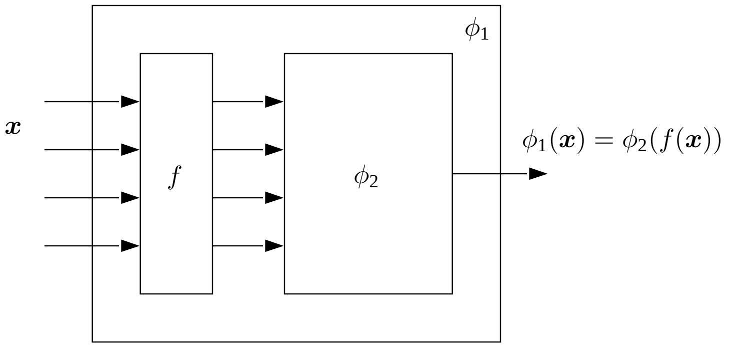

Figure 5.12 The characteristic function ϕ_i of L_i, $i = 1, 2$ has value 1 on strings in L_i and 0 otherwise. Because the language L_1 is reducible to the language L_2, there is a function f such that for all $\boldsymbol{x}$, $\phi_1(\boldsymbol{x}) = \phi_2(f(\boldsymbol{x}))$.

Reducibility is a fundamental idea that is formally introduced in Section 2.4 and used throughout this book. Reductions of the type defined above are known as **many-to-one reductions**. (See Section 8.7 for more on this subject.)

The following lemma is a tool to show that problems are unsolvable. We use the same mechanism in Chapter 8 to classify languages by their use of time, space and other computational resources.

LEMMA 5.8.1 *Let L_1 be reducible to L_2. If L_2 is decidable, then L_1 is decidable. If L_1 is unsolvable and L_2 is recursively enumerable, L_2 is also unsolvable.*

Proof Let T be a Turing machine implementing the algorithm that translates strings over the alphabet of L_1 to strings over the alphabet of L_2. If L_2 is decidable, there is a halting Turing machine M_2 that accepts it. A multi-tape Turing machine M_1 that decides L_1 can be constructed as follows: On input string $\boldsymbol{w}$, M_1 invokes T to generate the string $\boldsymbol{z}$, which it then passes to M_2. If M_2 accepts $\boldsymbol{z}$, M_1 accepts $\boldsymbol{w}$. If M_2 rejects it, so does M_1. Thus, M_1 decides L_1.

Suppose now that L_1 is unsolvable. Assuming that L_2 is decidable, from the above construction, L_1 is decidable, contradicting this assumption. Thus, L_2 cannot be decidable. ■

The power of this lemma will be apparent in the next section.

5.8.2 Unsolvable Problems

In this section we examine six representative unsolvable problems. They range from the classical halting problem to Rice's theorem.

We begin by considering the **halting problem** for Turing machines. The problem is to determine for an arbitrary TM M and an arbitrary input string $\boldsymbol{x}$ whether M with input $\boldsymbol{x}$ halts or not. We characterize this problem by the language $\mathcal{L}_H$ shown below. We show it is unsolvable, that is, $\mathcal{L}_H$ is recursively enumerable but not decidable. No Turing machine exists to decide this language.

$$\mathcal{L}_H = \{\rho(M), \boldsymbol{w} \mid M \text{ halts on input } \boldsymbol{w}\}$$

THEOREM 5.8.1 *The language $\mathcal{L}_H$ is recursively enumerable but not decidable.*

Proof To show that $\mathcal{L}_H$ is recursively enumerable, pass the encoding $\rho(M)$ of the TM M and the input string $\boldsymbol{w}$ to the universal Turing machine U of Section 5.5. This machine simulates M and halts on the input $\boldsymbol{w}$ if and only if M halts on $\boldsymbol{w}$. Thus, $\mathcal{L}_H$ is recursively enumerable.

To show that $\mathcal{L}_H$ is undecidable, we follow the pattern described in Lemma 5.8.1. We assume that $\mathcal{L}_H$ is decidable by a Turing machine M_H and exhibit a TM that translates $\mathcal{L}_1$ to $\mathcal{L}_H$. This implies that $\mathcal{L}_1$ is decidable, which is a contradiction.

We assume that M_H exists and has an encoding $\rho(M_H)$. We use this encoding to construct a Turing machine M_1 that decides $\mathcal{L}_1$ as follows: a) given the input string $\boldsymbol{w}$, M_1 determines the value of i such that $\boldsymbol{w}$ is the ith string, $\boldsymbol{x}_i$, in the lexicographical ordering of input strings; b) M_1 generates an encoding for the ith Turing machine M_i using the procedure described in Section 5.6; c) M_1 simulates M_H to determine if M_i halts on $\boldsymbol{w} = \boldsymbol{x}_i$; d) if M_H says that M_i does not halt, M_1 accepts $\boldsymbol{w}$; e) if M_H says that M_i does halt, M_1 simulates M_i on input string $\boldsymbol{w}$. M_1 rejects $\boldsymbol{w}$ if M_i accepts it and accepts $\boldsymbol{w}$ if M_i rejects it. Clearly, M_1 recognizes strings in $\mathcal{L}_1$, which contradicts the nature of $\mathcal{L}_1$. Thus, M_H cannot exist. ■

The second unsolvable problem we consider is the **empty tape acceptance problem**: given a Turing machine M, we ask if we can tell whether it accepts the empty string. We reduce the halting problem to it. (See Fig. 5.13.)

$$\mathcal{L}_{\mathrm{ET}} = \{\rho(M) \mid L(M) \text{ contains the empty string}\}$$

THEOREM 5.8.2 *The language $\mathcal{L}_{\mathrm{ET}}$ is not decidable.*

Proof To show that $\mathcal{L}_{\mathrm{ET}}$ is not decidable, we assume that it is and derive a contradiction. The contradiction is produced by assuming the existence of a TM M_{ET} that decides $\mathcal{L}_{\mathrm{ET}}$ and then showing that this implies the existence of a TM M_H that decides $\mathcal{L}_{\mathrm{H}}$.

Given an encoding $\rho(M)$ for an arbitrary TM M and an arbitrary input $\boldsymbol{w}$, the TM M_H constructs a TM $T(M, \boldsymbol{w})$ that writes $\boldsymbol{w}$ on the tape when the tape is empty and simulates M on $\boldsymbol{w}$, halting if M halts. Thus, $T(M, \boldsymbol{w})$ accepts the empty tape if M halts on $\boldsymbol{w}$. M_H decides $\mathcal{L}_{\mathrm{H}}$ by constructing an encoding of $T(M, \boldsymbol{w})$ and passing it to M_{ET}. (See Fig. 5.13.) The language accepted by $T(M, \boldsymbol{w})$ includes the empty string if and only if M halts on $\boldsymbol{w}$. Thus, M_H decides the halting problem, which as shown earlier cannot be decided. ■

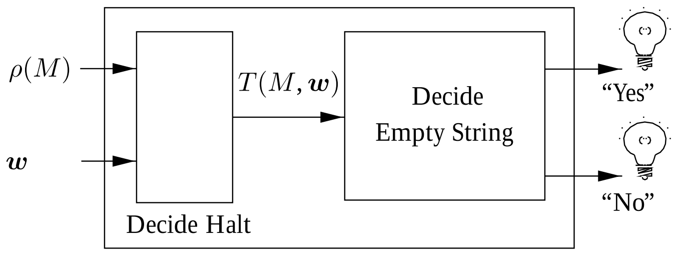

Figure 5.13 Schematic representation of the reduction from $\mathcal{L}_H$ to $\mathcal{L}_{\mathrm{ET}}$.

The third unsolvable problem we consider is the **empty set acceptance problem**: Given a Turing machine, we ask if we can tell if the language it accepts is empty. We reduce the halting problem to this language.

$$\mathcal{L}_{\mathrm{EL}} = \{\rho(M) \mid L(M) = \emptyset\}$$

THEOREM 5.8.3 *The language $\mathcal{L}_{\mathrm{EL}}$ is not decidable.*

Proof We reduce $\mathcal{L}_H$ to $\mathcal{L}_{\mathrm{EL}}$, assume that $\mathcal{L}_{\mathrm{EL}}$ is decidable by a TM M_{EL}, and then show that a TM M_H exists that decides $\mathcal{L}_H$, thereby establishing a contradiction.

Given an encoding $\rho(M)$ for an arbitrary TM M and an arbitrary input $\boldsymbol{w}$, the TM M_H constructs a TM $T(M, \boldsymbol{w})$ that accepts the string placed on its tape if it is $\boldsymbol{w}$ and M halts on it; otherwise it enters an infinite loop. M_H can implement $T(M, \boldsymbol{w})$ by entering an infinite loop if its input string is not $\boldsymbol{w}$ and otherwise simulating M on $\boldsymbol{w}$ with a universal Turing machine.

It follows that $L(T(M, \boldsymbol{w}))$ is empty if M does not halt on $\boldsymbol{w}$ and contains $\boldsymbol{w}$ if it does halt. Under the assumption that M_{EL} decides $\mathcal{L}_{\mathrm{EL}}$, M_H can decide $\mathcal{L}_{\mathrm{H}}$ by constructing $T(M, \boldsymbol{w})$ and passing it to M_{EL}, which accepts $\rho(T(M, \boldsymbol{w}))$ if M does not halt on $\boldsymbol{w}$ and rejects it if M does halt. Thus, M_H decides $\mathcal{L}_H$, a contradiction. ■

The fourth problem we consider is the **regular machine recognition problem**. In this case we ask if a Turing machine exists that can decide from the description of an arbitrary Turing machine M whether the language accepted by M is regular or not:

$$\mathcal{L}_R = \{\rho(M) \mid L(M) \text{ is regular}\}$$

THEOREM 5.8.4 *The language $\mathcal{L}_R$ is not decidable.*

Proof We assume that a TM M_R exists to decide $\mathcal{L}_R$ and show that this implies the existence of a TM M_H that decides $\mathcal{L}_H$, a contradiction. Thus, M_R cannot exist.

Given an encoding $\rho(M)$ for an arbitrary TM M and an arbitrary input $\boldsymbol{w}$, the TM M_H constructs a TM $T(M, \boldsymbol{w})$ that scans its tape. If it finds a string in $\{0^n1^n \mid n \geq 0\}$, it accepts it; if not, $T(M, \boldsymbol{w})$ erases the tape and simulates M on $\boldsymbol{w}$, halting only if M halts on $\boldsymbol{w}$. Thus, $T(M, \boldsymbol{w})$ accepts all strings in $\mathcal{B}^*$ if M halts on $\boldsymbol{w}$ but accepts only strings in $\{0^n1^n \mid n \geq 0\}$ otherwise. Thus, $T(M, \boldsymbol{w})$ accepts the regular language $\mathcal{B}^*$ if M halts on $\boldsymbol{w}$ and accepts the context-free language $\{0^n1^n \mid n \geq 0\}$ otherwise. Thus, M_H can be implemented by constructing $T(M, \boldsymbol{w})$ and passing it to M_R, which is presumed to decide $\mathcal{L}_R$. ■

The fifth problem generalizes the above result and is known as **Rice's theorem**. It says that no algorithm exists to determine from the description of a TM whether or not the language it accepts falls into any proper subset of the recursively enumerable languages.

Let **RE** be the set of recursively enumerable languages over $\mathcal{B}$. For each set $\mathcal{C}$ that is a proper subset of **RE**, define the following language:

$$\mathcal{L}_{\mathcal{C}} = \{\rho(M) \mid L(M) \in \mathcal{C}\}$$

Rice's theorem says that, for all $\mathcal{C}$ such that $\mathcal{C} \neq \emptyset$ and $\mathcal{C} \subset$ **RE**, the language $\mathcal{L}_{\mathcal{C}}$ defined above is undecidable.

THEOREM 5.8.5 (Rice) *Let $\mathcal{C} \subset$ **RE**, $\mathcal{C} \neq \emptyset$. The language $\mathcal{L}_{\mathcal{C}}$ is not decidable.*

Proof To prove that $\mathcal{L}_{\mathcal{C}}$ is not decidable, we assume that it is decidable by the TM $M_{\mathcal{C}}$ and show that this implies the existence of a TM M_H that decides $\mathcal{L}_H$, which has been shown previously not to exist. Thus, $M_{\mathcal{C}}$ cannot exist.

We consider two cases, the first in which $\mathcal{B}^*$ is in not $\mathcal{C}$ and the second in which it is in $\mathcal{C}$. In the first case, let L be a language in $\mathcal{C}$. In the second, let L be a language in **RE** $- \mathcal{C}$. Since $\mathcal{C}$ is a proper subset of **RE** and not empty, there is always a language L such that one of L and $\mathcal{B}^*$ is in $\mathcal{C}$ and the other is in its complement **RE** $- \mathcal{C}$.

Given an encoding $\rho(M)$ for an arbitrary TM M and an arbitrary input $\boldsymbol{w}$, the TM M_H constructs a (four-tape) TM $T(M, \boldsymbol{w})$ that simulates two machines in parallel (by alternatively simulating one step of each machine). The first, M_0, uses a phrase-structure grammar for L to see if $T(M, \boldsymbol{w})$'s input string $\boldsymbol{x}$ is in L; it holds $\boldsymbol{x}$ on one tape, holds the current choice inputs for the NDTM M_L of Theorem 5.4.2 on a second, and uses a third tape for the deterministic simulation of M_L. (See the comments following Theorem 5.4.2.) $T(M, \boldsymbol{w})$ halts if M_0 generates $\boldsymbol{x}$. The second TM writes $\boldsymbol{w}$ on the fourth tape and simulates M on it. $T(M, \boldsymbol{w})$ halts if M halts on $\boldsymbol{w}$. Thus, $T(M, \boldsymbol{w})$ accepts the regular language $\mathcal{B}^*$ if M halts on $\boldsymbol{w}$ and accepts L otherwise. Thus, M_H can be implemented by constructing $T(M, \boldsymbol{w})$ and passing it to $M_{\mathcal{C}}$, which is presumed to decide $\mathcal{L}_{\mathcal{C}}$. ■

Our last problem is the **self-terminating machine problem**. The question addressed is whether a Turing machine M given a description $\rho(M)$ of itself as input will halt or not. The problem is defined by the following language. We give a direct proof that it is undecidable; that is, we do not reduce some other problem to it.

$$\mathcal{L}_{\mathrm{ST}} = \{\rho(M) \mid M \text{ is self-terminating}\}$$

THEOREM 5.8.6 *The language $\mathcal{L}_{\mathrm{ST}}$ is recursively enumerable but not decidable.*

Proof To show that $\mathcal{L}_{\mathrm{ST}}$ is recursively enumerable we exhibit a TM T that accepts strings in $\mathcal{L}_{\mathrm{ST}}$. T makes a copy of its input string $\rho(M)$ and simulates M on $\rho(M)$ by passing $(\rho(M), \rho(M))$ to a universal TM that halts and accepts $\rho(M)$ if it is in $\mathcal{L}_{\mathrm{ST}}$.

To show that $\mathcal{L}_{\mathrm{ST}}$ is not decidable, we assume that it is and arrive at a contradiction. Let M_{ST} decide $\mathcal{L}_{\mathrm{ST}}$. We design a TM M^* that does the following: M^* simulates M_{ST} on the input string $\boldsymbol{w}$. If M_{ST} halts and accepts $\boldsymbol{w}$, M^* enters an infinite loop. If M_{ST} halts and rejects $\boldsymbol{w}$, M^* accepts $\boldsymbol{w}$. (M_{ST} halts on all inputs.)

The new machine M^* is either self-terminating or it is not. If M^* is self-terminating, then on input $\rho(M^*)$, which is an encoding of itself, M^* enters an infinite loop because M_{ST} detects that it is self-terminating. Thus, M^* is not self-terminating. On the other hand, if M^* is not self-terminating, on input $\rho(M^*)$ it halts and accepts $\rho(M^*)$ because M_{ST} detects that it is not self-terminating and enters the rejecting halt state. But this contradicts the assumption that M^* is not self-terminating. Since we arrive at a contradiction in both cases, the assumption that $\mathcal{L}_{\mathrm{ST}}$ is decidable must be false. ■

5.9 Functions Computed by Turing Machines

In this section we introduce the partial recursive functions, a family of functions in which each function is constructed from three basic function types, zero, successor, and projection, and three operations on functions, composition, primitive recursion, and minimalization. Although we do not have the space to show this, the functions computed by Turing machines are exactly the partial recursive functions. In this section, we show one half of this result, namely, that every partial recursive function can be encoded as a RAM program (see Section 3.4.3) that can be executed by Turing machines.

We begin with the primitive recursive functions then describe the partial recursive functions. We then show that partial recursive functions can be realized by RAM programs.

5.9.1 Primitive Recursive Functions

Let $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ be the set of non-negative integers. The partial recursive functions, $f : \mathbb{N}^n \mapsto \mathbb{N}^m$, map n-tuples of integers over $\mathbb{N}$ to m-tuples of integers in $\mathbb{N}$ for arbitrary n and m. Partial recursive functions may be partial functions. They are constructed from three base function types, the **successor function** $S : \mathbb{N} \mapsto \mathbb{N}$, where $S(x) = x + 1$, the **predecessor function** $P : \mathbb{N} \mapsto \mathbb{N}$, where $P(x)$ returns either 0 if $x = 0$ or the integer one less than x, and the **projection functions** $U^n_j : \mathbb{N}^n \mapsto \mathbb{N}$, $1 \leq j \leq n$, where $U^n_j(x_1, x_2, \ldots, x_n) = x_j$. These basic functions are combined using a finite number of applications of function composition, primitive recursion, and minimalization.

Function composition is studied in Chapters 2 and 6. A function $f : \mathbb{N}^n \mapsto \mathbb{N}$ of n arguments is defined by the composition of a function $g : \mathbb{N}^m \mapsto \mathbb{N}$ of m arguments with m functions $f_1 : \mathbb{N}^n \mapsto \mathbb{N}$, $f_2 : \mathbb{N}^n \mapsto \mathbb{N}$, $\ldots$, $f_m : \mathbb{N}^n \mapsto \mathbb{N}$, each of n arguments, as follows:

$$f(x_1, x_2, \ldots, x_n) = g(f_1(x_1, x_2, \ldots, x_n), \ldots, f_m(x_1, x_2, \ldots, x_n))$$

A function $f : \mathbb{N}^{n+1} \mapsto \mathbb{N}$ of $n + 1$ arguments is defined by **primitive recursion** from a function $g : \mathbb{N}^n \mapsto \mathbb{N}$ of n arguments and a function $h : \mathbb{N}^{n+2} \mapsto \mathbb{N}$ on $n + 2$ arguments if and only if for all values of $x_1, x_2, \ldots, x_n$ and y in $\mathbb{N}$:

$$\begin{aligned} f(x_1, x_2, \ldots, x_n, 0) &= g(x_1, x_2, \ldots, x_n) \\ f(x_1, x_2, \ldots, x_n, y + 1) &= h(x_1, x_2, \ldots, x_n, y, f(x_1, x_2, \ldots, x_n, y)) \end{aligned}$$

In the above definition if $n = 0$, we adopt the convention that the value of f is a constant. Thus, $f(x_1, x_2, \ldots, x_n, k)$ is defined recursively in terms of h and itself with k replaced by $k - 1$ unless $k = 0$.

DEFINITION 5.9.1 *The class of* **primitive recursive functions** *is the smallest class of functions that contains the base functions and is closed under composition and primitive recursion.*

Many functions of interest are primitive recursive. Among these is the **zero function** $Z : \mathbb{N} \mapsto \mathbb{N}$, where $Z(x) = 0$. It is defined by primitive recursion by $Z(0) = 0$ and

$$Z(x + 1) = U^2_2(x, Z(x))$$

Other important primitive recursive functions are addition, subtraction, multiplication, and division, as we now show. Let $f_{\text{add}} : \mathbb{N}^2 \mapsto \mathbb{N}$, $f_{\text{sub}} : \mathbb{N}^2 \mapsto \mathbb{N}$, $f_{\text{mult}} : \mathbb{N}^2 \mapsto \mathbb{N}$, and $f_{\text{div}} : \mathbb{N}^2 \mapsto \mathbb{N}$ denote integer addition, subtraction, multiplication, and division.

For the **integer addition function** f_{add} introduce the function $h_1 : \mathbb{N}^3 \mapsto \mathbb{N}$ on three arguments, where h_1 is defined below in terms of the successor and projection functions:

$$h_1(x_1, x_2, x_3) = S(U^3_3(x_1, x_2, x_3))$$

Then, $h_1(x_1, x_2, x_3) = x_3 + 1$. Now define $f_{\mathrm{add}}(x, y)$ using primitive recursion, as follows:

$$\begin{aligned} f_{\mathrm{add}}(x, 0) &= U_1^1(x) \\ f_{\mathrm{add}}(x, y+1) &= h_1(x, y, f_{\mathrm{add}}(x, y)) \end{aligned}$$

The role of h is to carry the values of x and y from one recursive invocation to another. To determine the value of $f_{\mathrm{add}}(x, y)$ from this definition, if $y = 0$, $f_{\mathrm{add}}(x, y) = x$. If $y > 0$, $f_{\mathrm{add}}(x, y) = h_1(x, y-1, f_{\mathrm{add}}(x, y-1))$. This in turn causes other recursive invocations of f_{add}. The infix notation $+$ is used for f_{add}; that is, $f_{\mathrm{add}}(x, y) = x + y$.

Because the primitive recursive functions are defined over the non-negative integers, the subtraction function $f_{\mathrm{sub}}(x, y)$ must return the value 0 if y is larger than x, an operation called **proper subtraction**. (Its infix notation is $\dot{-}$ and we write $f_{\mathrm{sub}}(x, y) = x \dot{-} y$.) It is defined as follows:

$$\begin{aligned} f_{\mathrm{sub}}(x, 0) &= U_1^1(x) \\ f_{\mathrm{sub}}(x, y+1) &= U_3^3(x, y, P(f_{\mathrm{sub}}(x, y))) \end{aligned}$$

The value of $f_{\mathrm{sub}}(x, y)$ is x if $y = 0$ and is the predecessor of $f_{\mathrm{sub}}(x, y-1)$ otherwise.

The **integer multiplication function**, f_{mult}, is defined in terms of the function $h_2 : \mathbb{N}^3 \mapsto \mathbb{N}$:

$$h_2(x_1, x_2, x_3) = f_{\mathrm{add}}(U_1^3(x_1, x_2, x_3), U_3^3(x_1, x_2, x_3))$$

Using primitive recursion, we have

$$\begin{aligned} f_{\mathrm{mult}}(x, 0) &= Z(x) \\ f_{\mathrm{mult}}(x, y+1) &= h_2(x, y, f_{\mathrm{mult}}(x, y)) \end{aligned}$$

The value of $f_{\mathrm{mult}}(x, y)$ is zero if $y = 0$ and otherwise is the result of adding x to itself y times. To see this, note that the value of h_2 is the sum of its first and third arguments, x and $f_{\mathrm{mult}}(x, y)$. On each invocation of primitive recursion the value of y is decremented by 1 until the value 0 is reached. The definition of the division function is left as Problem 5.26.

Define the function $f_{\mathrm{sign}} : \mathbb{N} \mapsto \mathbb{N}$ so that $f_{\mathrm{sign}}(0) = 0$ and $f_{\mathrm{sign}}(x+1) = 1$. To show that f_{sign} is primitive recursive it suffices to invoke the projection operator formally. A function with value 0 or 1 is called a **predicate**.

5.9.2 Partial Recursive Functions

The partial recursive functions are obtained by extending the primitive recursive functions to include minimalization. **Minimalization** defines a function $f : \mathbb{N}^n \mapsto \mathbb{N}$ in terms of a second function $g : \mathbb{N}^{n+1} \mapsto \mathbb{N}$ by letting $f(\boldsymbol{x})$ be the smallest integer $y \in \mathbb{N}$ such that $g(\boldsymbol{x}, y) = 0$ and $g(\boldsymbol{x}, z)$ is defined for all $z \leq y$, $z \in \mathbb{N}$. Note that if $g(\boldsymbol{x}, z)$ is not defined for all $z \leq y$, then $f(\boldsymbol{x})$ is not defined. Thus, minimalization can result in partial functions.

DEFINITION 5.9.2 *The set of* **partial recursive functions** *is the smallest set of functions containing the base functions that is closed under composition, primitive recursion, and minimalization.*

A partial recursive function that is defined for all points in its domain is called a **recursive function**.

5.9.3 Partial Recursive Functions are RAM-Computable

There is a nice correspondence between RAM programs and partial recursive functions. The straight-line programs result from applying composition to the base functions. Adding primitive recursion corresponds to adding for-loops whereas adding minimilization corresponds to adding while loops.

It is not difficult to see that every partial recursive function can be described by a program in the RAM assembly language of Section 3.4.3. For example, to compute the zero function, $Z(x)$, it suffices for a RAM program to clear register R_1. To compute the successor function, $S(x)$, it suffices to increment register R_1. Similarly, to compute the projection function U_j^n, one need only load register R_1 with the contents of register R_j. Function composition it is straightforward: one need only insure that the functions f_j, $1 \leq j \leq m$, deposit their values in registers that are accessed by g. Similar constructions are possible for primitive recursion and minimalization. (See Problems 5.29, 5.30, and 5.31.)

Problems

THE STANDARD TURING MACHINE MODEL

5.1 Show that the standard Turing machine model of Section 5.1 and the model of Section 3.7 are equivalent in that one can simulate the other.

PROGRAMMING THE TURING MACHINE

5.2 Describe a Turing machine that generates the binary strings in lexicographical order. The first few strings in this ordering are 0, 1, 00, 01, 10, 11, 000, 001, ...

5.3 Describe a Turing machine recognizing $\{x^i y^j x^k \mid i, j, k \geq 1 \text{ and } k = i \cdot j\}$.

5.4 Describe a Turing machine that computes the function whose value on input $a^i b^j$ is c^k, where $k = i \cdot j$.

5.5 Describe a Turing machine that accepts the string $(\boldsymbol{u}, \boldsymbol{v})$ if $\boldsymbol{u}$ is a substring of $\boldsymbol{v}$.

5.6 The **element distinctness language**, L_{ed}, consists of binary strings no two of which are the same; that is, $L_{\text{ed}} = \{2\boldsymbol{w}_1 2 \dots 2\boldsymbol{w}_k 2 \mid \boldsymbol{w}_i \in \mathcal{B}^* \text{ and } \boldsymbol{w}_i \neq \boldsymbol{w}_j, \text{ for } i \neq j\}$. Describe a Turing machine that accepts this language.

EXTENSIONS TO THE STANDARD TURING MACHINE MODEL

5.7 Given a Turing machine with a double-ended tape, show how it can be simulated by one with a single-ended tape.

5.8 Show equivalence between the standard Turing machine and the one-tape **double-headed** Turing machine with two heads that can move independently on its one tape.

5.9 Show that a pushdown automaton with two pushdown tapes is equivalent to a Turing machine.

5.10 Figure 5.14 shows a representation of a Turing machine with a two-dimensional tape whose head can move one step vertically or horizontally. Give a complete definition of a two-dimensional TM and sketch a proof that it can be simulated by a standard TM.