consists of a finite sequence of instructions, each of which can be carried out without requiring insight, ingenuity, or guesswork.
- Deterministic: When presented with identical input, it always performs the same computation.

A procedure that satisfies the preceding properties is often called *effective*.

The computations of a standard Turing machine are clearly mechanistic and deterministic. A Turing machine solution that halts for every input string is also complete. Because of the intuitive effectiveness of their computations, we will use Turing machines as the framework for solving decision problems. The transformation of problem instances into input strings for a Turing machine constitutes the representation of the decision problem. A problem instance is answered affirmatively if the corresponding input string is accepted by the Turing machine and negatively if it is rejected.

The Church-Turing Thesis for decision problems asserts that a Turing machine can be designed to solve any decision problem that is solvable by any effective procedure. A more general interpretation of the Church-Turing Thesis is that any procedure or process that can be algorithmically computed can be realized by a suitably designed Turing machine. This chapter begins by establishing the relationship between decision problems, Turing machines, and recursive languages. The remainder of the chapter presents the Church-Turing Thesis and discusses the importance and implications of the assertion.

11.1 Representation of Decision Problems

The first step in a Turing machine solution of a decision problem is to express the problem in terms of the acceptance of strings. This requires constructing a representation of the problem. Recall the newspaper vending machine described at the beginning of Chapter 5. Thirty cents in nickels, dimes, and quarters is required to open the latch. If more than 30 cents is inserted, the machine keeps the entire amount. Now consider the problem of a miser who wants to buy a newspaper but refuses to pay more than the minimum. A solution to this problem is an effective procedure that determines whether a set of coins contains a combination that totals exactly 30 cents.

A Turing machine representation of the miser's problem transforms an instance of the problem from its natural domain of coins into an equivalent problem of accepting a string. This can be accomplished by representing a set of coins as an element of $\{n, d, q\}^*$ where n, d, and q designate a nickel, a dime, and a quarter, respectively. Using this representation, a Turing machine that solves the miser's problem accepts strings $qnnn$, $nddnd$ and rejects $nnnd$ and $qdqdqqq$. In Exercise 1 you are asked to build a Turing machine that solves this problem.

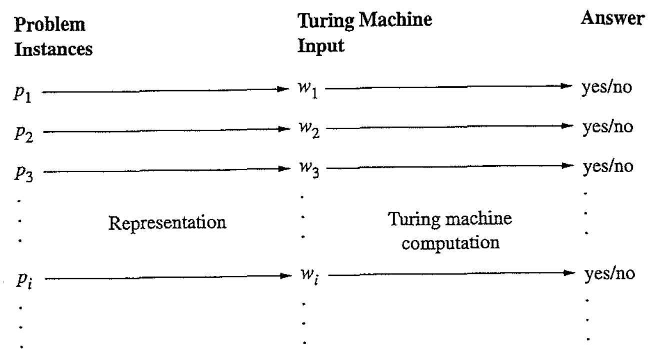
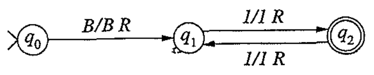


FIGURE 11.1 Solution to decision problem.

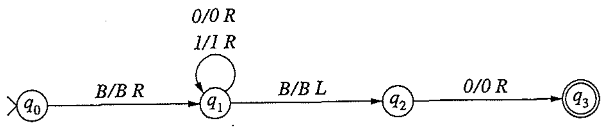
Constructing a Turing machine solution to a decision problem follows the two-step process outlined in Figure 11.1. The first step is the selection of an alphabet and a string representation of the problem instances. The properties of the representation are then utilized in the design of the Turing machine that solves the problem. We illustrate the impact of the representation by considering the problem of determining whether a natural number is even. Two common representations of natural numbers are the unary and binary representations. The alphabet of the unary representation is $\{1\}$ and the number n is represented by the string 1^{n+1}. The alphabet $\{0, 1\}$ is used by the standard binary representation of natural numbers.

The Turing machine



solves the even number problem for the unary representation. The states q_1 and q_2 record whether an even or odd number of 1's have been processed. In the unary representation, a string of odd length represents an even number. Thus the language of M_1 is $\{1^i \mid i \text{ is odd}\}$.

The binary representation of an even number has 0 in the rightmost position. The Turing machine



accepts precisely these strings. The strategies employed by M_1 and M_2 illustrate the dependence of the Turing machine on the choice of the representation.

There are many different ways to represent the instances of a decision problem as strings. A decision problem has a Turing machine solution if there is at least one combination of representation and Turing machine that solves the problem. There may, of course, be many.

11.2 Decision Problems and Recursive Languages

We have chosen the standard Turing machine as a formal system for solving decision problems. Once a string representation of the problem instances is selected, the remainder of the solution consists of the analysis of the input by a Turing machine. Since the completeness property requires the computation of the Turing machine to terminate for every input string, the language accepted by the machine is recursive. Thus every Turing machine solution of a decision problem defines a recursive language. Conversely, every recursive language L can be considered to be the solution of a decision problem. The decision problem, called the membership problem for L, consists of the questions "Is the string w in L?" for every string w over the alphabet of L.

The duality between solvable decision problems and recursive languages can be exploited to broaden the techniques available for establishing the decidability of a decision problem. Since computations of deterministic multitrack and multitape machines can be simulated by a standard Turing machine, solutions using these machines also establish the decidability of a problem.

Example 11.2.1

The decision problem of determining whether a natural number is a perfect square is decidable. The three-tape Turing machine from Example 8.6.2 solves the perfect square problem with the natural number n represented by the string a^n. □

Determinism is one of the fundamental properties of algorithms. However, it is often easier to design a nondeterministic Turing machine than a deterministic one to accept a language. In Section 8.7 it was shown that every language accepted by a nondeterministic Turing machine is also accepted by a deterministic one. A solution to a decision problem requires more than a machine that accepts the appropriate strings; it also demands that all computations terminate. A nondeterministic machine in which every computation terminates can be used to establish the existence of a decision procedure. The languages of such machines are recursive (Exercise 8.23), ensuring the existence of a complete deterministic solution.

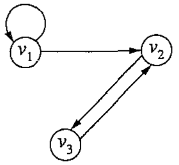
Example 11.2.2

We will use nondeterminism to show that the problem of determining whether there is a path from a node v_i to a node v_j in a directed graph is decidable. A directed graph consists

of a set of nodes $N = \{v_1, \ldots, v_n\}$ and arcs $A \subseteq N \times N$. To represent a graph as a string over $\{0, 1\}$, node v_k is encoded as 1^{k+1} using the unary representation of the subscript of the node. An arc $[v_s, v_t]$ is represented by the string $en(v_s)0en(v_t)$, where $en(v_s)$ and $en(v_t)$ are the encodings of nodes v_s and v_t. The string *00* is used to separate arcs.

The input to the machine consists of a representation of the graph followed by the encoding of nodes v_i and v_j. Three *0*'s separate $en(v_i)$ and $en(v_j)$ from the representation of the graph. The directed graph

$$N = \{v_1, v_2, v_3\}$$
$$A = \{[v_1, v_2], [v_1, v_1], [v_2, v_3], [v_3, v_2]\}$$



is represented by the string *110111001101100111011110011110111*. A computation to determine whether there is a path from v_3 to v_1 in this graph begins with the input *11011100110110011101111001111011100011110 11*.

A nondeterministic two-tape Turing machine M is designed to solve the path problem. The actions of M are summarized as follows:

1. The input is checked to determine if its format is that of a representation of a directed graph followed by the encoding of two nodes. If not, M halts and rejects the string.
2. The input is now assumed to have the form $R(G)000en(v_i)0en(v_j)$, where $R(G)$ is the representation of a directed graph G. If $v_i = v_j$, M halts in an accepting state.
3. The encoding of node v_i followed by *0* is written on tape 2.
4. Let v_s be the rightmost node encoded on tape 2. An arc from v_s to v_t is nondeterministically chosen from $R(G)$. If no such arc exists or v_t is already on the path encoded on tape 2, M halts in a rejecting state.
5. If $v_t = v_j$, then M halts in an accepting state. Otherwise, $en(v_t)0$ is written at the end of the string on tape 2 and the computation continues with step 4.

Steps 4 and 5 generate paths beginning with node v_i on tape 2. Since step 4 guarantees that only noncyclic paths are written on tape 2, every computation of M terminates. It follows that L(M) is recursive and the problem is decidable. □

A decision problem will frequently be defined by describing its instances and the condition that must be satisfied to obtain a positive answer. Using this method of problem definition, the path problem of Example 11.2.2 can be written

Path Problem for Directed Graphs
Input: Directed graph $G = (N, A)$, nodes $v_i, v_j \in N$
Output: yes; if there is a path from v_i to v_j in G
no; otherwise.

With the correspondence between solvable decision problems and recursive languages, should we speak of problems or languages? We will use the terminology of decision problems when the problem statement is given using high-level concepts and a representation is required to transform the problem instances into strings. When a problem is specified in terms of the acceptance of strings, we will use the terminology of recursive languages. In either case, a decision problem or a language is decidable if there is an algorithm that produces the correct answer for each problem instance or the correct membership value for each string, respectively.

11.3 Problem Reduction

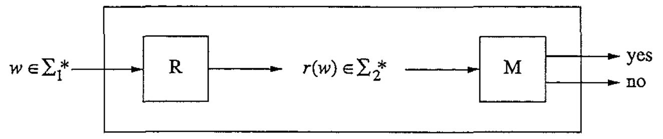
Reduction is a problem-solving technique commonly employed to avoid "reinventing the wheel" when encountering a new problem. The objective of a reduction is to transform the instances of the new problem into those of a problem that we already know how to solve. Reduction is an important tool for establishing the decidability of problems and, as we will see in Chapter 12, also for showing that certain problems do not have algorithmic solutions.

We will examine the mappings and requirements needed for problem reduction both on the level of languages and on the level of decision problems. We begin with the definition of reduction for membership in languages.

Definition 11.3.1

Let L be a language over Σ_1 and Q a language over Σ_2. L is **many-to-one reducible** to Q if there is a Turing computable function $r : \Sigma_1^* \to \Sigma_2^*$ such that $w \in$ L if, and only if, $r(w) \in$ Q.

If a language L is reducible to a decidable language Q by a function r, then L is also decidable. Let R be the Turing machine that computes the reduction and M the machine that accepts Q. The sequential execution of R and M on strings from Σ_1^* constitutes a solution to the membership problem for L.



Note that the reduction machine R does not determine membership in either L or Q; it simply transforms strings from Σ_1^* to Σ_2^*. Membership in Q is determined by M and membership in L by the combination of R and M.

To illustrate the reduction of one language to another, we will show that L $= \{x^i y^i z^k \mid i \geq 0, k \geq 0\}$ is reducible to Q $= \{a^i b^i \mid i \geq 0\}$. A reduction of L to Q may be described in the tabular form

Reduction	Input	Condition
L $= \{x^i y^i z^k \mid i \geq 0, k \geq 0\}$	$w \in \{x, y, z\}^*$	$w \in$ L
to	$\downarrow r$	if, and only if,
Q $= \{a^i b^i \mid i \geq 0\}$	$v \in \{a, b\}^*$	$r(w) \in$ Q

A string $w \in \{x, y, z\}^*$ is transformed to the string $r(w) \in \{a, b\}^*$ as follows:

i) If w has no x's or y's occurring after a z, replace each x with an a, each y with a b, and erase the z's.

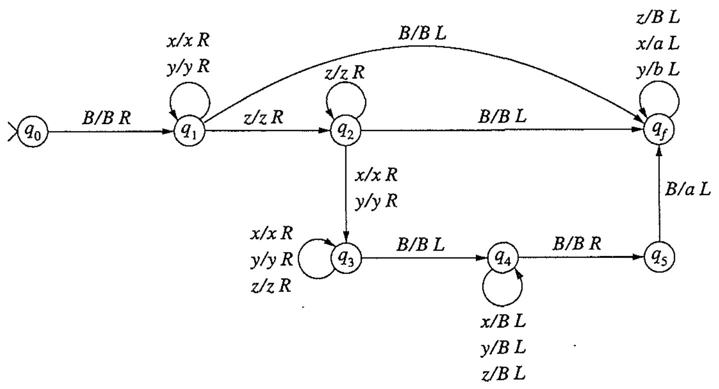
ii) If w has an x or y occurring after a z, erase the entire string and write a single a in the input position.

The following table gives the result of the transformation of several strings in Σ_1^*.

$w \in \Sigma_1^*$	In L?	$r(w) \in \Sigma_2^*$	In Q?
$xxyy$	yes	$aabb$	yes
$xxyyzzz$	yes	$aabb$	yes
$yxxyz$	no	$baab$	no
$xxzyy$	no	a	no
$zyzx$	no	a	no
λ	yes	λ	yes
zzz	yes	λ	yes

The examples show why the transformation is called a many-to-one reduction; multiple strings in Σ_1^* can map to the same string in Σ_2^*.

The Turing machine



performs the reduction of L to Q. Strings that have the form $(x \cup y)^*z^*$ are identified in states q_1 and q_2 and transformed in state q_f. Strings in which a z precedes an x or y are erased in state q_4 and an a is written on the tape in the transition to q_f.

Example 11.3.1

Consider the problem of accepting strings in the language $L = \{uu \mid u = a^i b^i c^i$ for some $i \geq 0\}$. The machine M in Example 8.2.2 accepts the language $\{a^i b^i c^i \mid i \geq 0\}$. We will sketch a reduction of the membership problem of L to that of recognizing a single instance of $a^i b^i c^i$. The original problem can then be solved using the reduction and the machine M. The reduction is obtained as follows:

1. The input string w is copied. The copy of w is used to determine whether $w = uu$ for some string $u \in \{a, b, c\}^*$.
2. If $w \neq uu$, then the tape is erased and a single a is written in the input position.
3. If $w = uu$, then the copy and the second u in the input string are erased leaving u in the input position.

If the input string w has the form uu, then $w \in L$ if, and only if, $u = a^i b^i c^i$ for some i. The reduction does not check the number or the order of the a's, b's, and c's; the machine M has been designed to perform that task.

If a string w does not have the form uu, the reduction produces the string a. This string is subsequently rejected by M, indicating that the input w is not in L. □

A decision problem P is many-to-one reducible to a problem Q if there is a transformation of problem instances of P into instances of the Q that preserves the affirmative and negative answers. Formally, a reduction transforms the string representations of the problem instances. Frequently, we will define a reduction directly on the problem instances, with the assumption that the modifications could be performed at the string level if we so desire. This technique, along with the implications for the string representations, is illustrated in the following example.

Example 11.3.2

We will show that the path problem for directed graphs, which was introduced in Example 11.2.2, is reducible to the problem:

Cycle with Fixed Node (CFN) Problem
Input: Directed graph G = (N, A), node $v_k \in N$
Output: yes; if there is a cycle containing v_k in G
no; otherwise.

The reduction requires constructing a graph G′ from G so that the existence of a path from v_i to v_j in G is equivalent to G′ having a cycle containing the node v_k. The first step in the

reduction is to identify the node v_k in the CFN problem to the initial node v_i of the path problem. With the selection of v_i as the node in the CFN problem, the reduction becomes

Reduction	Instances	Condition
Path Problem	Graph G, nodes v_i, v_j	G has a path from v_i to v_j
to	$\downarrow r$	if, and only if,
CFN Problem	Graph G′, node v_i	G′ has a cycle containing v_i

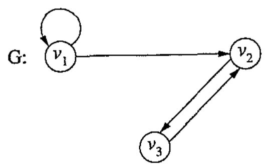
The graph G′ is obtained by modifying G as follows:

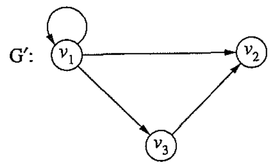
i) Deleting all arcs $[v_t, v_i]$ that enter v_i.
ii) Adding an arc $[v_j, v_i]$.

If there is a path from v_i to v_j in G, then there is a path in which v_i occurs only as the first node; cycles in the path that reenter v_i may be removed without changing either the initial or terminal node. Consequently, the deletion of the arcs $[v_t, v_i]$ does not affect the presence or absence of a path from v_i to v_j. After the arc deletion in step (i), there are no cycles that contain v_i since there are no arcs that enter v_i.

The addition of the arc $[v_j, v_i]$ in step (ii) will produce a cycle in G′ if, and only if, there is a path from v_i to v_j in the original graph G. Thus the modification of G is a reduction of the path problem to the CFN problem.

The reduction of instance G, v_3, v_1 of the path problem, where G is the graph from Example 11.2.2, produces





Since there is no path from v_3 to v_1 in G, G′ has no cycle containing v_3.

On the Turing machine level, an instance of the CFN problem consisting of a graph G′ and node v_i may be represented by the string $R(G')000en(v_i)$. The reduction of the instance G, v_3, v_1 of the path problem to the instance G′, v_3 of the CFN problem changes

$$R(G)000en(v_3)en(v_1) = 1101110011011001110111100111101110001111011$$

to

$$R(G')000en(v_3) = 110111001101100111011110011110110001111.$$

A Turing machine that performs the reduction must delete the representations of the arcs entering v_i, add the representation of the arc from v_j to v_i, and erase v_j from the end of the input. □

As with languages, reducing a decision problem P to a decidable problem Q shows that P is also decidable. A solution to P can be obtained by sequentially combining the reduction with the algorithm that solves Q.

11.4 The Church-Turing Thesis

The notion of algorithmic computation is not new. In fact, the word *algorithm* comes from the name of the 9th-century Arabian mathematician Abu Ja'far Muhammad ibn Musa al-Khwarizmi. In what is generally considered the first book on algebra, Al-Khwarizmi presented a set of rules for solving linear and quadratic equations. Step-by-step mechanistic procedures have been employed for centuries to describe calculations, processes, and mathematical derivations. This informal usage matured in the early 20th century when mathematicians sought to precisely determine the meaning, capabilities, and limitations of algorithmic computation.

The investigation into the properties of computability led to a number of approaches and formalisms for performing algorithmic computation. Effective procedures have been defined by rules that transform strings, by the evaluation of functions, by the computations of abstract machines, and more recently, by programs in high-level programming languages. Examples of each of these types of systems include

- String Transformations: Post systems [Post, 1936], Markov systems [Markov, 1961], unrestricted grammars
- Evaluation of Functions: partial and μ-recursive functions [Gödel, 1931; Kleene, 1936], lambda calculus [Church, 1941]
- Abstract Computing Machines: Register Machines [Shepherdson, 1963], Turing machines
- Programming languages: while-programs [Kfoury et al., 1982], TM from Chapter 9

While-programs, listed in a final category, are programs that can be written in a minimal programming language that consists of assignment, conditional, for, and while statements. Having a small number of statements facilitates the analysis of programs, but while-programs have the same computational ability as programs in standard programming languages such as C, C++, Java, and so on.

We have used Turing machines as the computational framework for solving decision problems. However, any of the other algorithmic systems could just as well have been selected. Would this in any way have changed our ability to solve problems? Ideally the answer should be no—the existence of a solution to a problem should be an inherent feature of the problem itself and not an artifact of our choice of an algorithmic system. The Church-Turing Thesis validates this intuition.

What do all of the previously mentioned algorithmic systems have in common? It has been shown that they are all capable of performing precisely the same computations. This claim may seem remarkable, since these systems were designed to perform different types of operations on different types of data. However, you have already seen one example of

the equivalence and will see another in Chapter 13. In Section 10.1 we proved that the computation of a Turing machine can be simulated by the rules of an unrestricted grammar. Conversely, any language generated by an unrestricted grammar is accepted by a Turing machine. Consequently, the power of Turing machines for recognizing languages is identical to that of unrestricted grammars for generating languages. In Chapter 13 we will show that the algorithmic approach to the definition and evaluation of number-theoretic functions introduced by Gödel and Kleene produces exactly the functions that can be computed by Turing machines.

The realization that the various approaches to effective computation produced systems that have the same computational power led to the belief that the capabilities of these systems define the bounds of algorithmic computation. There is no single definition of *algorithm* and no single system for performing effective computation. However, there is a well-defined bound on what can be accomplished in any of these systems. The Church-Turing Thesis formalizes this belief in a general statement about the capabilities and limitations of algorithmic computation. We will present three variations, one corresponding to each of the types of computations that we have studied. We begin with the interpretation of the Church-Turing Thesis for decision problems.

The Church-Turing Thesis for Decision Problems There is an effective procedure to solve a decision problem if, and only if, there is a Turing machine that halts for all input strings and solves the problem.

A solution to a decision problem requires the computation to return an answer for every instance of the problem. Relaxing this restriction, we obtain the notion of a partial solution. A partial solution to a decision problem P is a not necessarily complete but otherwise effective procedure that returns an affirmative response for every problem instance $p \in$ P whose answer is yes. If the answer to p is negative, however, the procedure may return no or fail to produce an answer. That is, the computation recognizes affirmative instances.

Just as a solution to a decision problem can be formulated as a question of membership in a recursive language, a partial solution to a decision problem is equivalent to the question of membership in a recursively enumerable language. The Church-Turing Thesis encompasses algorithms that recognize languages as well as those that decide languages.

The Church-Turing Thesis for Recognition Problems A decision problem P is partially solvable if, and only if, there is a Turing machine that accepts precisely the instances of P whose answer is yes.

Turing machines compute functions using the symbols on the tape when the machine halts to define the result of a computation. A functional approach to solving decision problems uses the computed values one and zero to designate affirmative and negative responses. The method of specifying the answer does not affect the set problems that have Turing machine solutions (Exercise 9.4). Thus the formulation of the Church-Turing Thesis in terms of computable functions subsumes and extends the two previous versions of the thesis.

The Church-Turing Thesis for Computable Functions A function f is effectively computable if, and only if, there is a Turing machine that computes f.

After establishing the equivalence of Turing computable functions and μ-recursive functions in Chapter 13, we will give a more concise version of the Church-Turing Thesis and present a natural generalization from computable number-theoretic functions to computable functions on arbitrary sets.

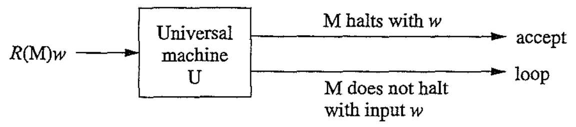
To appreciate the content of the Church-Turing Thesis, it is necessary to understand the nature of the assertion. The Church-Turing Thesis is not a mathematical theorem; it cannot be proved. This would require a formal definition of the intuitive notion of an effective procedure. The claim could, however, be disproved. This could be accomplished by discovering an effective procedure that cannot be computed by a Turing machine. The equivalence of Turing machines to other algorithmic systems, the robustness of the Turing machine architecture, and the lack of a counterexample highlight an impressive pool of evidence that suggests that such a procedure will not be found.

A proof by the Church-Turing Thesis is a shortcut often taken in establishing the existence of a decision algorithm. Rather than constructing a Turing machine solution to a decision problem, we describe an intuitively effective procedure that solves the problem. The Church-Turing Thesis guarantees that a Turing machine can be designed to solve the problem. We have tacitly been using the Church-Turing Thesis in this manner throughout the presentation of Turing computability. For complicated machines, we simply gave a description of the actions of a computation of the machine. We assumed that the complete machine could then be explicitly constructed, if desired.

11.5 A Universal Machine

One of the most significant advances in computer design occurred in the mid-1940s with the development of the stored program model of computation. Early computers were designed to perform a single task; the input could vary, but the same program would be executed for each input. Making a change to the instructions would frequently require reconfiguration of the hardware. In the stored program model, the instructions are electronically loaded into memory along with the data. A computation in a stored program computer is a cycle consisting of the retrieval of an instruction from memory followed by its execution.

The Turing machines in the preceding chapters, like the early computers, were designed to execute a single set of instructions. The Turing machine architecture has its own version of the stored program concept, which preceded the first stored program computer by a decade. A **universal Turing machine** is designed to simulate the computations of an arbitrary Turing machine M. To do so, the input to the universal machine must contain a representation of the machine M and the string w to be processed by M. For simplicity, we will assume that M is a standard Turing machine that accepts by halting. The action of a universal machine U is depicted by



where R(M) is the representation of the machine M. The output labeled loop indicates that the computation of U does not terminate. If M halts and accepts input w, U does the same. If M does not halt with w, neither does U. The machine U is called universal since the computation of any Turing machine M can be simulated by U.

The first step in the construction of a universal machine is to design the string representation of a Turing machine. Because of the ability to encode arbitrary symbols as strings over $\{0, 1\}$, we consider Turing machines with input alphabet $\{0, 1\}$ and tape alphabet $\{0, 1, B\}$. The states of a Turing machine are assumed to be named $\{q_0, q_1, \ldots, q_n\}$, with q_0 the start state.

A Turing machine M is defined by its transition function. A transition of a standard Turing machine has the form $\delta(q_i, x) = [q_j, y, d]$, where $q_i, q_j \in Q$; $x, y \in \Gamma$; and $d \in \{L, R\}$. We encode the elements of M using strings of 1's:

Symbol	Encoding
0	1
1	11
B	111
q_0	1
q_1	11
$\vdots$	$\vdots$
q_n	1^{n+1}
L	1
R	11

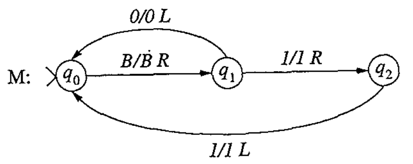
Let $en(z)$ denote the encoding of a symbol z. A transition $\delta(q_i, x) = [q_j, y, d]$ is encoded by the string

$$en(q_i)0en(x)0en(q_j)0en(y)0en(d).$$

The 0's separate the components of the transition. A representation of the machine is constructed from the encoded transitions. Two consecutive 0's are used to separate transitions. The beginning and end of the representation are designated by three 0's.

Example 11.5.1

The computation of the Turing machine



halts for the null string and strings that begin with *1*, and does not terminate for strings beginning with *0*. The encoded transitions of M are given in the following table.

Transition	Encoding
$\delta(q_0, B) = [q_1, B, R]$	*101110110111011*
$\delta(q_1, 0) = [q_0, 0, L]$	*1101010101*
$\delta(q_1, 1) = [q_2, 1, R]$	*110110111011011*
$\delta(q_2, 1) = [q_0, 1, L]$	*1110110101101*

The machine M is represented by the string

00010111011011101100110101010100110110111011011001110110101101000. □

A Turing machine can be constructed to determine whether an arbitrary string $u \in \{0, 1\}^*$ is the encoding of a deterministic Turing machine. The computation examines u to see if it consists of a prefix *000* followed by a finite sequence of encoded transitions separated by *00*'s followed by *000*. A string that satisfies these conditions is the representation of some Turing machine M. The machine M is deterministic if the combination of the state and input symbol in every encoded transition is distinct.

We will now outline the design of a three-tape, deterministic universal machine U. A computation of U begins with the input on tape 1. If the input string has the form $R(\mathrm{M})w$, the computation of M with input w is simulated on tape 3. A computation of U consists of the following actions:

1. If the input string does not have the form $R(\mathrm{M})w$ for a deterministic Turing machine M and string w, U moves to the right forever.
2. The string w is written on tape 3 beginning at position one. The tape head is then repositioned at the leftmost square of the tape. The configuration of tape 3 is the initial configuration of a computation of M with input w.
3. A single *1*, the encoding of state q_0, is written on tape 2.
4. A transition of M is simulated on tape 3. The transition of M is determined by the symbol scanned on tape 3 and the state encoded on tape 2. Let x be the symbol from tape 3 and q_i the state encoded on tape 2.
   a) Tape 1 is scanned for a transition whose first two components match $en(q_i)$ and $en(x)$. If there is no such transition, U halts accepting the input.
   b) If tape 1 contains an encoded transition $en(q_i)0en(x)0en(q_j)0en(y)0en(d)$, then
      i) $en(q_i)$ is replaced by $en(q_j)$ on tape 2.
      ii) The symbol y is written on tape 3.
      iii) The tape head of tape 3 is moved in the direction specified by d.
5. The computation continues with step 4 to simulate the next transition of M.

Theorem 11.5.1

The language $L_H = \{R(\mathrm{M})w \mid \mathrm{M}$ halts with input $w\}$ is recursively enumerable.

Proof. The universal machine accepts strings of the form $R(\mathrm{M})w$ where $R(\mathrm{M})$ is the representation of a Turing machine and M halts when run with input w. For all other strings, the computation of U does not terminate. Thus the language of U is L_H. ■

The language L_H is known as the language of the Halting Problem. A string is in L_H if it is the combination of the representation of a Turing M and a string w such that M halts when run with w.

The computation of the universal machine U with input $R(\mathrm{M})w$ simulates the computation M with input w. The ability to obtain the results of one machine via the computations of another facilitates the design of complicated Turing machines. When we say that a Turing machine M′ "runs machine M with input w," we mean that M′ is supplied with $R(\mathrm{M})$ and w and simulates the computation of M in the manner of the universal machine.

Example 11.5.2

A solution to the decision problem

Halts on *n*'th Transition Problem
Input: Turing machine M, string w, integer n
Output: yes; if the computation of M with input w performs exactly n transitions before halting
no; otherwise.

can be obtained by simulating the computations of M. Intuitively, a solution "runs M with input w" and counts the transitions of M.

A machine U′ that solves this problem can be constructed by adding a fourth tape to the universal machine to record the number of transitions in a computation of M. A problem instance will be represented by a string of the form $R(\mathrm{M})w0001^{n+1}$ with the unary representation of n separated from $R(\mathrm{M})w$ by three zeroes. The computation of U′ with input string u consists of the following actions:

1. If the input string u does not end with 0001^{n+1}, U′ halts rejecting the input.
2. The string 1^n is written on tape 4 beginning in position one; 0001^{n+1} is erased from the end of the string on tape 1; and the tape head on tape 4 moves to position one.
3. If the string remaining on tape 1 does not have the form $R(\mathrm{M})w$, U′ halts rejecting the input.
4. The string w is copied to tape 3 and the encoding of state q_0 is written on tape 2.
5. Following the strategy of the universal machine, tape 1 is searched for a transition that matches the symbol x scanned on tape 3 and the state q_i encoded on tape 2.

a) If there is no transition for q_i, x and a *1* is read on tape 4, then U′ halts rejecting the input.

b) If there is no transition for q_i, x and a blank is read on tape 4, then U′ halts accepting the input.

c) If there is a transition $\delta(q_i, x)$ encoded on tape 1 and a blank is read on tape 4, then U′ halts rejecting the input.

d) If there is a transition $\delta(q_i, x)$ encoded on tape 1 and a *1* is read on tape 4, then the transition is simulated on tapes 2 and 3 and the tape head on tape 4 is moved one square to the right.

6. The computation continues with step 5 to examine the next transition of M.

If M halts prior to the nth transition, $R(\mathrm{M})w0001^{n+1}$ is rejected in step 5 (a). After the simulation of n transitions of M, the counter on tape 4 reads a blank. If M has no applicable transition at this point, U′ accepts. Otherwise, the input is rejected in step 5 (c). □

Exercises

1. Give a state diagram of a Turing machine M that solves the miser problem from Section 11.1. A set of coins is represented as an element of $\{n, d, q\}^*$ where n, d, and q designate a nickel, a dime, and a quarter, respectively.

In Exercises 2 through 7, describe a Turing machine that solves the specified decision problem. Use Example 11.2.2 as a model for defining the actions of a computation of the machine. You need not explicitly construct the transition function nor the state diagram of your solution. You may use multitape Turing machines and nondeterminism in your solutions.

2. Design a two-tape Turing machine that determines whether two strings u and v over $\{0, 1\}$ are identical. The computation begins with $BuBvB$ on the tape and should require no more than $3(length(u) + 1)$ transitions.

3. Using the unary representation of the natural numbers, design a Turing machine whose computations decide whether a natural number is prime.

4. Using the unary representation of the natural numbers, design a Turing machine that solves the "2^n" problem. *Hint*: The input is the representation of a natural number i and the output is yes if $i = 2^n$ for some n, no otherwise.

5. A directed graph is said to be *cyclic* if it contains at least one cycle. Using the representation of a directed graph from Section 11.2, design a Turing machine whose computations decide whether a directed graph is cyclic.

6. A tour in a directed graph is a path $p_0, p_1, \ldots, p_n$ in which

   i) $p_0 = p_n$.

   ii) For $0 < i,\ j \le n,\ i \ne j$ implies $p_i \ne p_j$.

   iii) Every node in the graph occurs in the path.

   That is, a tour visits every node exactly once and ends where it begins. Design a Turing machine that decides whether a directed graph contains a tour. Use the representation of a directed graph given in Section 11.2.

*7. Let G = (V, Σ, P, *S*) be a regular grammar.

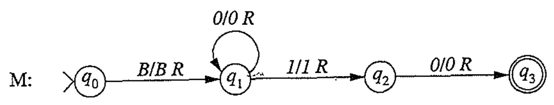
   a) Construct a representation for the grammar G over $\{0, 1\}$.

   b) Design a Turing machine that decides whether a string $w \in \Sigma^*$ is in L(G). The use of nondeterminism facilitates the construction of the desired machine.

8. Construct a Turing machine that reduces the language L to Q. In each case the alphabet of L is $\{x, y\}$ and the alphabet of Q is $\{a, b\}$.

a) $L = (xy)^*$	$Q = (aa)^*$
b) $L = x^+y^*$	$Q = a^+b$
c) $L = \{x^iy^{i+1} \mid i \ge 0\}$	$Q = \{a^ib^i \mid i \ge 0\}$
d) $L = \{x^iy^jz^i \mid i \ge 0,\ j \ge 0\}$	$Q = \{a^ib^i \mid i \ge 0\}$
e) $L = \{x^i(yy)^i \mid i \ge 0\}$	$Q = \{a^ib^i \mid i \ge 0\}$
f) $L = \{x^iy^ix^i \mid i \ge 0\}$	$Q = \{a^ib^i \mid i \ge 0\}$

9. Let M be the Turing machine



   a) What is L(M)?

   b) Give the representation of M using the encoding from Section 11.5.

10. Construct a Turing machine that decides whether a string over $\{0, 1\}^*$ is the encoding of a nondeterministic Turing machine. What would be required to change this to a machine that decides whether the input is the representation of a deterministic Turing machine?

11. Design a Turing machine with input alphabet $\{0, 1\}$ that accepts an input string u if

   i) $u = R(\mathrm{M})w$ for some Turing machine M and input string w, and

   ii) when M is run with input w, there is a transition in the computation that prints a *1*.

   Your machine need not halt for all inputs.

12. Given an arbitrary Turing machine M and input string w, will the computation of M with input w halt in fewer than 100 transitions? Describe a Turing machine that solves this decision problem.
13. Show that the decision problem

    Input: Turing machine M
    Output: yes; if the third transition of M prints a blank when run with a blank tape
    no; otherwise.

    is decidable. The answer for a Turing machine M is no if M halts prior to its third transition.
* 14. Show that the decision problem

    Input: Turing machine M
    Output: yes; if there is some string $w \in \Sigma^*$ for which the computation of M takes more than 10 transitions
    no; otherwise.

    is decidable.
15. The universal machine introduced in Section 11.5 was designed to simulate the actions of Turing machines that accept by halting. Consequently, the representation scheme $R(M)$ did not encode accepting states.
    a) Extend the representation $R(M)$ of a Turing machine M to explicitly encode the accepting states of M.
    b) Design a universal machine U_f that accepts input of the form $R(M)w$ if the machine M accepts input w by final state.

Bibliographic Notes

Turing [1936] envisioned the theoretical computing machine he designed to be capable of performing all effective computations. This viewpoint, now known as the Church-Turing Thesis, was formalized by Church [1936]. Turing's 1936 paper also included the design of a universal machine. The original plans for the development of a stored program computer were reported by von Neumann [von Neumann, 1945], and the first working models appeared in 1949.

In our construction of the universal machine, we limited the input and tape alphabets of the Turing machines to {*0*, *1*} and {*0*, *1*, *B*}, respectively. A proof that an arbitrary Turing machine can be simulated by a machine with these alphabets can be found in Hopcroft and Ullman [1979].

CHAPTER 12
Undecidability

The Church-Turing Thesis asserts that a Turing machine can be designed to solve any decision problem that is solvable by any effective procedure. A Turing machine computation is not encumbered by the physical restrictions that are inherent in any "real" computing device. Thus the existence of a Turing machine solution to a decision problem depends entirely on the nature of the problem itself and not on the availability of memory or central processor time. The Church-Turing Thesis also has consequences for undecidability. If a problem cannot be solved by a Turing machine, it cannot be solved by any effective procedure. A decision problem that has no algorithmic solution is said to be **undecidable**.

In Section 9.5 it was shown that there are only countably many Turing machines. The number of languages over a nonempty alphabet, however, is uncountable. It follows that there are languages whose membership problem is undecidable. The comparison of cardinalities ensures us of the existence of undecidable decision problems but gives us no idea of what such a problem might look like. In this chapter we show that some particular decision problems concerning the computational capabilities of Turing machines, derivations in grammars, and even playing a game with dominoes are undecidable.

The first problem that we consider is the Halting Problem for Turing Machines. To appreciate the significance of the Halting Problem, we will describe it in terms of C programs rather than Turing machines. The Halting Problem for C Programs can be stated as

Halting Problem for C Programs
Input: C program *Prog*,
input file *inpt* for *Prog*
Output: yes; if *Prog* halts when run with input *inpt*
no; otherwise.

If the Halting Problem for C Programs were decidable, a bane of all programmers—the infinite loop—would be a thing of the past. The execution of a program would become a two-step process:

1. running the algorithm that solves the Halting Problem on *Prog* and *inpt*;
2. if the algorithm indicates *Prog* will halt, then running *Prog* with *inpt*.

A solution to the Halting Problem does not tell us the result of the computation, only that a result will be produced. After receiving an affirmative response from the halting algorithm, the result could be obtained by running *Prog* with the input file *inpt*. Unfortunately, the Halting Problem for C Programs, like its counterpart for Turing machines, is undecidable.

Throughout the first four sections of this chapter, we will consider Turing machines with input alphabet $\{0, 1\}$ and tape alphabet $\{0, 1, B\}$. The restriction on the alphabets imposes no limitation on the computational capabilities of Turing machines since the computation of an arbitrary Turing machine M can be simulated by a machine with these restricted alphabets. The simulation requires encoding the symbols of M as strings over $\{0, 1\}$. This is precisely the approach employed by digital computers, which use the ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code), or Unicode encodings to represent characters as binary strings.

12.1 The Halting Problem for Turing Machines

The most famous of the undecidable problems is concerned with the properties of Turing machines themselves. The Halting Problem may be formulated as follows: Given an arbitrary Turing machine M with input alphabet Σ and a string $w \in \Sigma^*$, will the computation of M with input w halt? We will show that there is no algorithm that solves the Halting Problem. The undecidability of the Halting Problem is one of the fundamental results in the theory of computer science.

It is important to understand the statement of the problem. We may be able to determine that a particular Turing machine will halt for a given string. In fact, the exact set of strings for which a Turing machine halts may be known. For example, the machine in Example 8.3.1 halts for all and only the strings that contain aa as a substring. A solution to the Halting Problem, however, requires a general algorithm that answers the halting question for every possible combination of Turing machine and input string.

Since the Halting Problem asks a question about a Turing machine, the input must contain a Turing machine, or more precisely the representation of a Turing machine. We will use the Turing machine representation developed in Section 11.5, which encodes a Turing machine with input alphabet $\{0, 1\}$ as a string over $\{0, 1\}$. The proof of the undecidability of the Halting Problem does not depend upon the features of this particular encoding. The argument is valid for any representation that encodes a Turing machine as a string over its input alphabet. As before, the representation of a machine M is denoted $R(\mathrm{M})$.

The proof of the undecidability of the Halting Problem is by contradiction. We assume that there is a Turing machine H that solves the Halting Problem. We then make several simple modifications to H to obtain a new machine D that produces a self-referential contradiction; an impossible situation occurs when the machine D is run with its own representation as input. Since the assumption of the existence of a machine H that solves the Halting Problem produces a contradiction, the Halting Problem is not solvable.

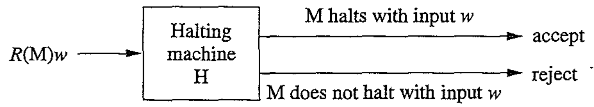
Theorem 12.1.1

The Halting Problem for Turing Machines is undecidable.

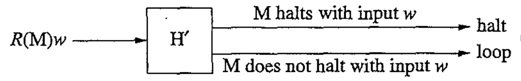
Proof. Assume that the Turing machine H solves the Halting Problem. A string $z \in \{0, 1\}$ is accepted by H if

i) z consists of the representation of a Turing machine M followed by a string w and
ii) the computation of M with input w halts.

If either of these conditions is not satisfied, H rejects the input. The operation of the machine H is depicted by the diagram

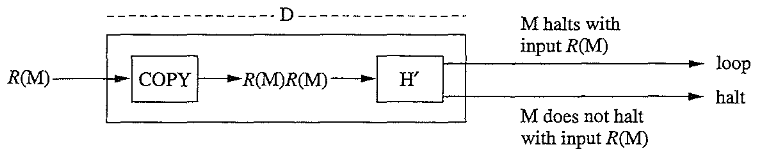


The machine H is modified to construct a new Turing machine H′. The computations of H′ are the same as H except H′ continues when H halts in an accepting state. At that point, H′ moves to the right forever. The transition function of H′ is obtained from that of H by adding transitions that cause H′ to move indefinitely to the right upon entering an accepting configuration of H. The action of H′ may be depicted by

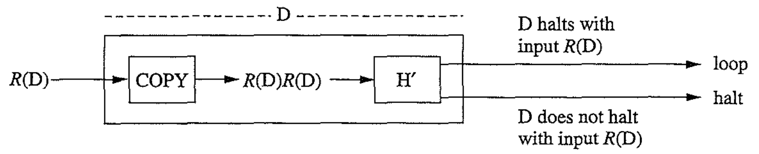


From this point on in the proof, we are concerned only with whether a computation halts or continues indefinitely. The latter case is denoted by the word *loop* in the diagrams.

The machine H′ is combined with a copy machine to construct another Turing machine D. The input to D is a Turing machine representation $R(\mathrm{M})$. A computation of D begins by creating the string $R(\mathrm{M})R(\mathrm{M})$ from the input $R(\mathrm{M})$. The computation continues by running H′ on $R(\mathrm{M})R(\mathrm{M})$.



The input to the machine D may be the representation of any Turing machine with alphabet $\{0, 1, B\}$. In particular, D itself is such a machine. Consider a computation of D with input $R(\mathrm{D})$. Rewriting the previous diagram with M replaced by D and $R(\mathrm{M})$ by $R(\mathrm{D})$, we get



Examining the diagram, we see that D halts with input $R(\mathrm{D})$ if, and only if, D does not halt with input $R(\mathrm{D})$. This is obviously impossible. However, the machine D can be constructed directly from a machine H that solves the Halting Problem. The assumption that the Halting Problem is decidable produces the preceding contradiction. Therefore, we conclude that the Halting Problem is undecidable. ■

The contradiction in the preceding proof uses self-reference and diagonalization. To obtain the standard relational table for a diagonalization argument, we consider every string $v \in \{0, 1\}^*$ to represent a Turing machine; if v does not have the form $R(\mathrm{M})$, the one-state Turing machine with no transitions is assigned to v. Thus the Turing machines can be listed $\mathrm{M}_0, \mathrm{M}_1, \mathrm{M}_2, \mathrm{M}_3, \mathrm{M}_4, \ldots$ corresponding to strings $\lambda, 0, 1, 00, 01, \ldots$. Now consider a table that lists the Turing machines along the horizontal and vertical axes. The i, jth entry of the table is

$$\begin{cases} 1 & \text{if } \mathrm{M}_i \text{ halts when run with } R(\mathrm{M}_j) \\ 0 & \text{if } \mathrm{M}_i \text{ does not halt when run with } R(\mathrm{M}_j). \end{cases}$$

The diagonal of the table represents the answers to the self-referential question, "Does M_i halt when run on itself?" The machine D was constructed to produce a contradiction in response to that question.

A similar argument can be used to establish the undecidability of the Halting Problem for Turing Machines with arbitrary alphabets. The essential feature of this approach is the ability to encode the transitions of a Turing machine as a string over its own input alphabet. Two symbols are sufficient to construct such an encoding.

The undecidability of the Halting Problem and the ability of the universal machine to simulate computations of Turing machines combine to show that the recursive languages are

a proper subset of the recursively enumerable languages. Corollary 12.1.2 is the restatement of the undecidability of the Halting Problem in the terminology of recursive languages.

Corollary 12.1.2

The language $\mathrm{L_H} = \{R(\mathrm{M})w \mid R(\mathrm{M})$ is the representation of a Turing machine M and M halts with input $w\}$ over $\{0, 1\}^*$ is not recursive.

Corollary 12.1.3

The recursive languages are a proper subset of the recursively enumerable languages.

Proof. The universal machine U accepts $\mathrm{L_H}$; a string is accepted by U only if it is of the form $R(\mathrm{M})w$ and M halts when run with input w. The acceptance of $\mathrm{L_H}$ by the universal machine demonstrates that $\mathrm{L_H}$ is recursively enumerable, while Corollary 12.1.2 established that $\mathrm{L_H}$ is not recursive. ■

In Exercise 8.26 it was shown that a language L is recursive if both L and $\overline{\mathrm{L}}$ are recursively enumerable. Combining this with Corollary 12.1.2 yields

Corollary 12.1.4

The language $\overline{\mathrm{L_H}}$ is not recursively enumerable.

Corollary 12.1.4 tells us that there is no algorithm that can either accept or recognize the strings of the language $\overline{\mathrm{L_H}}$. From a pattern recognition perspective, machines are designed to detect patterns that are common to all elements in a set of strings. When a language is not recursively enumerable, any common pattern among the elements of the language is too complex to be detected algorithmically.

12.2 Problem Reduction and Undecidability

Reduction was introduced in Chapter 11 as a tool for constructing solutions to decision problems. A decision problem P is reducible to Q if there is a Turing computable function r that transforms instances of P into instances of Q, and the transformation preserves the answer to the problem instance of P. As in Chapter 11, we will use a table of the form

Reduction	Input	Condition
P	instances $p_0, p_1, \ldots$	the answer to p_i is yes
to	$\downarrow r$	if, and only if,
Q	instances $q_0, q_1, \ldots$	the answer to $r(p_i)$ is yes

to describe the components and conditions of a reduction of P to Q.

Reduction has important implications for undecidability as well for decidability. If P is undecidable and reducible to a problem Q, then Q must also be undecidable. If Q were decidable, combining the reduction of P to Q with the algorithm that solves Q produces a decision procedure for P as follows: For an input p_i to P

i) Use the reduction to transform p_i to $r(p_i)$.
ii) Use the algorithm for Q to determine the answer for $r(p_i)$.

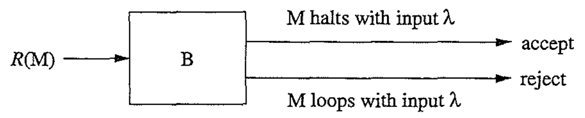
Since r is a reduction, the answer to the decision problem P for input p_i is the same as the answer to $r(p_i)$ for problem Q. The sequential execution of the reduction and the algorithm that solves Q produces a solution to P. This is a contradiction since P was known to be undecidable. Consequently, our assumption that Q is decidable must be false.

The *Blank Tape Problem* is the problem of deciding whether a Turing machine halts when a computation is initiated with a blank tape. The Blank Tape Problem is a special case of the Halting Problem since it is concerned only with the question of halting when the input is the null string. We will show that the Halting Problem is reducible to the Blank Tape Problem and, consequently, that the Blank Tape Problem is undecidable.

Theorem 12.2.1

There is no algorithm that determines whether an arbitrary Turing machine halts when a computation is initiated with a blank tape.

Proof. Assume that there is a machine B that solves the Blank Tape Problem. Such a machine can be represented

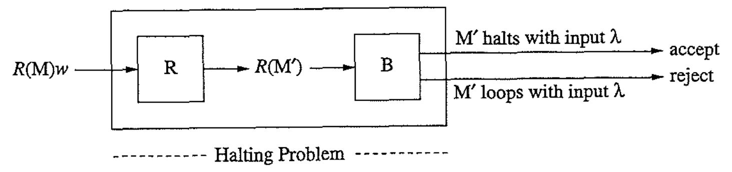


The reduction of the Halting Problem to the Blank Tape Problem is accomplished by a machine R. The input to R is the representation of a Turing machine M followed by an input string w. The result of a computation of R is the representation of a machine M′ that

1. writes w on a blank tape,
2. returns the tape head to the initial position with the machine in the start state of M, and
3. runs M.

R(M′) is obtained by adding encoded transitions to R(M) and suitably renaming the start state of M. The machine M′ has been constructed so that it halts when run with a blank tape if, and only if, M halts with input w.

A new machine is constructed by adding R as a preprocessor to B. Sequentially running the machines R and B produces the composite machine

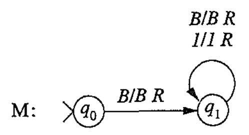


Tracing a computation, we see that the composite machine solves the Halting Problem. Since the preprocessor R reduces the Halting Problem to the Blank Tape Problem, the Blank Tape Problem is undecidable. ■

The preprocessor R, which performs the reduction of the Halting Problem to the Blank Tape Problem, modifies the representation of a Turing machine M to construct the representation of a Turing machine M′. Example 12.2.1 shows the result of a transformation performed by the preprocessor R.

Example 12.2.1

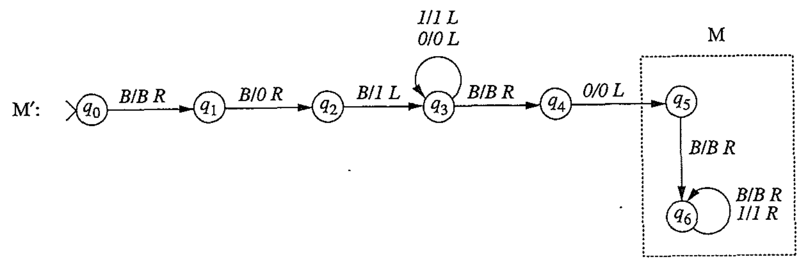
Let M be the Turing machine



that halts whenever the input string contains *0*. The encoding R(M) of M is

0001011101101110110011011101101110110011011011011000.

With input R(M)*01*, the preprocessor R constructs the encoding of the Turing machine M′.



When run with a blank tape, the first five states of M′ are used to write *01* in the input position. A copy of the machine M is then run with tape $B01B$. It is clear from the construction that M halts with input *01* if, and only if, M′ halts when run with a blank tape. □

Since the Blank Tape Problem is a subproblem of the Halting Problem, this is an ideal time to consider the relationship between problems, subproblems, and undecidability. Each of following problems is obtained by fixing one of the inputs of the Halting Problem:

Subproblem	Input	Decidable?
Blank Tape Problem	$R(M)$, (input string fixed)	Undecidable
Halting of the universal machine U	(machine fixed), $R(M)w$	Undecidable
Halting of M from Example 8.3.1	(machine fixed), w	Decidable

The Halting Problem for the universal machine asks if U will halt with input $R(M)w$. A solution to this problem would determine if an arbitrary Turing machine M halts with input w and thus provide a solution to the Halting Problem. The preceding table shows that subproblems of an undecidable problem may or may not be undecidable depending upon which features of the problem are retained. On the other hand, if Q is a subproblem of a decision problem P and Q is undecidable, then P is necessarily undecidable; any algorithm that solves P is automatically a solution to all of its subproblems.

The reduction of the Halting Problem to the Blank Tape Problem was accomplished by a Turing computable function r that transformed strings of the form $R(M)w$ to a string $R(M')$. Theorem 12.2.1 and Example 12.2.1 showed how the Turing machine representation $R(M)$ is modified to produce $R(M')$. In the remainder of examples, we will give a high-level explanation of the reduction and omit the details of the manipulation of the string representations.

12.3 Additional Halting Problem Reductions

We have shown that there is no algorithm that determines whether a Turing machine computation will halt, either with an arbitrary string or with a blank tape. There are many other questions that we could ask about Turing machines: "Does a computation enter a particular state?" Or "Does a computation print a particular symbol on its final transition?" And so on. Many such questions can also be shown to be undecidable using reduction and the undecidability of the Halting Problem.

We will demonstrate the general strategy for establishing the undecidability of such questions by considering the problem of whether a Turing machine computation reenters its start state. A computation that reenters the start state begins $q_0BwB \overset{*}{\vdash} uq_0vB$. The computation need not halt in the start state or even halt at all; all that is required is that the machine returns to state q_0 at some point after the start of the computation.

We will show that the Reenter Problem is undecidable by reducing the Halting Problem to it. The reduction has the form

Reduction	Input	Condition
Halting Problem to Reenter Problem	Turing machine M, string w ↓ Turing machine M′, string w	M halts with input w if, and only if, M′ reenters its start state when run with w

As indicated, we will use the same string w as the input for the machine M in the Halting Problem and the machine M′ in the Reenter Problem.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ and w be an instance of the Halting Problem. We must construct a machine M′ that reenters its start state when run with w if, and only if, M halts when run with w. First we note that, in an arbitrary Turing machine, the halting of a computation is in no way connected to whether the computation reenters the start state. In designing the reduction, it is our task to connect them.

The idea behind the construction of the machine M′ is to start with M, add a new start state q_0' that has the same transitions as q_0, and add a transition to q_0' for every halting configuration of M. Formally, M′ is defined from the components of M:

$$Q' = (Q \cup \{q_0'\}),\ \Sigma' = \Sigma,\ \Gamma' = \Gamma,\ F' = F$$

$$\delta'(q_i, x) = \delta(q_i, x) \text{ if } \delta(q_i, x) \text{ is defined}$$

$$\delta'(q_0', x) = \delta(q_0, x) \text{ for all } x \in \Gamma$$

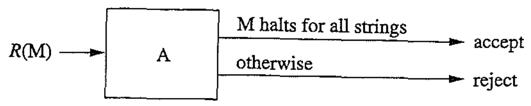
$$\delta'(q_i, x) = [q_0', x, R] \text{ if } \delta(q_i, x) \text{ is undefined}$$

with q_0' the start state of M′. If the computation of M halts with input w, the corresponding computation of M′ takes one additional transition and reenters q_0'. If M does not halt, a transition to q_0' is never taken and M′ does not reenter its start state. The construction transforms the question of whether M halts with input w to the question of whether M′ reenters its start state when run with w. It follows that the Reenter Problem is also undecidable.

Example 12.3.1

A proof by contradiction is used to show that the problem of determining whether an arbitrary Turing machine halts for all input strings is undecidable. Assume that there is a Turing machine A that solves this problem. The input to such a machine is a string $v \in \{0, 1\}^*$. The input is accepted if $v = R(M)$ for some Turing machine M that halts for all input strings. The input is rejected if either v is not the representation of a Turing machine or it is the representation of a machine that does not halt for some input string.

The computation of machine A can be depicted by



Problem reduction is used to create a solution to the Halting Problem from the machine A. It follows that the 'halts for all strings' problem is undecidable.

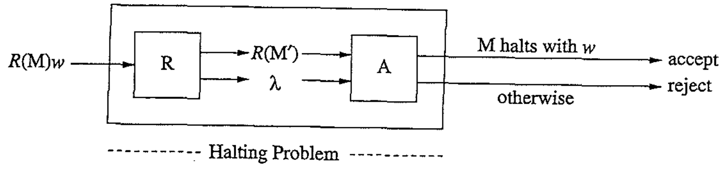
The language of the Halting Problem consists of strings of the form $R(\mathrm{M})w$, where the machine M halts when run with input w. The reduction is accomplished by a machine R. The first action of R is to determine whether the input string has the expected format of the representation of some Turing machine M followed by a string w. If the input does not have this form, R erases the input, leaving the tape blank.

When the input has the form $R(\mathrm{M})w$, the computation of R constructs the encoding of a machine M′ that, when run with any input string y,

1. erases y from the tape,
2. writes w on the tape, and
3. runs M on w.

$R(\mathrm{M}')$ is obtained from $R(\mathrm{M})$ by adding the encoding of two sets of transitions: one set that erases the input that is initially on the tape and another set that then writes the w in the input position. The machine M′ has been constructed to completely ignore its input. Every computation of M′ halts if, and only if, the computation of M with input w halts.

The machine consisting of the combination of R and A



provides a solution to the Halting Problem. If the input does not have the form $R(\mathrm{M})w$, the null string is produced by R and subsequently rejected by A. Otherwise R generates $R(\mathrm{M}')$. Tracing the sequential operation of the machines, the input is accepted if, and only if, it is the representation of a Turing machine M that halts when run with w.

Since the Halting Problem is undecidable and the reduction machine R is constructible, we conclude that there is no machine A that solves the 'halts for all strings' problem. □

The relationship between Turing machines and unrestricted grammars developed in Section 10.1 can be used to convert undecidability results from the domain of machines to the domain of grammars. Consider the problem of deciding whether a string w is generated by an unrestricted grammar G. A reduction that establishes the undecidability of the derivability problem has the form

Reduction	**Input**	**Condition**
Halting Problem to Derivability Problem	Turing machine M, string w $\downarrow$ unrestricted grammar G, string w	M halts with input w if, and only if, there is a derivation $S \overset{*}{\Rightarrow} w$ in G

Let M be a Turing machine and w an input string for M. The first step in the reduction is to modify M to obtain a machine M′ that accepts every string for which M halts. This is accomplished by making every state of M an accepting state in M′. In M′, halting and accepting are synonymous.

Using Theorem 10.1.3, we can construct a grammar $\mathrm{G}_{\mathrm{M}'}$ with $\mathrm{L}(\mathrm{G}_{\mathrm{M}'}) = \mathrm{L}(\mathrm{M}')$. An algorithm that decides whether $w \in \mathrm{L}(\mathrm{G}_{\mathrm{M}'})$ also determines whether the computation of M′ (and M) halts. Thus no such algorithm is possible.

12.4 Rice's Theorem

In the preceding sections we have shown that it is impossible to construct an algorithm to answer certain questions about a computation of an arbitrary Turing machine. The first example of this was the Halting Problem, which posed the question, "Will a Turing machine M halt when run with input w?" Problem reduction allowed us to establish that there is no algorithm that answers the question, "Will a Turing machine M halt when run with a blank tape?" In each of these problems, the input contained a Turing machine and the decision problem was concerned with determining the result of the computation of the machine.

Rather than asking about the computation of a Turing machine with a particular input string, we will now focus on determining whether the language accepted by a Turing machine satisfies a prescribed property. For example, we might be interested in the existence of an algorithm that, when given a Turing machine M as input, produces an answer to questions of the form

i) Is λ in L(M)?
ii) Is $\mathrm{L}(\mathrm{M}) = \emptyset$?
iii) Is L(M) a regular language?
iv) Is $\mathrm{L}(\mathrm{M}) = \Sigma^*$?

The ability to encode Turing machines as strings over {*0*, *1*} permits us to transform the preceding questions into questions about membership in a language. Employing the encoding, a set of Turing machines defines a language over {*0*, *1*} and the question of whether the set of strings accepted by a Turing machine M satisfies a property can be posed as a question of membership $R(\mathrm{M})$ in the appropriate language. For example, the question, "Is $\mathrm{L(M)} = \emptyset$?" can be rephrased in terms of membership as, "Is $R(\mathrm{M}) \in \mathrm{L}_{\emptyset}$?" Using this approach, the languages associated with the previous questions are

i) $\mathrm{L}_{\lambda} = \{R(\mathrm{M}) \mid \lambda \in \mathrm{L(M)}\}$
ii) $\mathrm{L}_{\emptyset} = \{R(\mathrm{M}) \mid \mathrm{L(M)} = \emptyset\}$
iii) $\mathrm{L}_{reg} = \{R(\mathrm{M}) \mid \mathrm{L(M)} \text{ is regular}\}$
iv) $\mathrm{L}_{\Sigma^*} = \{R(\mathrm{M}) \mid \mathrm{L(M)} = \Sigma^*\}$.

Example 12.3.1 showed that the question of membership in L_{Σ^*} is undecidable. That is, there is no algorithm that decides whether a Turing machine halts for all (and accepts) input strings.

The reduction strategy employed in Example 12.3.1 can be generalized to show that many languages consisting of representations of Turing machines are not recursive. A property $\mathbb{P}$ of recursively enumerable languages describes a condition that a recursively enumerable language may satisfy. For example, $\mathbb{P}$ may be "The language contains the null string"; "The language is the empty set"; "The language is regular"; or "The language contains all strings." The language of a property $\mathbb{P}$ is defined by $\mathrm{L}_{\mathbb{P}} = \{R(\mathrm{M}) \mid \mathrm{L(M)} \text{ satisfies } \mathbb{P}\}$. Thus $\mathrm{L}_{\emptyset}$, the language associated with the property "The language is the empty set" consists of the representations of all Turing machines that do not accept any strings.

A property $\mathbb{P}$ of recursively enumerable languages is called *trivial* if there are no recursively enumerable languages that satisfy $\mathbb{P}$ or if every recursively enumerable language satisfies $\mathbb{P}$. For a trivial property, $\mathrm{L}_{\mathbb{P}}$ is either the empty set or consists of all representations of Turing machines. Membership in both of these languages is decidable. Rice's Theorem shows that any property that is satisfied by some, but not all, recursively enumerable languages is undecidable.

Theorem 12.4.1 (Rice's Theorem)

If $\mathbb{P}$ is a nontrivial property of recursively enumerable languages, then $\mathrm{L}_{\mathbb{P}}$ is not recursive.

Proof. Let $\mathbb{P}$ be a nontrivial property that is not satisfied by the empty language. We will show that $\mathrm{L}_{\mathbb{P}} = \{R(\mathrm{M}) \mid \mathrm{L(M)} \text{ satisfies } \mathbb{P}\}$ is not recursive.

Since $\mathrm{L}_{\mathbb{P}}$ is nontrivial, there is at least one language $\mathrm{L} \in \mathrm{L}_{\mathbb{P}}$. Moreover, L is not $\emptyset$ by the assumption that the empty language does not satisfy $\mathbb{P}$. Let $\mathrm{M_L}$ be a Turing machine that accepts L.

The reducibility of the Halting Problem to $\mathrm{L}_{\mathbb{P}}$ will be used to show that $\mathrm{L}_{\mathbb{P}}$ is not recursive. As in Example 12.3.1, a preprocessor R will be designed to transform input $R(\mathrm{M})w$ into the encoding of a machine M′. The action of M′ when run with input y is to

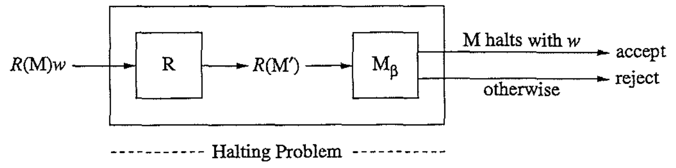
1. write w to the right of y, producing $ByBwB$;
2. run the transitions of M on w; and
3. if M halts when run with w, then run $\mathrm{M_L}$ with input y.

The role of the machine M and the string w is that of a gatekeeper. The processing of the input string y by $\mathrm{M_L}$ is allowed only if M halts with input w.

If the computation of M halts when run with w, then $\mathrm{M_L}$ is allowed to process input y. In this case the result of a computation of M′ with an input string y is exactly that of the computation of $\mathrm{M_L}$ with y. Consequently, $\mathrm{L(M')} = \mathrm{L(M_L)} = \mathrm{L}$ and $\mathrm{L(M')}$ satisfies $\mathbb{P}$. If the computation of M does not halt when run with w, then M′ never halts regardless of the input string y. Thus no string is accepted by M′ and $\mathrm{L(M')} = \emptyset$, which does not satisfy $\mathbb{P}$.

The machine M′ accepts $\emptyset$ when M does not halt with input w, and M′ accepts L when M halts with w. Since L satisfies $\mathbb{P}$ and $\emptyset$ does not, $\mathrm{L(M')}$ satisfies $\mathbb{P}$ if, and only if, M halts when run with input w.

Now assume that $\mathrm{L}_{\mathbb{P}}$ is recursive. Then there is a machine $\mathrm{M}_{\mathbb{P}}$ that decides membership in $\mathrm{L}_{\mathbb{P}}$. The machines R and $\mathrm{M}_{\mathbb{P}}$ combine to produce a solution to the Halting Problem.



Consequently, the property $\mathbb{P}$ is not decidable.

Originally, we assumed that $\mathbb{P}$ was not satisfied by the empty set. If $\emptyset \in \mathrm{L}_{\mathbb{P}}$, the preceding argument can be used to show that $\overline{\mathrm{L}_{\mathbb{P}}}$ is not recursive. It follows from Exercise 8.26 that $\mathrm{L}_{\mathbb{P}}$ must also be nonrecursive. ■

Rice's Theorem makes it easy to demonstrate the undecidability of many questions about properties of languages accepted by Turing machines, as is seen in the following example.

Example 12.4.1

The problem of determining whether the language accepted by a Turing machine is context-free is undecidable. By Rice's Theorem, all that is necessary is to show that the property "is context-free" is a nontrivial property of recursively enumerable languages. This is accomplished by finding one recursively enumerable language that is context-free and another that is not. The languages $\emptyset$ and $\{a^ib^ic^i \mid i \geq 0\}$ are both recursively enumerable; the former is context-free, and the latter is not (Example 7.4.1). □

12.5 An Unsolvable Word Problem

Semi-Thue Systems, named after their originator Norwegian mathematician Axel Thue, are a special type of grammar consisting of a single alphabet Σ and a set P of rules. A rule has the form $u \to v$, where $u \in \Sigma^+$ and $v \in \Sigma^*$. There is no division of the symbols into variables and terminals, nor is there a designated start symbol. The Word Problem for Semi-Thue

Consequently, an algorithm that determines whether two grammars generate the same language can be used to determine whether a Post correspondence system has a solution. ■

Exercises

1. Prove that the Halting Problem for the universal machine is undecidable. That is, there is no Turing machine that can determine whether the computation of U with an arbitrary input string will halt.
2. Explain the fundamental difference between the Halts on n'th Transition Problem from Example 11.5.2 and the Halting Problem that makes the former decidable and the latter undecidable.
3. Let M be any deterministic Turing machine that accepts a nonrecursive language. Prove that the Halting Problem for M is undecidable. That is, there is no Turing machine that takes input w and determines whether the computation of M halts with input w.

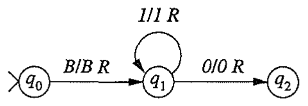
For Exercises 4 through 8, use reduction to establish the undecidability of the each of the decision problems.

4. Prove that there is no algorithm that determines whether an arbitrary Turing machine halts when run with the input string *101*.
5. Prove that there is no algorithm that determines whether an arbitrary Turing machine halts for at least one input string.
6. Prove that there is no algorithm with input consisting of a Turing machine M = (Q, Σ, Γ, δ, q_0, F), a state $q_i \in Q$, and a string $w \in \Sigma^*$ that determines whether the computation of M with input w enters state q_i.
7. Prove that there is no algorithm that determines whether an arbitrary Turing machine prints a *1* on its final transition.
8. Prove that there is no algorithm that determines whether an arbitrary Turing machine prints the symbol *1* on three consecutive transitions when run with a blank tape.
9. Why can't we successfully argue that the Blank Tape Problem is undecidable as follows: The Blank Tape Problem is a subproblem of the Halting Problem, which is undecidable and therefore must be undecidable itself.
10. Show that the problem of deciding whether a string over $\Sigma = \{1\}$ has even length is reducible to the Blank Tape Problem. Why is it incorrect to conclude from this that the problem of determining whether a string has even length is undecidable?
11. Give an example of a property of languages that is not satisfied by any recursively enumerable language.

12. Use Rice's Theorem to show that the following properties of recursively enumerable languages are undecidable. To establish the undecidability, all you need do is show that the property is nontrivial.
    a) L contains a particular string w.
    b) L is finite.
    c) L is regular.
    d) L is $\{0, 1\}^*$.
13. Let $L = \{R(M) \mid M \text{ halts when run with } R(M)\}$.
    a) Show that L is not recursive.
    b) Show that L is recursively enumerable.

* 14. Let $L_{\neq\emptyset} = \{R(M) \mid L(M) \text{ is nonempty}\}$.
    a) Show that $L_{\neq\emptyset}$ is not recursive.
    b) Show that $L_{\neq\emptyset}$ is recursively enumerable.

15. Let M be the Turing machine



    a) Give the rules of the Semi-Thue System S_M that simulate the computations of M.
    b) Trace the computation of M with input *01* and give the corresponding derivation in S_M.
16. Find a solution for each of the following Post correspondence systems.
    a) $[a, aa]$, $[bb, b]$, $[a, bb]$
    b) $[a, aaa]$, $[aab, b]$, $[abaa, ab]$
    c) $[aa, aab]$, $[bb, ba]$, $[abb, b]$
    d) $[a, ab]$, $[ba, aba]$, $[b, aba]$, $[bba, b]$
17. Show that the following Post correspondence systems have no solutions.
    a) $[b, ba]$, $[aa, b]$, $[bab, aa]$, $[ab, ba]$
    b) $[ab, a]$, $[ba, bab]$, $[b, aa]$, $[ba, ab]$
    c) $[ab, aba]$, $[baa, aa]$, $[aba, baa]$
    d) $[ab, bb]$, $[aa, ba]$, $[ab, abb]$, $[bb, bab]$
    e) $[abb, ab]$, $[aba, ba]$, $[aab, abab]$

* 18. Prove that the Post Correspondence Problem for systems with a one-symbol alphabet is decidable.

19. Let P be the Post correspondence system defined by $[b, bbb]$, $[babbb, ba]$, $[bab, aab]$, $[ba, a]$.
    a) Give a solution to P.
    b) Construct the grammars G_U and G_L from P.
    c) Give the derivations in G_U and G_L corresponding to the solution in (a).
20. Build the context-free grammars G_U and G_L that are constructed from the Post correspondence system $[b, bb]$, $[aa, baa]$, $[ab, a]$. Is $L(G_U) \cap L(G_L) = \emptyset$?
*21. Let C be a Post correspondence system. Construct a context-free grammar that generates $\overline{L(G_U)}$.
*22. Prove that there is no algorithm that determines whether the intersection of the languages of two context-free grammars contains infinitely many elements.
23. Prove that there is no algorithm that determines whether the complement of the language of a context-free grammar contains infinitely many elements.
*24. Prove that there is no algorithm that determines whether the languages of two arbitrary context-free grammars G_1 and G_2 satisfy $L(G_1) \subseteq L(G_2)$.

Bibliographic Notes

The undecidability of the Halting Problem was established by Turing [1936]. The proof given in Section 12.1 is from Minsky [1967]. Techniques for establishing undecidability using properties of languages were presented in Rice [1953] and [1956]. The string transformation systems of Thue were introduced in Thue [1914] and the undecidability of the Word Problem for Semi-Thue Systems was established by Post [1947].

The undecidability of the Post Correspondence Problem was presented in Post [1946]. The proof of Theorem 12.6.1, based on the technique of Floyd [1964], is from Davis and Weyuker [1983]. Undecidability results for context-free languages, including Theorem 12.7.1, can be found in Bar-Hillel, Perles, and Shamir [1961]. The undecidability of ambiguity of context-free languages was established by Cantor [1962], Floyd [1962], and Chomsky and Schutzenberger [1963]. The question of inherent ambiguity was shown to be unsolvable by Ginsburg and Ullian [1966a].

CHAPTER 13

Mu-Recursive Functions

In Chapter 9 we introduced computable functions from a mechanical perspective; the transitions of a Turing machine produced the values of a function. The Church-Turing Thesis asserts that every algorithmically computable function can be realized in this manner, but exactly what functions are Turing computable? In this chapter we will provide an answer to this question and, in doing so, obtain further support for the Church-Turing Thesis.

We now consider computable functions from a macroscopic viewpoint. Rather than focusing on elementary Turing machine operations, functions themselves are the fundamental objects of study. We introduce two families of functions, the primitive recursive functions and the μ-recursive functions. The primitive recursive functions are built from a set of intuitively computable functions using the operations of composition and primitive recursion. The μ-recursive functions are obtained by adding unbounded minimalization, a functional representation of sequential search, to the function building operations.

The computability of the primitive and μ-recursive functions is demonstrated by outlining an effective method for producing the values of the functions. The analysis of effective computation is completed by showing the equivalence of the notions of Turing computability and μ-recursivity. This answers the question posed in the opening paragraph—the functions computable by a Turing machine are exactly the μ-recursive functions.

13.1 Primitive Recursive Functions

A family of intuitively computable number-theoretic functions, known as the primitive recursive functions, is obtained from the basic functions

i) the successor function s: $s(x) = x + 1$
ii) the zero function z: $z(x) = 0$
iii) the projection functions $p_i^{(n)}$: $p_i^{(n)}(x_1, \ldots, x_n) = x_i$, $1 \le i \le n$

using operations that construct new functions from functions already in the family. The simplicity of the basic functions supports their intuitive computability. The successor function requires only the ability to add one to a natural number. Computing the zero function is even less complex; the value of the function is zero for every argument. The value of the projection function $p_i^{(n)}$ is simply its ith argument.

The primitive recursive functions are constructed from the basic functions by applications of two operations that preserve computability. The first operation is functional composition (Definition 9.4.2). Let f be defined by the composition of the n-variable function h with the k-variable functions $g_1, g_2, \ldots, g_n$. If each of the components of the composition is computable, then the value of $f(x_1, \ldots, x_k)$ can be obtained from h and $g_1(x_1, \ldots, x_k), g_2(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k)$. The computability of f follows from the computability of its constituent functions. The second operation for producing new functions is primitive recursion.

Definition 13.1.1

Let g and h be total number-theoretic functions with n and $n + 2$ variables, respectively. The $n + 1$-variable function f defined by

i) $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$
ii) $f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$

is said to be obtained from g and h by **primitive recursion.**

The x_i's are called the *parameters* of a definition by primitive recursion. The variable y is the *recursive variable.*

The operation of primitive recursion provides its own algorithm for computing the value of $f(x_1, \ldots, x_n, y)$ whenever g and h are computable. For a fixed set of parameters $x_1, \ldots, x_n$, $f(x_1, \ldots, x_n, 0)$ is obtained directly from the function g:

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n).$$

The value $f(x_1, \ldots, x_n, y + 1)$ is obtained from the computable function h using

i) the parameters $x_1, \ldots, x_n$,
ii) y, the previous value of the recursive variable, and
iii) $f(x_1, \ldots, x_n, y)$, the previous value of the function.

For example, $f(x_1, \ldots, x_n, y + 1)$ is obtained by the sequence of computations

$$\begin{aligned} f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, 1) &= h(x_1, \ldots, x_n, 0, f(x_1, \ldots, x_n, 0)) \\ f(x_1, \ldots, x_n, 2) &= h(x_1, \ldots, x_n, 1, f(x_1, \ldots, x_n, 1)) \\ &\vdots \\ f(x_1, \ldots, x_n, y + 1) &= h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)). \end{aligned}$$

Since h is computable, this iterative process can be used to determine $f(x_1, \ldots, x_n, y + 1)$ for any value of the recursive variable y.

Definition 13.1.2

A function is **primitive recursive** if it can be obtained from the successor, zero, and projection functions by a finite number of applications of composition and primitive recursion.

A function defined by composition or primitive recursion from total functions is itself total. This is an immediate consequence of the definitions of the operations and is left as an exercise. Since the basic primitive recursive functions are total and the operations preserve totality, it follows that all primitive recursive functions are total.

Taken together, composition and primitive recursion provide powerful tools for the construction of functions. The following examples show that arbitrary constant functions, addition, multiplication, and factorial are primitive recursive functions.

Example 13.1.1

The constant functions $c_i^{(n)}(x_1, \ldots, x_n) = i$ are primitive recursive. Example 9.4.2 defines the constant functions as the composition of the successor, zero, and projection functions. □

Example 13.1.2

Let add be the function defined by primitive recursion from the functions $g(x) = x$ and $h(x, y, z) = z + 1$. Then

$$\begin{aligned} add(x, 0) &= g(x) = x \\ add(x, y + 1) &= h(x, y, add(x, y)) = add(x, y) + 1. \end{aligned}$$

The function add computes the sum of two natural numbers. The definition of $add(x, 0)$ indicates that the sum of any number with zero is the number itself. The latter condition defines the sum of x and $y + 1$ as the sum of x and y (the result of add for the previous value of the recursive variable) incremented by one.

The preceding definition establishes that addition is primitive recursive. Both g and h, the components of the definition by primitive recursion, are primitive recursive since $g = p_1^{(1)}$ and $h = s \circ p_3^{(3)}$.

The result of the addition of two natural numbers can be obtained from the primitive recursive definition of *add* by repeatedly applying the condition $add(x, y+1) = add(x, y) + 1$ to reduce the value of the recursive variable. For example,

$$\begin{aligned} add(2, 4) &= add(2, 3) + 1 \\ &= (add(2, 2) + 1) + 1 \\ &= ((add(2, 1) + 1) + 1) + 1 \\ &= (((add(2, 0) + 1) + 1) + 1) + 1 \\ &= (((2 + 1) + 1) + 1) + 1 \\ &= 6. \end{aligned}$$

When the recursive variable is zero, the function g is used to initiate the evaluation of the expression. □

Example 13.1.3

Let g and h be the primitive functions $g = z$ and $h = add \circ (p_3^{(3)}, p_1^{(3)})$. Multiplication can be defined by primitive recursion from g and h as follows:

$$\begin{aligned} mult(x, 0) &= g(x) = 0 \\ mult(x, y+1) &= h(x, y, mult(x, y)) = mult(x, y) + x. \end{aligned}$$

The infix expression corresponding to the primitive recursive definition is the identity $x \cdot (y+1) = x \cdot y + x$, which follows from the distributive property of addition and multiplication. □

Adopting the convention that a zero-variable function is a constant, we can use Definition 13.1.1 to define one-variable functions using primitive recursion and a two-variable function h. The definition of such a function f has the form

i) $f(0) = n_0$, where $n_0 \in \mathrm{N}$
ii) $f(y+1) = h(y, f(y))$.

Example 13.1.4

The one-variable factorial function defined by

$$fact(y) = \begin{cases} 1 & \text{if } y = 0 \\ \prod_{i=1}^{y} i & \text{otherwise} \end{cases}$$

is primitive recursive. Let $h(x, y) = mult \circ (p_2^{(2)}, s \circ p_1^{(2)}) = y \cdot (x+1)$. The factorial function is defined using primitive recursion from h by

$$\begin{aligned} fact(0) &= 1 \\ fact(y+1) &= h(y, fact(y)) = fact(y) \cdot (y+1). \end{aligned}$$

Note that the definition uses $y + 1$, the value of the recursive variable. This is obtained by applying the successor function to y, the value provided to the function h.

The evaluation of the function *fact* for the first five input values illustrates how the primitive recursive definition generates the factorial function.

$$\begin{aligned} fact(0) &= 1 \\ fact(1) &= fact(0) \cdot (0+1) = 1 \\ fact(2) &= fact(1) \cdot (1+1) = 2 \\ fact(3) &= fact(2) \cdot (2+1) = 6 \\ fact(4) &= fact(3) \cdot (3+1) = 24 \end{aligned}$$

The factorial function is usually denoted $fact(x) = x!$. □

The primitive recursive functions were defined as a family of intuitively computable functions. The Church-Turing Thesis asserts that these functions must also be computable using our Turing machine approach to functional computation. The Theorem 13.1.3 shows that this is indeed the case.

Theorem 13.1.3

Every primitive recursive function is Turing computable.

Proof. Turing machines that compute the basic functions were constructed in Section 9.2. To complete the proof, it suffices to prove that the Turing computable functions are closed under composition and primitive recursion. The former was established in Section 9.4. All that remains is to show that the Turing computable functions are closed under primitive recursion; that is, if f is defined by primitive recursion from Turing computable functions g and h, then f is Turing computable.

Let g and h be Turing computable functions and let f be the function

$$\begin{aligned} f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, y+1) &= h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) \end{aligned}$$

defined from g and h by primitive recursion. Since g and h are Turing computable, there are standard Turing machines G and H that compute them. A composite machine F is constructed to compute f. The computation of $f(x_1, x_2, \ldots, x_n, y)$ begins with tape configuration $B\overline{x}_1B\overline{x}_2B \ldots B\overline{x}_nB\overline{y}B$.

1. A counter, initially set to 0, is written to the immediate right of the input. The counter is used to record the value of the recursive variable for the current computation.

The parameters are then written to the right of the counter, producing the tape configuration

$$B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB\overline{y}B\overline{0}B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB.$$

2. The machine G is run on the final n values of the tape, producing

$$B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB\overline{y}B\overline{0}B\overline{g(x_1,x_2,\ldots,x_n)}B.$$

   The computation of G generates $g(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_n, 0)$.
3. The tape now has the form

$$B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB\overline{y}B\overline{i}B\overline{f(x_1,x_2,\ldots,x_n,i)}B.$$

   If the counter i is equal to y, the computation of $f(x_1, x_2, \ldots, x_n, y)$ is completed by erasing the initial $n+2$ numbers on the tape and translating the result to tape position one.
4. If $i < y$, the tape is configured to compute the next value of f.

$$B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB\overline{y}B\overline{i+1}B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB\overline{i}B\overline{f(x_1,x_2,\ldots,x_n,i)}B$$

   The machine H is run on the final $n+2$ values on the tape, producing

$$B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB\overline{y}B\overline{i+1}B\overline{h(x_1,x_2,\ldots,x_n,i,f(x_1,x_2,\ldots,x_n,i))}B,$$

   where the rightmost value on the tape is $f(x_1, x_2, \ldots, x_n, i+1)$. The computation continues with the comparison in step 3. ■

13.2 Some Primitive Recursive Functions

A function is primitive recursive if it can be constructed from the zero, successor, and projection functions by a finite number of applications of composition and primitive recursion. Composition permits g and h, the functions used in a primitive recursive definition, to utilize any function that has previously been shown to be primitive recursive.

Primitive recursive definitions are constructed for several common arithmetic functions. Rather than explicitly detailing the functions g and h, a definition by primitive recursion is given in terms of the parameters, the recursive variable, the previous value of the function, and other primitive recursive functions. Note that the definitions of addition and multiplication are identical to the formal definitions given in Examples 13.1.2 and 13.1.3, with the intermediate step omitted.

Because of the compatibility with the operations of composition and primitive recursion, the definitions in Tables 13.1 and 13.2 are given using the functional notation. The standard infix representations of the binary arithmetic functions, given below the function names, are used in the arithmetic expressions throughout the chapter. The notation "+ 1" denotes the successor operator.

TABLE 13.1 Primitive Recursive Arithmetic Functions

Description	Function	Definition
Addition	$add(x, y)$	$add(x, 0) = x$
	$x + y$	$add(x, y+1) = add(x, y) + 1$
Multiplication	$mult(x, y)$	$mult(x, 0) = 0$
	$x \cdot y$	$mult(x, y+1) = mult(x, y) + x$
Predecessor	$pred(y)$	$pred(0) = 0$
		$pred(y+1) = y$
Proper subtraction	$sub(x, y)$	$sub(x, 0) = x$
	$x \mathbin{\dot{-}} y$	$sub(x, y+1) = pred(sub(x, y))$
Exponentation	$exp(x, y)$	$exp(x, 0) = 1$
	x^y	$exp(x, y+1) = exp(x, y) \cdot x$

A primitive recursive predicate is a primitive recursive function whose range is the set $\{0, 1\}$. Zero and one are interpreted as false and true, respectively. The first two predicates in Table 13.2, the sign predicates, specify the sign of the argument. The function sg is true when the argument is positive. The complement of sg, denoted $cosg$, is true when the input is zero. Binary predicates that compare the input can be constructed from the arithmetic functions and the sign predicates using composition.

TABLE 13.2 Primitive Recursive Predicates

Description	Predicate	Definition
Sign	$sg(x)$	$sg(0) = 0$
		$sg(y+1) = 1$
Sign complement	$cosg(x)$	$cosg(0) = 1$
		$cosg(y+1) = 0$
Less than	$lt(x, y)$	$sg(y \mathbin{\dot{-}} x)$
Greater than	$gt(x, y)$	$sg(x \mathbin{\dot{-}} y)$
Equal to	$eq(x, y)$	$cosg(lt(x, y) + gt(x, y))$
Not equal to	$ne(x, y)$	$cosg(eq(x, y))$

Predicates are functions that exhibit the truth or falsity of a proposition. The logical operations negation, conjunction, and disjunction can be constructed using the arithmetic functions and the sign predicates. Let p_1 and p_2 be two primitive recursive predicates. Logical operations on p_1 and p_2 can be defined as follows:

Predicate	Interpretation
$cosg(p_1)$	not p_1
$p_1 \cdot p_2$	p_1 and p_2
$sg(p_1 + p_2)$	p_1 or p_2

Applying *cosg* to the result of a predicate interchanges the values, yielding the negation of the predicate. This technique was used to define the predicate *ne* from the predicate *eq*. Determining the value of a disjunction begins by adding the truth values of the component predicates. Since the sum is 2 when both of the predicates are true, the disjunction is obtained by composing the addition with *sg*. The resulting predicates are primitive recursive since the components of the composition are primitive recursive.

Example 13.2.1

The equality predicates can be used to explicitly specify the value of a function for a finite set of arguments. For example, f is the identity function for all input values other than 0, 1, and 2:

$$f(x) = \begin{cases} 2 & \text{if } x = 0 \\ 5 & \text{if } x = 1 \\ 4 & \text{if } x = 2 \\ x & \text{otherwise} \end{cases} \qquad \begin{aligned} f(x) = {} & eq(x, 0) \cdot 2 \\ & + eq(x, 1) \cdot 5 \\ & + eq(x, 2) \cdot 4 \\ & + gt(x, 2) \cdot x. \end{aligned}$$

The function f is primitive recursive since it can be written as the composition of primitive recursive functions eq, gt, $\cdot$, and $+$. The four predicates in f are exhaustive and mutually exclusive; that is, one and only one of them is true for any natural number. The value of f is determined by the single predicate that holds for the input. □

The technique presented in the previous example, constructing a function from exhaustive and mutually exclusive primitive recursive predicates, is used to establish the following theorem.

Theorem 13.2.1

Let g be a primitive recursive function and f a total function that is identical to g for all but a finite number of input values. Then f is primitive recursive.

Proof. Let g be primitive recursive and let f be defined by

$$f(x) = \begin{cases} y_1 & \text{if } x = n_1 \\ y_2 & \text{if } x = n_2 \\ \vdots & \\ y_k & \text{if } x = n_k \\ g(x) & \text{otherwise.} \end{cases}$$

The equality predicate is used to specify the values of f for input $n_1, \ldots, n_k$. For all other input values, $f(x) = g(x)$. The predicate obtained by the product

$$ne(x, n_1) \cdot ne(x, n_2) \cdot \cdots \cdot ne(x, n_k)$$

is true whenever the value of f is determined by g. Using these predicates, f can be written

$$\begin{aligned} f(x) = {} & eq(x, n_1) \cdot y_1 + eq(x, n_2) \cdot y_2 + \cdots + eq(x, n_k) \cdot y_k \\ & + ne(x, n_1) \cdot ne(x, n_2) \cdot \cdots \cdot ne(x, n_k) \cdot g(x). \end{aligned}$$

Thus f is also primitive recursive. ■

The order of the variables is an essential feature of a definition by primitive recursion. The initial variables are the parameters and the final variable is the recursive variable. Combining composition and the projection functions permits a great deal of flexibility in specifying the number and order of variables in a primitive recursive function. This flexibility is demonstrated by considering alterations to the variables in a two-variable function.

Theorem 13.2.2

Let $g(x, y)$ be a primitive recursive function. Then the functions obtained by

i) (adding dummy variables) $f(x, y, z_1, z_2, \ldots, z_n) = g(x, y)$

ii) (permuting variables) $f(x, y) = g(y, x)$

iii) (identifying variables) $f(x) = g(x, x)$

are primitive recursive.

Proof. Each of the functions is primitive recursive since it can be obtained from g and the projections by composition as follows:

i) $f = g \circ (p_1^{(n+2)}, p_2^{(n+2)})$

ii) $f = g \circ (p_2^{(2)}, p_1^{(2)})$

iii) $f = g \circ (p_1^{(1)}, p_1^{(1)})$. ■

Dummy variables are used to make functions with different numbers of variables compatible for composition. The definition of the composition $h \circ (g_1, g_2)$ requires that g_1

and g_2 have the same number of variables. Consider the two-variable function f defined by $f(x, y) = (x \cdot y) + x!$. The constituents of the addition are obtained from a multiplication and a factorial operation. The former function has two variables and the latter has one. Adding a dummy variable to the function *fact* produces a two-variable function *fact'* satisfying $fact'(x, y) = fact(x) = x!$. Finally, we note that $f = add \circ (mult, fact')$ so that f is also primitive recursive.

13.3 Bounded Operators

The sum of a sequence of natural numbers can be obtained by repeated applications of the binary operation of addition. Addition and projection can be combined to construct a function that adds a fixed number of arguments. For example, the primitive recursive function

$$add \circ (p_1^{(4)}, add \circ (p_2^{(4)}, add \circ (p_3^{(4)}, p_4^{(4)})))$$

returns the sum of its four arguments. This approach cannot be used when the number of summands is variable. Consider the function

$$f(y) = \sum_{i=0}^{y} g(i) = g(0) + g(1) + \cdots + g(y).$$

The number of additions is determined by the input variable y. The function f is called the *bounded sum* of g. The variable i is the index of the summation. Computing a bounded sum consists of three actions: the generation of the summands, binary addition, and the comparison of the index with the input y.

We will prove that the bounded sum of a primitive recursive function is primitive recursive. The technique presented can be used to show that repeated applications of any binary primitive recursive operation is also primitive recursive.

Theorem 13.3.1

Let $g(x_1, \ldots, x_n, y)$ be a primitive recursive function. Then the functions

i) (bounded sum) $f(x_1, \ldots, x_n, y) = \sum_{i=0}^{y} g(x_1, \ldots, x_n, i)$

ii) (bounded product) $f(x_1, \ldots, x_n, y) = \prod_{i=0}^{y} g(x_1, \ldots, x_n, i)$

are primitive recursive.

Proof. The sum

$$\sum_{i=0}^{y} g(x_1, \ldots, x_n, i)$$

is obtained by adding $g(x_1, \ldots, x_n, y)$ to

$$\sum_{i=0}^{y-1} g(x_1, \ldots, x_n, i).$$

Translating this into the language of primitive recursion, we get

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n, 0)$$
$$f(x_1, \ldots, x_n, y+1) = f(x_1, \ldots, x_n, y) + g(x_1, \ldots, x_n, y+1). \qquad \blacksquare$$

The bounded operations just introduced begin with index zero and terminate when the index reaches the value specified by the argument y. Bounded operations can be generalized by having the range of the index variable determined by two computable functions. The functions l and u are used to determine the lower and upper bounds of the index.

Theorem 13.3.2

Let g be an $n+1$-variable primitive recursive function and let l and u be n-variable primitive recursive functions. Then the functions

i) $f(x_1, \ldots, x_n) = \sum_{i=l(x_1,\ldots,x_n)}^{u(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i)$

ii) $f(x_1, \ldots, x_n) = \prod_{i=l(x_1,\ldots,x_n)}^{u(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i)$

are primitive recursive.

Proof. Since the lower and upper bounds of the summation are determined by the functions l and u, it is possible that the lower bound may be greater than the upper bound. When this occurs, the result of the summation is assigned the default value zero. The predicate

$$gt(l(x_1, \ldots, x_n), u(x_1, \ldots, x_n))$$

is true in precisely these instances.

If the lower bound is less than or equal to the upper bound, the summation begins with index $l(x_1, \ldots, x_n)$ and terminates when the index reaches $u(x_1, \ldots, x_n)$. Let g' be the primitive recursive function defined by

$$g'(x_1, \ldots, x_n, y) = g(x_1, \ldots, x_n, y + l(x_1, \ldots, x_n)).$$

The values of g' are obtained from those of g and $l(x_1, \ldots, x_n)$:

$$g'(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n, l(x_1, \ldots, x_n))$$
$$g'(x_1, \ldots, x_n, 1) = g(x_1, \ldots, x_n, 1 + l(x_1, \ldots, x_n))$$
$$\vdots$$
$$g'(x_1, \ldots, x_n, y) = g(x_1, \ldots, x_n, y + l(x_1, \ldots, x_n)).$$

By Theorem 13.3.1, the function

$$\begin{aligned} f'(x_1, \ldots, x_n, y) &= \sum_{i=0}^{y} g'(x_1, \ldots, x_n, i) \\ &= \sum_{i=l(x_1,\ldots,x_n)}^{y+l(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i) \end{aligned}$$

is primitive recursive. The generalized bounded sum can be obtained by composing f' with the functions u and l:

$$f'(x_1, \ldots, x_n, (u(x_1, \ldots, x_n) \dot{-} l(x_1, \ldots, x_n))) = \sum_{i=l(x_1,\ldots,x_n)}^{u(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i).$$

Multiplying this function by the predicate that compares the upper and lower bounds ensures that the bounded sum returns the default value whenever the lower bound exceeds the upper bound. Thus

$$\begin{aligned} f(x_1, \ldots, x_n) = cosg(&gt(l(x_1, \ldots, x_n), u(x_1, \ldots, x_n))) \\ &\cdot f'(x_1, \ldots, x_n, (u(x_1, \ldots, x_n) \dot{-} l(x_1, \ldots, x_n))). \end{aligned}$$

Since each of the constituent functions is primitive recursive, it follows that f is also primitive recursive.

A similar argument can be used to show that the generalized bounded product is primitive recursive. When the lower bound is greater than the upper, the bounded product defaults to one. ■

The value returned by a predicate p designates whether the input satisfies the property represented by p. For fixed values $x_1, \ldots, x_n$,

$$\mu z[p(x_1, \ldots, x_n, z)]$$

is defined to be the smallest natural number z such that $p(x_1, \ldots, x_n, z) = 1$. The notation $\mu z[p(x_1, \ldots, x_n, z)]$ is read "the least z satisfying $p(x_1, \ldots, x_n, z)$." This construction is called the *minimalization* of p, and μz is called the μ-operator. The minimalization of an $n + 1$-variable predicate defines an n-variable function

$$f(x_1, \ldots, x_n) = \mu z[p(x_1, \ldots, x_n, z)].$$

An intuitive interpretation of minimalization is that it performs a search over the natural numbers. Initially, the variable z is set to zero. The search sequentially examines the natural numbers until a value of z for which $p(x_1, \ldots, x_n, z) = 1$ is encountered.

Unfortunately, the function obtained by the minimalization of a primitive recursive predicate need not be primitive recursive. In fact, such a function may not even be total. Consider the function

$$f(x) = \mu z[eq(x, z \cdot z)].$$

Using the characterization of minimalization as search, f searches for the first z such that $z^2 = x$. If x is a perfect square, then $f(x)$ returns the square root of x. Otherwise, f is undefined.

By restricting the range over which the minimalization occurs, we obtain a bounded minimalization operator. An $n + 1$-variable predicate defines an $n + 1$-variable function

$$\begin{aligned} f(x_1, \ldots, x_n, y) &= \overset{y}{\mu} z[p(x_1, \ldots, x_n, z)] \\ &= \begin{cases} z & \text{if } p(x_1, \ldots, x_n, i) = 0 \text{ for } 0 \le i < z \le y \\ & \quad \text{and } p(x_1, \ldots, x_n, z) = 1 \\ y + 1 & \text{otherwise.} \end{cases} \end{aligned}$$

The bounded μ-operator returns the first natural number z less than or equal to y for which $p(x_1, \ldots, x_n, z) = 1$. If no such value exists, the default value of $y + 1$ is assigned. Limiting the search to the range of natural numbers between zero and y ensures the totality of the function

$$f(x_1, \ldots, x_n, y) = \overset{y}{\mu} z[p(x_1, \ldots, x_n, z)].$$

In fact, the bounded minimalization operator defines a primitive recursive function whenever the predicate is primitive recursive.

Theorem 13.3.3

Let $p(x_1, \ldots, x_n, y)$ be a primitive recursive predicate. Then the function

$$f(x_1, \ldots, x_n, y) = \overset{y}{\mu} z[p(x_1, \ldots, x_n, z)]$$

is primitive recursive.

Proof. The proof is given for a two-variable predicate $p(x, y)$ and easily generalizes to n-variable predicates. We begin by defining an auxiliary predicate

$$\begin{aligned} g(x, y) &= \begin{cases} 1 & \text{if } p(x, i) = 0 \text{ for } 0 \le i \le y \\ 0 & \text{otherwise} \end{cases} \\ &= \prod_{i=0}^{y} cosg(p(x, i)). \end{aligned}$$

This predicate is primitive recursive since it is a bounded product of the primitive recursive predicate $cosg \circ p$.

The bounded sum of the predicate g produces the bounded μ-operator. To illustrate the use of g in constructing the minimalization operator, consider a two-variable predicate p with argument n whose values are given in the left column:

$$\begin{array}{lll}
p(n,0)=0 & g(n,0)=1 & \sum_{i=0}^{0} g(n,i)=1 \\
p(n,1)=0 & g(n,1)=1 & \sum_{i=0}^{1} g(n,i)=2 \\
p(n,2)=0 & g(n,2)=1 & \sum_{i=0}^{2} g(n,i)=3 \\
p(n,3)=1 & g(n,3)=0 & \sum_{i=0}^{3} g(n,i)=3 \\
p(n,4)=0 & g(n,4)=0 & \sum_{i=0}^{4} g(n,i)=3 \\
p(n,5)=1 & g(n,5)=0 & \sum_{i=0}^{5} g(n,i)=3 \\
\vdots & \vdots & \vdots
\end{array}$$

The value of g is one until the first number z with $p(n, z) = 1$ is encountered. All subsequent values of g are zero. The bounded sum adds the results generated by g. Thus

$$\sum_{i=0}^{y} g(n,i) = \begin{cases} y+1 & \text{if } z > y \\ z & \text{otherwise.} \end{cases}$$

The first condition also includes the possibility that there is no z satisfying $p(n, z) = 1$. In this case the default value is returned regardless of the specified range.

By the preceding argument, we see that the bounded minimalization of a primitive recursive predicate p is given by the function

$$f(x,y) = \overset{y}{\mu} z[p(x,z)] = \sum_{i=0}^{y} g(x,i),$$

and consequently is primitive recursive. ■

Bounded minimalization $f(y) = \overset{y}{\mu} z[p(x, z)]$ can be thought of as a search for the first value of z in the range 0 to y that makes p true. Example 13.3.1 shows that minimalization can also be used to find first value in a subrange or the largest value z in a specified range that satisfies p.

Example 13.3.1

Let $p(x, z)$ be a primitive recursive predicate. Then the functions

i) $f_1(x, y_0, y)$ = the first value in the range $[y_0, y]$ for which $p(x, z)$ is true,
ii) $f_2(x, y)$ = the second value in the range $[0, y]$ for which $p(x, z)$ is true, and
iii) $f_3(x, y)$ = the largest value in the range $[0, y]$ for which $p(x, z)$ is true

are also primitive recursive. For each of these functions, the default is $y + 1$ if there is no value of z that satisfies the specified condition.

To show that f_1 is primitive recursive, the primitive recursive function ge, greater than or equal to, is used to enforce a lower bound on the value of the function. The predicate $p(x, z) \cdot ge(z, y_0)$ is true whenever $p(x, z)$ is true and z is greater than or equal to y_0. The bounded minimalization

$$f_1(x, y_0, y) = \overset{y}{\mu} z[p(x,z) \cdot ge(z, y_0)],$$

returns the first value in the range $[y_0, y]$ for which $p(x, z)$ is true.

The minimalization $\overset{y}{\mu} z'[p(x, z')]$ is the first value in $[0, y]$ for which $p(x, z)$ is true. The second value that makes $p(x, z)$ true is the first value greater than $\overset{y}{\mu} z'[p(x, z')]$ that satisfies p. Using the preceding technique, the function

$$f_2(x,y) = \overset{y}{\mu} z\big[p(x,z) \cdot gt(z, \overset{y}{\mu} z'[p(x,z')])\big]$$

returns the second value in the range $[0, y]$ for which p is true.

A search for the largest value in the range $[0, y]$ must sequentially examine $y, y - 1, y - 2, \ldots, 1, 0$. The bounded minimalization $\overset{y}{\mu} z[p(x, y \dot{-} z)]$ examines the values in the desired order; when $z = 0$, $p(x, y)$ is tested, when $z = 1$, $p(x, y - 1)$ is tested, and so on. The function $f'(x, y) = y \dot{-} \overset{y}{\mu} z[p(x, y \dot{-} z)]$ returns the largest value less than or equal to y that satisfies p. However, the result of f' is $y \dot{-} (y + 1) = 0$ when no such value exists. A comparison is used to produce the proper default value. The first condition in the function

$$f_3(x,y) = eq(y+1, \overset{y}{\mu} z[p(x,z)]) \cdot (y+1) + neq(y+1, \overset{y}{\mu} z[p(x,z)]) \cdot f'(x,y))$$

returns the default $y + 1$ if there is no value in $[0, y]$ that satisfies p. Otherwise, the largest such value is returned. □

Bounded minimalization can be generalized by computing the upper bound of the search with a function u. If u is primitive recursive, so is the resulting function. The proof is similar to that of Theorem 13.3.2 and is left as an exercise.

Theorem 13.3.4

Let p be an $n+1$-variable primitive recursive predicate and let u be an n-variable primitive recursive function. Then the function

$$f(x_1, \ldots, x_n) = \overset{u(x_1,\ldots,x_n)}{\mu z}[p(x_1, \ldots, x_n, z)]$$

is primitive recursive.

13.4 Division Functions

The fundamental operation of integer division, *div*, is not total. The function $div(x, y)$ returns the quotient, the integer part of the division of x by y, when the second argument is nonzero. The function is undefined when y is zero. Since all primitive recursive functions are total, it follows that *div* is not primitive recursive. A primitive recursive division function quo is defined by assigning a default value when the denominator is zero:

$$quo(x, y) = \begin{cases} 0 & \text{if } y = 0 \\ div(x, y) & \text{otherwise.} \end{cases}$$

The division function quo is constructed using the primitive recursive operation of multiplication. For values of y other than zero, $quo(x, y) = z$ implies that z satisfies $z \cdot y \leq x < (z+1) \cdot y$. That is, $quo(x, y)$ is the smallest natural number z such that $(z+1) \cdot y$ is greater than x. The search for the value of z that satisfies the inequality succeeds before z reaches x since $(x+1) \cdot y$ is greater than x. The function

$$\overset{x}{\mu z}[gt((z+1) \cdot y,\ x)]$$

determines the quotient of x and y whenever the division is defined. The default value is obtained by multiplying the minimalization by $sg(y)$. Thus

$$quo(x, y) = sg(y) \cdot \overset{x}{\mu z}[gt((z+1) \cdot y,\ x)],$$

where the bound is determined by the primitive recursive function $p_1^{(2)}$. The previous definition demonstrates that quo is primitive recursive since it has the form prescribed by Theorem 13.3.4.

The quotient function can be used to define a number of division-related functions and predicates including those given in Table 13.3. The function *rem* returns the remainder of the division of x by y whenever the division is defined. Otherwise, $rem(x, 0) = x$. The predicate *divides* defined by

$$divides(x, y) = \begin{cases} 1 & \text{if } x > 0,\ y > 0, \text{ and } y \text{ is a divisor of } x \\ 0 & \text{otherwise} \end{cases}$$

is true whenever y divides x. By convention, zero is not considered to be divisible by any number. The multiplication by $sg(x)$ in the definition of *divides* in Table 13.3 enforces this condition. The default value of the remainder function guarantees that $divides(x, 0) = 0$.

TABLE 13.3 Primitive Recursive Division Functions

Description	Function	Definition
Quotient	$quo(x, y)$	$sg(y) \cdot \overset{x}{\mu z}[gt((z+1) \cdot y,\ x)]$
Remainder	$rem(x, y)$	$x \mathbin{\dot-} (y \cdot quo(x, y))$
Divides	$divides(x, y)$	$eq(rem(x, y), 0) \cdot sg(x)$
Number of divisors	$ndivisors(x, y)$	$\sum_{i=0}^{x} divides(x, i)$
Prime	$prime(x)$	$eq(ndivisors(x), 2)$

The generalized bounded sum can be used to count the number of divisors of a number. The upper bound of the sum is obtained from the input by the primitive recursive function $p_1^{(1)}$. This bound is satisfactory since no number greater than x is a divisor of x. A prime number is a number whose only divisors are 1 and itself. The predicate *prime* simply checks if the number of divisors is two.

The predicate prime and bounded minimalization can be used to construct a primitive recursive function pn that enumerates the primes. The value of $pn(i)$ is the ith prime. Thus, $pn(0) = 2$, $pn(1) = 3$, $pn(2) = 5$, $pn(3) = 7, \ldots$. The $x+1$st prime is the first prime number greater than $pn(x)$. Bounded minimalization is ideally suited for performing this type of search. To employ the bounded μ-operator, we must determine an upper bound for the minimalization. By Theorem 13.3.4, the bound may be calculated using the input value x.

Lemma 13.4.1

Let $pn(x)$ denote the xth prime. Then $pn(x+1) \leq pn(x)! + 1$.

Proof. Each of the primes $pn(i)$, $i = 0, 1, \ldots, x$, divides $pn(x)!$. Since a prime cannot divide two consecutive numbers, either $pn(x)! + 1$ is prime or its prime decomposition contains a prime other than $pn(0), pn(1), \ldots, pn(x)$. In either case, $pn(x+1) \leq pn(x)! + 1$. ■

The bound provided by the preceding lemma is computed by the primitive recursive function $fact(x) + 1$. The xth prime function is obtained by primitive recursion as follows:

$$pn(0) = 2$$

$$pn(x+1) = \overset{fact(pn(x))+1}{\mu z}[prime(z) \cdot gt(z, pn(x))].$$

Let us take a moment to reflect on the consequences of the relationship between the family of primitive recursive functions and Turing computability. By Theorem 13.1.3, every primitive recursive function is Turing computable. Designing Turing machines that explicitly compute functions such as pn or $ndivisors$ would require a large number of states and a complicated transition function. Using the macroscopic approach to computation, these functions are easily shown to be computable. Without the tedium inherent in constructing complicated Turing machines, we have shown that many useful functions and predicates are Turing computable.

13.5 Gödel Numbering and Course-of-Values Recursion

Many common computations involving natural numbers are not number-theoretic functions. Sorting a sequence of numbers returns a sequence, not a single number. However, there are many sorting algorithms that we consider effective procedures. We now introduce primitive recursive constructions that allow us to perform this type of operation. The essential feature is the ability to encode a sequence of numbers in a single value. The coding scheme utilizes the unique decomposition of a natural number into a product of primes. Such codes are called *Gödel numberings* after German logician Kurt Gödel, who developed the technique.

A sequence $x_0, x_1, \ldots, x_{n-1}$ of n natural numbers is encoded by

$$pn(0)^{x_0+1} \cdot pn(1)^{x_1+1} \cdot \cdots \cdot pn(n)^{x_n+1} = 2^{x_0+1} \cdot 3^{x_1+1} \cdot \cdots \cdot pn(n)^{x_n+1}.$$

Since our numbering begins with zero, the elements of a sequence of length n are numbered $0, 1, \ldots, n-1$. Examples of the Gödel numbering of several sequences are

Sequence	Encoding
1, 2	$2^2 3^3 = 108$
0, 1, 3	$2^1 3^2 5^4 = 11{,}250$
0, 1, 0, 1	$2^1 3^2 5^1 7^2 = 4{,}410$

An encoded sequence of length n is a product of powers of the first n primes. The choice of the exponent $x_i + 1$ guarantees that $pn(i)$ occurs in the encoding even when x_i is zero.

The definition of a function that encodes a fixed number of inputs can be obtained directly from the definition of the Gödel numbering. We let

$$gn_n(x_0, \ldots, x_n) = pn(0)^{x_0+1} \cdot \cdots \cdot pn(n)^{x_n+1} = \prod_{i=0}^{n} pn(i)^{x_i+1}$$

be the $n+1$-variable function that encodes a sequence $x_0, x_1, \ldots, x_n$. The function gn_{n-1} can be used to encode the components of an ordered n-tuple. The Gödel number associated with the ordered pair $[x_0, x_1]$ is $gn_1(x_0, x_1)$.

A decoding function is constructed to retrieve the components of an encoded sequence. The function

$$dec(i, x) = \overset{x}{\mu} z[cosg(divides(x, pn(i)^{z+1}))] \dot{-} 1$$

returns the ith element of the sequence encoded in the Gödel number x. The bounded μ-operator is used to find the power of $pn(i)$ in the prime decomposition of x. The minimalization returns the first value of z for which $pn(i)^{z+1}$ does not divide x. The ith element in an encoded sequence is one less than the power of $pn(i)$ in the encoding. The decoding function $dec(x, i)$ returns zero for every prime $pn(i)$ that does not occur in the prime decomposition of x.

When a computation requires n previously computed values, the Gödel encoding function gn_{n-1} can be used to encode the values. The encoded values can be retrieved when they are needed by the computation.

Example 13.5.1

The Fibonacci numbers are defined as the sequence 0, 1, 1, 2, 3, 5, 8, 13, ..., where an element in the sequence is the sum of its two predecessors. The function

$$f(0) = 0$$
$$f(1) = 1$$
$$f(y+1) = f(y) + f(y-1) \text{ for } y > 1$$

generates the Fibonacci numbers. This is not a definition by primitive recursion since the computation of $f(y+1)$ utilizes both $f(y)$ and $f(y-1)$. To show that the Fibonacci numbers are generated by a primitive recursive function, the Gödel numbering function gn_1 is used to store the two values as a single number. An auxiliary function h encodes the ordered pair with first component $f(y-1)$ and second component $f(y)$:

$$h(0) = gn_1(0, 1) = 2^1 3^2 = 18$$
$$h(y+1) = gn_1(dec(1, h(y)),\ dec(0, h(y)) + dec(1, h(y))).$$

The initial value of h is the encoded pair $[f(0), f(1)]$. The calculation of $h(y+1)$ begins by producing the components of the subsequent ordered pair

$$[dec(1, h(y)),\ dec(0, h(y)) + dec(1, h(y))] = [f(y),\ f(y-1) + f(y)].$$

Encoding the pair with gn_1 completes the evaluation of $h(y+1)$. This process constructs the sequence of Gödel numbers of the pairs $[f(0), f(1)], [f(1), f(2)], [f(2), f(3)], \ldots$. The primitive recursive function $f(y) = dec(0, h(y))$ extracts the Fibonacci numbers from the first components of the ordered pairs. □

The Gödel numbering functions gn_i encode a fixed number of arguments. A Gödel numbering function can be constructed in which the number of elements to be encoded is computed from the arguments of the function. The approach is similar to that taken in constructing the bounded sum and product operations. The values of a one-variable primitive recursive function f with input $0, 1, \ldots, n$ define a sequence $f(0), f(1), \ldots, f(n)$ of length $n+1$. Using the bounded product, the Gödel numbering function

$$gn_f(x_1, \ldots, x_n, y) = \prod_{i=0}^{y} pn(i)^{f(i)+1}$$

encodes the first $y+1$ values of f. The relationship between a function f and its encoding function gn_f is established in Theorem 13.5.1.

Theorem 13.5.1

Let f be an $n+1$-variable function and gn_f the encoding function defined from f. Then f is primitive recursive if, and only if, gn_f is primitive recursive.

Proof. If $f(x_1, \ldots, x_n, y)$ is primitive recursive, then the bounded product

$$gn_f(x_1, \ldots, x_n, y) = \prod_{i=0}^{y} pn(i)^{f(x_1,\ldots,x_n,i)+1}$$

computes the Gödel encoding function. On the other hand, the decoding function can be used to recover the values of f from the Gödel number generated by gn_f:

$$f(x_1, \ldots, x_n, y) = dec(y, gn_f(x_1, \ldots, x_n, y)).$$

Thus f is primitive recursive whenever gn_f is. ■

The primitive recursive functions have been introduced because of their intuitive computability. In a definition by primitive recursion, the computation is permitted to use the result of the function with the previous value of the recursive variable. Consider the function defined by

$$\begin{aligned}
f(0) &= 1\\
f(1) &= f(0)\cdot 1 = 1\\
f(2) &= f(0)\cdot 2 + f(1)\cdot 1 = 3\\
f(3) &= f(0)\cdot 3 + f(1)\cdot 2 + f(2)\cdot 1 = 8\\
f(4) &= f(0)\cdot 4 + f(1)\cdot 3 + f(2)\cdot 2 + f(3)\cdot 1 = 21\\
&\ \vdots
\end{aligned}$$

The function f can be written as

$$\begin{aligned}
f(0) &= 1\\
f(y+1) &= \sum_{i=0}^{y} f(i)\cdot(y+1-i).
\end{aligned}$$

The definition, as formulated, is not primitive recursive since the computation of $f(y+1)$ utilizes all of the previously computed values. The function, however, is intuitively computable; the definition itself outlines an algorithm by which any value can be calculated.

When the result of a function with recursive variable $y+1$ is defined in terms of $f(0), f(1), \ldots, f(y)$, the function f is said to be defined by course-of-values recursion. Determining the result of a function defined by course-of-values recursion appears to utilize a different number of inputs for each value of the recursive variable. In the preceding example, $f(2)$ requires only $f(0)$ and $f(1)$, while $f(4)$ requires $f(0), f(1), f(2)$, and $f(3)$. No single function can be used to compute both $f(2)$ and $f(4)$ directly from the preceding values since a function is required to have a fixed number of arguments.

Regardless of the value of the recursive variable $y+1$, the preceding results can be encoded in the Gödel number $gn_f(y)$. This observation provides the framework for a formal definition of course-of-values recursion.

Definition 13.5.2

Let g and h be $n+2$-variable total number-theoretic functions, respectively. The $n+1$-variable function f defined by

i) $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$
ii) $f(x_1, \ldots, x_n, y+1) = h(x_1, \ldots, x_n, y, gn_f(x_1, \ldots, x_n, y))$

is said to be obtained from g and h by **course-of-values recursion.**

Theorem 13.5.3

Let f be an $n+1$-variable function defined by course-of-values recursion from primitive recursive functions g and h. Then f is primitive recursive.

Proof. We begin by defining gn_f by primitive recursion directly from the primitive recursive functions g and h.

$$\begin{aligned}
gn_f(x_1, \ldots, x_n, 0) &= 2^{f(x_1,\ldots,x_n,0)+1}\\
&= 2^{g(x_1,\ldots,x_n)+1}\\
gn_f(x_1, \ldots, x_n, y+1) &= gn_f(x_1, \ldots, x_n, y)\cdot pn(y+1)^{f(x_1,\ldots,x_n,y+1)+1}\\
&= gn_f(x_1, \ldots, x_n, y)\cdot pn(y+1)^{h(x_1,\ldots,x_n,y,gn_f(x_1,\ldots,x_n,y))+1}
\end{aligned}$$

The evaluation of $gn_f(x_1, \ldots, x_n, y+1)$ uses only

i) the parameters $x_0, \ldots, x_n$,
ii) y, the previous value of the recursive variable,
iii) $gn_f(x_1, \ldots, x_n, y)$, the previous value of gn_f, and
iv) the primitive recursive functions h, pn, $\cdot$, $+$, and exponentiation.

Thus, the function gn_f is primitive recursive. By Theorem 13.5.1, it follows that f is also primitive recursive. ■

In mechanical terms, the Gödel numbering gives computation the equivalent of unlimited memory. A single Gödel number is capable of storing any number of preliminary results. The Gödel numbering encodes the values $f(x_0, \ldots, x_n, 0)$, $f(x_0, \ldots, x_n, 1), \ldots$, $f(x_0, \ldots, x_n, y)$ that are required for the computation of $f(x_0, \ldots, x_n, y+1)$. The decoding function provides the connection between the memory and the computation. Whenever a stored value is needed by the computation, the decoding function makes it available.

Example 13.5.2

Let h be the primitive recursive function

$$h(x, y) = \sum_{i=0}^{x} dec(i, y) \cdot (x + 1 - i).$$

The function f, which was defined earlier to introduce course-of-values computation, can be defined by course-of-values recursion from h.

$$\begin{aligned} f(0) &= 1 \\ f(y+1) &= h(y, gn_f(y)) = \sum_{i=0}^{y} dec(i, gn_f(y)) \cdot (y + 1 - i) \\ &= \sum_{i=0}^{y} f(i) \cdot (y + 1 - i) \end{aligned}$$ □

13.6 Computable Partial Functions

The primitive recursive functions were defined as a family of intuitively computable functions. We have established that all primitive recursive functions are total. Conversely, are all computable total functions primitive recursive? Moreover, should we restrict our analysis of computability to total functions? In this section we will present arguments for a negative response to both of these questions.

We will use a diagonalization argument to establish the existence of a total computable function that is not primitive recursive. The first step is to show that the syntactic structure of the primitive recursive functions allows them to be effectively enumerated. The ability to list the primitive recursive functions permits the construction of a computable function that differs from every function in the list.

Theorem 13.6.1

The set of primitive recursive functions is a proper subset of the set of effectively computable total number-theoretic functions.

Proof. The primitive recursive functions can be represented as strings over the alphabet $\Sigma = \{s, p, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), \circ, :, \langle, \rangle\}$. The basic functions s, z, and $p_i^{(j)}$ are represented by $\langle s \rangle$, $\langle z \rangle$, and $\langle pi(j) \rangle$. The composition $h \circ (g_1, \ldots, g_n)$ is encoded $\langle \langle h \rangle \circ (\langle g_1 \rangle, \ldots, \langle g_n \rangle) \rangle$, where $\langle h \rangle$ and $\langle g_i \rangle$ are the representations of the constituent functions. A function defined by primitive recursion from functions g and h is represented by $\langle \langle g \rangle : \langle h \rangle \rangle$.

The strings in Σ^* can be generated by length: first the null string, followed by strings of length one, length two, and so on. A straightforward mechanical process can be designed to determine whether a string represents a correctly formed primitive recursive function. The enumeration of the primitive recursive functions is accomplished by repeatedly generating a string and determining if it is a syntactically correct representation of a function. The first correctly formed string is denoted f_0, the next f_1, and so on. In the same manner, we can enumerate the one-variable primitive recursive functions. This is accomplished by deleting all n-variable functions, $n > 1$, from the previously generated list. This sequence is denoted $f_0^{(1)}, f_1^{(1)}, f_2^{(1)}, \ldots$.

The total one-variable function

$$g(i) = f_i^{(1)}(i) + 1$$

is effectively computable. The effective enumeration of the one-variable primitive recursive functions establishes the computability of g. The value $g(i)$ is obtained by

i) determining the ith one-variable primitive recursive function $f_i^{(1)}$,

ii) computing $f_i^{(1)}(i)$, and

iii) adding one to $f_i^{(1)}(i)$.

Since each of these steps is effective, we conclude that g is computable. By the familiar diagonalization argument,

$$g(i) \neq f_i^{(1)}(i)$$

for any i. Consequently, g is total and computable but not primitive recursive. ■

Theorem 13.6.1 used diagonalization to demonstrate the existence of computable functions that are not primitive recursive. This can also be accomplished directly by constructing a computable function that is not primitive recursive. The two-variable number-theoretic function, known as *Ackermann's function*, defined by

i) $A(0, y) = y + 1$

ii) $A(x + 1, 0) = A(x, 1)$

iii) $A(x + 1, y + 1) = A(x, A(x + 1, y))$

is one such function. The values of A are defined recursively with the basis given in condition (i). A proof by induction on x establishes that A is uniquely defined for every pair of input values (Exercise 22). The computations in Example 13.6.1 illustrate the computability of Ackermann's function.

Example 13.6.1

The values $A(1, 1)$ and $A(3, 0)$ are constructed from the definition of Ackermann's function. The column on the right gives the justification for the substitution.

a) $A(1, 1) = A(0, A(1, 0))$ (iii)
$= A(0, A(0, 1))$ (ii)
$= A(0, 2)$ (i)
$= 3$

b) $A(2, 1) = A(1, A(2, 0))$ (iii)
$= A(1, A(1, 1))$ (ii)
$= A(1, 3)$ (a)
$= A(0, A(1, 2))$ (iii)
$= A(0, A(0, A(1, 1)))$ (iii)
$= A(0, A(0, 3))$ (a)
$= A(0, 4)$ (i)
$= 5$ (i) □

The values of Ackermann's function exhibit a remarkable rate of growth. By fixing the first variable, Ackermann's function generates the one-variable functions

$$A(1, y) = y + 2$$
$$A(2, y) = 2y + 3$$
$$A(3, y) = 2^{y+3} - 3$$
$$A(4, y) = 2^{2^{\cdot^{\cdot^{2^{16}}}}} - 3.$$

The number of 2's in the exponential chain in $A(4, y)$ is y. For example, $A(4, 0) = 16 - 3$, $A(4, 1) = 2^{16} - 3$, and $A(4, 2) = 2^{2^{16}} - 3$. The first variable of Ackermann's function determines the rate of growth of the function values. We state, without proof, the following theorem that compares the rate of growth of Ackermann's function with that of the primitive recursive functions.

Theorem 13.6.2

For every one-variable primitive recursive function f, there is some $i \in \mathbf{N}$ such that $f(i) < A(i, i)$.

Clearly, the one-variable function $A(i, i)$ obtained by identifying the variables of A is not primitive recursive. It follows that Ackermann's function is not primitive recursive. If it

were, then $A(i, i)$, which can be obtained by the composition $A \circ (p_1^{(1)}, p_1^{(1)})$, would also be primitive recursive.

Is it possible to increase the set of primitive recursive functions, possibly by adding some new basic functions or additional operations, to include all total computable functions? Unfortunately, the answer is no. Regardless of the set of total functions that we consider computable, the diagonalization argument in the proof of Theorem 13.6.1 can be used to show that there is no effective enumeration of all total computable functions. Therefore, we must conclude that the computable functions cannot be effectively generated or that there are computable nontotal functions. If we accept the latter proposition, the contradiction from the diagonalization disappears. The reason we can claim that g is not one of the f_i's is that $g(i) \neq f_i^{(1)}(i)$. If $f_i^{(1)}(i)\uparrow$, then $g(i) = f_i^{(i)}(i) + 1$ is also undefined. If we wish to be able to effectively enumerate the computable functions, it is necessary to include partial functions in the enumeration.

We now consider the computability of partial functions. Since composition and primitive recursion preserve totality, an additional operation is needed to construct partial functions from the basic functions. Minimalization has been informally described as a search procedure. Placing a bound on the range of the natural numbers to be examined ensures that the bounded minimalization operation produces total functions. *Unbounded minimalization* is obtained by performing the search without an upper limit on the set of natural numbers to be considered. The function

$$f(x) = \mu z[eq(x, z \cdot z)]$$

defined by unbounded minimalization returns the square root of x whenever x is a perfect square. Otherwise, the search for the first natural number satisfying the predicate continues ad infinitum. Although eq is a total function, the resulting function f is not. For example, $f(3)\uparrow$. A function defined by unbounded minimalization is undefined for input x whenever the search fails to return a value.

The introduction of partial functions forces us to reexamine the operations of composition and primitive recursion. The possibility of undefined values was considered in the definition of composition. The function $h \circ (g_1, \ldots, g_n)$ is undefined for input $x_1, \ldots, x_k$ if either

i) $g_i(x_1, \ldots, x_k)\uparrow$ for some $1 \le i \le n$; or
ii) $g_i(x_1, \ldots, x_k)\downarrow$ for all $1 \le i \le n$ and $h(g_1(x_1, \ldots, x_k), \ldots g_n(x_1, \ldots, x_k))\uparrow$.

An undefined value propagates from any of the g_i's to the composite function.

The operation of primitive recursion required both of the defining functions g and h to be total. This restriction is relaxed to permit definitions by primitive recursion using partial functions. Let f be defined by primitive recursion from partial functions g and h.

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$
$$f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$$

Determining the value of a function defined by primitive recursion is an iterative process. The function f is defined for recursive variable y only if the following conditions are satisfied:

i) $f(x_1, \ldots, x_n, 0) \downarrow$ if $g(x_1, \ldots, x_n) \downarrow$

ii) $f(x_1, \ldots, x_n, y+1) \downarrow$ if $f(x_1, \ldots, x_n, i) \downarrow$ for $0 \le i \le y$ and $h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) \downarrow$.

An undefined value for the recursive variable causes f to be undefined for all the subsequent values of the recursive variable.

With the conventions established for definitions with partial functions, a family of computable partial functions can be defined using the operations composition, primitive recursion, and unbounded minimalization.

Definition 13.6.3

The family of **μ-recursive functions** is defined as follows:

i) The successor, zero, and projection functions are μ-recursive.

ii) If h is an n-variable μ-recursive function and $g_1, \ldots, g_n$ are k-variable μ-recursive functions, then $f = h \circ (g_1, \ldots, g_n)$ is μ-recursive.

iii) If g and h are n and $n+2$-variable μ-recursive functions, then the function f defined from g and h by primitive recursion is μ-recursive.

iv) If $p(x_1, \ldots, x_n, y)$ is a total μ-recursive predicate, then $f = \mu z[p(x_1, \ldots, x_n, z)]$ is μ-recursive.

v) A function is μ-recursive only if it can be obtained from condition (i) by a finite number of applications of the rules in (ii), (iii), and (iv).

Conditions (i), (ii), and (iii) imply that all primitive recursive functions are μ-recursive. Notice that unbounded minimalization is not defined for all predicates, but only for total μ-recursive predicates.

The notion of Turing computability encompasses partial functions in a natural way. A Turing machine computes a partial number-theoretic function f if

i) the computation terminates with result $f(x_1, \ldots, x_n)$ whenever $f(x_1, \ldots, x_n) \downarrow$, and

ii) the computation does not terminate whenever $f(x_1, \ldots, x_n) \uparrow$.

The Turing machine computes the value of the function whenever possible. Otherwise, the computation continues indefinitely.

We will now establish the relationship between the μ-recursive and Turing computable functions. The first step is to show that every μ-recursive function is Turing computable. This is not a surprising result; it simply extends Theorem 13.1.3 to partial functions.

Theorem 13.6.4

Every μ-recursive function is Turing computable.

Proof. Since the basic functions are known to be Turing computable, the proof consists of showing that the Turing computable partial functions are closed under operations of composition, primitive recursion, and unbounded minimalization. The techniques developed in Theorems 9.4.3 and 13.1.3 demonstrate the closure of Turing computable total functions under composition and primitive recursion, respectively. These machines also establish the closure for partial functions. An undefined value in one of the constituent computations causes the entire computation to continue indefinitely.

The proof is completed by showing that the unbounded minimalization of a Turing computable total predicate is Turing computable. Let $f(x_1, \ldots, x_n) = \mu z[p(x_1, \ldots, x_n, y)]$ where $p(x_1, \ldots, x_n, y)$ is a total Turing computable predicate. A Turing machine to compute f can be constructed from P, the machine that computes the predicate p. The initial configuration of the tape is $B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B$.

1. The representation of the number zero is added to the right of the input. The search specified by the minimalization operator begins with the tape configuration

$$B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{0}B.$$

The number to the right of the input, call it j, is the index for the minimalization operator.

2. A working copy of the parameters and j is made, producing the tape configuration

$$B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{j} B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{j} B.$$

3. The machine P is run with the input consisting of the copy of the parameters and j, producing

$$B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{j} B\overline{p(x_1, x_2, \ldots, x_n, j)} B.$$

4. If $p(x_1, x_2, \ldots, x_n, j) = 1$, the value of the minimalization of p is j. Otherwise, the $p(x_1, x_2, \ldots, x_n, j)$ is erased, j is incremented, and the computation continues with step 2.

A computation terminates at step 4 when the first j for which $p(x_1, \ldots, x_n, j) = 1$ is encountered. If no such value exists, the computation loops indefinitely, indicating that the function f is undefined. ■

13.7 Turing Computability and Mu-Recursive Functions

It has already been established that every μ-recursive function can be computed by a Turing machine. We now turn our attention to the opposite inclusion, that every Turing computable function is μ-recursive. To show this, a number-theoretic function is designed to simulate the computations of a Turing machine. The construction of the simulating function requires moving from the domain of machines to the domain of natural numbers. The process of

Exercises

1. Let $g(x) = x^2$ and $h(x, y, z) = x + y + z$, and let $f(x, y)$ be the function defined from g and f by primitive recursion. Compute the values $f(1, 0)$, $f(1, 1)$, $f(1, 2)$ and $f(5, 0)$, $f(5, 1)$, $f(5, 2)$.
2. Using only the basic functions, composition, and primitive recursion, show that the following functions are primitive recursive. When using primitive recursion, give the functions g and h.
   a) $c_2^{(3)}$
   b) $pred$
   c) $f(x) = 2x + 2$
3. The functions below were defined by primitive recursion in Table 13.1. Explicitly, give the functions g and h that constitute the definition by primitive recursion.
   a) sg
   b) sub
   c) exp
4. a) Prove that a function f defined by the composition of total functions h and $g_1, \ldots, g_n$ is total.
   b) Prove that a function f defined by primitive recursion from total functions g and h is total.
   c) Conclude that all primitive recursive functions are total.
5. Let $g = id$, $h = p_1^{(3)} + p_3^{(3)}$, and let f be defined from g and h by primitive recursion.
   a) Compute the values $f(3, 0)$, $f(3, 1)$, and $f(3, 2)$.
   b) Give a closed-form (nonrecursive) definition of the function f.
6. Let $g(x, y, z)$ be a primitive recursive function. Show that each of the following functions is primitive recursive.
   a) $f(x, y) = g(x, y, x)$
   b) $f(x, y, z, w) = g(x, y, x)$
   c) $f(x) = g(1, 2, x)$
7. Let f be the function

$$f(x) = \begin{cases} x & \text{if } x > 2 \\ 0 & \text{otherwise.} \end{cases}$$

   a) Give the state diagram of a Turing machine that computes f.
   b) Show that f is primitive recursive.

8. Show that the following functions are primitive recursive. You may use the functions and predicates from Tables 13.1 and 13.2. Do not use the bounded operations.
   a) $max(x, y) = \begin{cases} x & \text{if } x \geq y \\ y & \text{otherwise} \end{cases}$
   b) $min(x, y) = \begin{cases} x & \text{if } x \leq y \\ y & \text{otherwise} \end{cases}$
   c) $min_3(x, y, z) = \begin{cases} x & \text{if } x \leq y \text{ and } x \leq z \\ y & \text{if } y \leq x \text{ and } y \leq z \\ z & \text{if } z \leq x \text{ and } z \leq y \end{cases}$
   d) $even(x) = \begin{cases} 1 & \text{if } x \text{ is even} \\ 0 & \text{otherwise} \end{cases}$
   e) $half(x) = div(x, 2)$
   *f) $sqrt(x) = \lfloor \sqrt{x} \rfloor$
9. Show that the following predicates are primitive recursive. You may use the functions and predicates from Tables 13.1 and 13.2 and Exercise 8. Do not use the bounded operators.
   a) $le(x, y) = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{otherwise} \end{cases}$
   b) $ge(x, y) = \begin{cases} 1 & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$
   c) $btw(x, y, z) = \begin{cases} 1 & \text{if } y < x < z \\ 0 & \text{otherwise} \end{cases}$
   d) $prsq(x) = \begin{cases} 1 & \text{if } x \text{ is a perfect square} \\ 0 & \text{otherwise} \end{cases}$
10. Let t be a two-variable primitive recursive function and define f as follows:

$$f(x, 0) = t(x, 0)$$
$$f(x, y + 1) = f(x, y) + t(x, y + 1)$$

   Explicitly give the functions g and h that define f by primitive recursion.
11. Let g and h be primitive recursive functions. Use bounded operators to show that the following functions are primitive recursive. You may use any functions and predicates that have been shown to be primitive recursive.
   a) $f(x, y) = \begin{cases} 1 & \text{if } g(i) < g(x) \text{ for all } 0 \leq i \leq y \\ 0 & \text{otherwise} \end{cases}$

b) $f(x, y) = \begin{cases} 1 & \text{if } g(i) = x \text{ for some } 0 \le i \le y \\ 0 & \text{otherwise} \end{cases}$

c) $f(y) = \begin{cases} 1 & \text{if } g(i) = h(j) \text{ for some } 0 \le i, j \le y \\ 0 & \text{otherwise} \end{cases}$

d) $f(y) = \begin{cases} 1 & \text{if } g(i) < g(i+1) \text{ for all } 0 \le i \le y \\ 0 & \text{otherwise} \end{cases}$

e) $nt(x, y) =$ the number of times $g(i) = x$ in the range $0 \le i \le y$

f) $thrd(x, y) = \begin{cases} 0 & \text{if } g(i) \text{ does not assume the value } x \text{ at least} \\ & \text{three times in the range } 0 \le i \le y \\ j & \text{if } j \text{ is the third integer in the range } 0 \le i \le y \\ & \text{for which } g(i) = x \end{cases}$

g) $lrg(x, y) =$ the largest value in the range $0 \le i \le y$ for which $g(i) = x$

12. Show that the following functions are primitive recursive.

a) $gcd(x, y) =$ the greatest common divisor of x and y

b) $lcm(x, y) =$ the least common multiple of x and y

c) $pw2(x) = \begin{cases} 1 & \text{if } x = 2^n \text{ for some } n \\ 0 & \text{otherwise} \end{cases}$

d) $twopr(x) = \begin{cases} 1 & \text{if } x \text{ is the product of exactly two primes} \\ 0 & \text{otherwise} \end{cases}$

* **13.** Let g be a one-variable primitive recursive function. Prove that the function

$$\begin{aligned} f(x) &= \overset{x}{\underset{i=0}{\min}}(g(i)) \\ &= min\{g(0), \ldots, g(x)\} \end{aligned}$$

is primitive recursive.

14. Prove that the function

$$f(x_1, \ldots, x_n) = \overset{u(x_1,\ldots,x_n)}{\mu z}[p(x_1, \ldots, x_n, z)]$$

is primitive recursive whenever p and u are primitive recursive.

15. Compute the Gödel number for the following sequence:

a) 3, 0

b) 0, 0, 1

c) 1, 0, 1, 2

d) 0, 1, 1, 2, 0

16. Determine the sequences encoded by the following Gödel numbers:

a) 18,000

b) 131,072

c) 2,286,900

d) 510,510

17. Prove that the following functions are primitive recursive:

a) $gdn(x) = \begin{cases} 1 & \text{if } x \text{ is the Gödel number of some sequence} \\ 0 & \text{otherwise} \end{cases}$

b) $gdln(x) = \begin{cases} n & \text{if } x \text{ is the Gödel number of a sequence of length } n \\ 0 & \text{otherwise} \end{cases}$

c) $g(x, y) = \begin{cases} 1 & \text{if } x \text{ is a Gödel number and } y \text{ occurs in the sequence encoded in } x \\ 0 & \text{otherwise} \end{cases}$

18. Construct a primitive recursive function whose input is an encoded ordered pair and whose output is the encoding of an ordered pair in which the positions of the elements have been swapped. For example, if the input is the encoding of $[x, y]$, then the output is the encoding of $[y, x]$.

19. Let f be the function defined by

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ 2 & \text{if } x = 1 \\ 3 & \text{if } x = 2 \\ f(x-3) + f(x-1) & \text{otherwise.} \end{cases}$$

Give the values $f(4)$, $f(5)$, and $f(6)$. Prove that f is primitive recursive.

* **20.** Let g_1 and g_2 be one-variable primitive recursive functions. Also let h_1 and h_2 be four-variable primitive recursive functions. The two functions f_1 and f_2 defined by

$$\begin{aligned} f_1(x, 0) &= g_1(x) \\ f_2(x, 0) &= g_2(x) \\ f_1(x, y+1) &= h_1(x, y, f_1(x, y), f_2(x, y)) \\ f_2(x, y+1) &= h_2(x, y, f_1(x, y), f_2(x, y)) \end{aligned}$$

are said to be constructed by *simultaneous recursion* from g_1, g_2, h_1, and h_2. The values $f_1(x, y+1)$ and $f_2(x, y+1)$ are defined in terms of the previous values of both of the functions. Prove that f_1 and f_2 are primitive recursive.

21. Let f be the function defined by

$$f(0) = 1$$
$$f(y+1) = \sum_{i=0}^{y} f(i)^y.$$

a) Compute $f(1)$, $f(2)$, and $f(3)$.
b) Use course-of-values recursion to show that f is primitive recursive.

22. Let A be Ackermann's function (see Section 13.6).
a) Compute $A(2, 2)$.
b) Prove that $A(x, y)$ has a unique value for every $x, y \in \mathbf{N}$.
c) Prove that $A(1, y) = y + 2$.
d) Prove that $A(2, y) = 2y + 3$.

23. Prove that the following functions are μ-recursive. The functions g and h are assumed to be primitive recursive.

a) $cube(x) = \begin{cases} 1 & \text{if } x \text{ is a perfect cube} \\ \uparrow & \text{otherwise} \end{cases}$

b) $root(c_0, c_1, c_2) =$ the smallest natural number root of the quadratic polynomial $c_2 \cdot x_2 + c_1 \cdot x + c_0$

c) $r(x) = \begin{cases} 1 & \text{if } g(i) = g(i + x) \text{ for some } i \geq 0 \\ \uparrow & \text{otherwise} \end{cases}$

d) $l(x) = \begin{cases} \uparrow & \text{if } g(i) - h(i) < x \text{ for all } i \geq 0 \\ 0 & \text{otherwise} \end{cases}$

e) $f(x) = \begin{cases} 1 & \text{if } g(i) + h(j) = x \text{ for some } i, j \in \mathbf{N} \\ \uparrow & \text{otherwise} \end{cases}$

f) $f(x) = \begin{cases} 1 & \text{if } g(y) = h(z) \text{ for some } y > x,\ z > x \\ \uparrow & \text{otherwise.} \end{cases}$

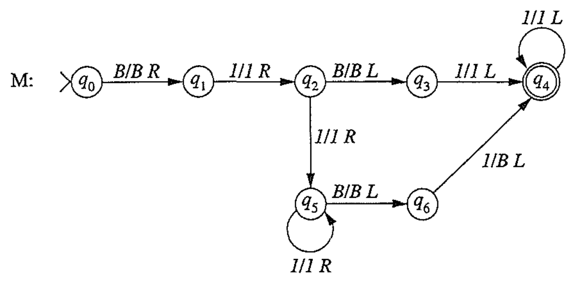
*24. The unbounded μ-operator can be defined for partial predicates as follows:

$$\mu z[p(x_1, \ldots, x_n, z)] = \begin{cases} j & \text{if } p(x_1, \ldots, x_n, i) = 0 \text{ for } 0 \leq i < j \\ & \quad \text{and } p(x_1, \ldots, x_n, j) = 1 \\ \uparrow & \text{otherwise.} \end{cases}$$

That is, the value is undefined if $p(x_1, \ldots, x_n, i) \uparrow$ for some i occurring before the first value j for which $p(x_1, \ldots, x_n, j) = 1$. Prove that the family of functions obtained by replacing the unbounded minimalization operator in Definition 13.6.3 with the preceding μ-operator is the family of Turing computable functions.

25. Construct the functions ns, ntp, and nts for the Turing machine S given in Example 13.7.1.

26. Let M be the machine



a) What unary number-theoretic function does M compute?
b) Give the tape numbers for each configuration that occurs in the computation of M with input $\overline{0}$.
c) Give the tape numbers for each configuration that occurs in the computation of M with input $\overline{2}$.

27. Let f be the function defined by

$$f(x) = \begin{cases} x + 1 & \text{if } x \text{ even} \\ x - 1 & \text{otherwise.} \end{cases}$$

a) Give the state diagram of a Turing machine M that computes f.
b) Trace the computation of your machine for input 1 ($B11B$). Give the tape number for each configuration in the computation. Give the value of $tr_M(1, i)$ for each step in the computation.
c) Show that f is primitive recursive. You may use the functions from the text that have been shown to be primitive recursive in Sections 13.1, 13.2, and 13.4.

*28. Let M be a Turing machine and tr_M the trace function of M.
a) Show that the function

$$prt(x, y) = \begin{cases} 1 & \text{if the } y\text{th transition of M with input } x \text{ prints} \\ & \text{a blank} \\ 0 & \text{otherwise} \end{cases}$$

is primitive recursive.