CHAPTER 1

Computability

Introduction

In the last few decades substantial effort has been made to define a *computing device* that will be general enough to compute every "computable" function. In 1936 Turing suggested the use of a machine, since then called a *Turing machine*, which is considered to be the *most general computing device*. It was hypothesized by Church [Church's thesis (1936)] that any computable function can be computed by a Turing machine. Since our notion of computable function is not mathematically precise, we can never hope to prove Church's thesis formally. However, from the definition of a Turing machine, it is apparent that any computation that can be described by means of a Turing machine can be mechanically carried out; and conversely, any computation that can be performed on a modern-day digital computer can be described by means of a Turing machine. Moreover, all other general computing devices that have been proposed [e.g., by Church (1936), Kleene (1936), and Post (1936)]† have been shown to have the same computing capability as Turing machines, which strengthens our belief in Church's thesis.

In this chapter we discuss several machines, such as Post machines and finite machines with pushdown stores, which have the same computing capability as Turing machines. We emphasize that there are actually three different ways to look at a Turing machine: (1) as an *acceptor* (accepts a recursive or a recursively enumerable set), (2) as a *generator* (computes a total recursive or a partial recursive function), and (3) as an *algorithm* (solves or partially solves a class of yes/no problems).

We devote most of this chapter to discussing the class of Turing machines (the "most general possible" computing device); however, for purposes of comparison, in the first section we discuss the "simplest possible" computing device, namely, the *finite automaton*.

† All three papers appear in the collection by Davis (1965).

1-1 FINITE AUTOMATA

Let Σ be any finite set of symbols, called an *alphabet*; the symbols are called the *letters* of the alphabet. A *word* (*string*) *over* Σ is any finite sequence of letters from Σ. The *empty word*, denoted by Λ, is the word consisting of no letters. Σ^* denotes the set of all words over Σ, including the empty word Λ. Thus, if $\Sigma = \{a, b\}$, then $\Sigma^* = \{\Lambda, a, b, aa, ab, ba, bb, aaa, \ldots\}$. ϕ denotes the empty set $\{\ \}$, that is, the set consisting of no words.†

We define the *product* (*concatenation*) UV of two subsets U, V of Σ^* by

$$UV = \{x | x = uv, u \text{ is in } U \text{ and } v \text{ is in } V\}$$

That is, each word in the set UV is formed by concatenating a word in U with a word in V. As an example, if $U = \{a, ab, aab\}$ and $V = \{b, bb\}$, then the set UV is $\{ab, abb, abbb, aabb, aabbb\}$. Note that the word abb is obtained in two ways: (1) as a concatenated with bb, and (2) as ab concatenated with b. The set VU is $\{ba, bab, baab, bba, bbab, bbaab\}$. Note that $UV \neq VU$, which shows that the product operation is not commutative. However, the product operation is associative, i.e., for any subsets U, V, and W of Σ^*, $(UV)W = U(VW)$.

The *closure* (*star*) of a set S, denoted by S^*, is the set consisting of the empty word and all words formed by concatenating a finite number of words in S. Thus if $S = \{ab, bb\}$, then

$$S^* = \{\Lambda, ab, bb, abab, abbb, bbab, bbbb, ababab, \ldots\}.$$

An alternative definition is

$$S^* = S^0 \cup S^1 \cup S^2 \cup S^3 \cup \ldots,$$

where $S^0 = \{\Lambda\}$, and $S^i = S^{i-1}S$, for $i > 0$.

We shall now discuss a special class of sets of words over Σ, called *regular sets*. The class of regular sets over Σ is defined recursively as follows:

1. Every *finite* set of words over Σ (including ϕ, the empty set) is a regular set.
2. If U and V are regular sets over Σ, so are their union $U \cup V$ and product UV.
3. If S is a regular set over Σ, so is its closure S^*.

This means that no set is regular unless it can be obtained by a finite number

† Note the difference between Λ, $\{\ \}$, and $\{\Lambda\}$: Λ is a *word* (the empty word); $\{\ \}$ is a *set of words* (the set consisting of no words); and $\{\Lambda\}$ is a *set of words* (the set consisting of a single word, the empty word Λ).

of applications of 1 to 3. In other words, the class of regular sets over Σ is the smallest class containing all finite sets of words over Σ and closed under union, product, and star.

Let $\Sigma = \{a, b\}$. For example, we shall show that the set of all words over Σ containing either two consecutive a's or two consecutive b's, and the set of all words over Σ containing an even number of a's and an even number of b's, are regular sets over Σ. On the other hand, we shall show that the set $\{a^n b^n | n \geqq 0\}$ (that is, the set of all words which consists of n a's followed by n b's for any $n \geqq 0$) is not a regular set over Σ. Similarly, it can be shown that the sets $\{a^n b a^n | n \geqq 0\}$ and $\{a^{n^2} | n \geqq 0\}$ are not regular sets over Σ.†

In this section we describe three different ways of expressing regular sets. A subset of Σ^* is regular *if and only if*: (1) it can be expressed by some *regular expression*, (2) it is accepted by some *finite automaton*, or (3) it is accepted by some *transition graph*. Finally we prove that there is an algorithm to determine whether or not two given regular sets over Σ (expressed by any one of the above forms) are equal, which is one of the most important results concerning regular sets.

1-1.1 Regular Expressions

We consider first the representation of regular sets by regular expressions. This representation is of special interest since it allows algebraic manipulations with regular sets. The class of *regular expressions over* Σ is defined recursively as follows:

1. Λ and ϕ are regular expressions over Σ.
2. Every letter $\sigma \in \Sigma$ is a regular expression over Σ.
3. If R_1 and R_2 are regular expressions over Σ, so are $(R_1 + R_2)$, $(R_1 \cdot R_2)$, and $(R_1)^*$.

For example, if $\Sigma = \{a, b\}$, then $(((a + (b \cdot a)))^* \cdot a)$ is a regular expression over Σ.

Every regular expression R over Σ describes a set $\tilde{R}$ of words over Σ, that is, $\tilde{R} \subseteq \Sigma^*$, defined recursively as follows:

1. If $R = \Lambda$, then $\tilde{R} = \{\Lambda\}$, that is, the set consisting of the empty word Λ; if $R = \phi$, then $\tilde{R} = \phi$, that is the empty set.

† Note the difference between a *word*, for example, *bbaba*, a *set of words*, for example, $\{a^n b^n | n \geqq 0\}$, and a *class of sets of words*, for example, $\{\{a^n | n \geqq 0\}, \{a^n b^n | n \geqq 0\}, \{a^n b^n a^n | n \geqq 0\}, \{a^n b^n a^n b^n | n \geqq 0\}, \ldots\}$

2. If $R = \sigma$, then $\tilde{R} = \{\sigma\}$, that is, the set consisting of the letter σ.
3. Let R_1 and R_2 be regular expressions over Σ which describe the set of words $\tilde{R}_1$ and $\tilde{R}_2$, respectively:

   If $R = (R_1 + R_2)$, then $\tilde{R} = \tilde{R}_1 \cup \tilde{R}_2 = \{x | x \in \tilde{R}_1 \text{ or } x \in \tilde{R}_2\}$, that is, set union.
   If $R = (R_1 \cdot R_2)$, then $\tilde{R} = \tilde{R}_1\tilde{R}_2 = \{xy | x \in \tilde{R}_1 \text{ and } y \in \tilde{R}_2\}$, that is set product.
   If $R = (R_1)^*$, then $\tilde{R} = \tilde{R}_1^* = \{\Lambda\} \cup \{x | x \text{ obtained by concatenating a finite number of words in } \tilde{R}_1\}$, that is, set closure.

Parentheses may be omitted from a regular expression when their omission can cause no confusion. There are several rules for the restoration of omitted parentheses: '$*$' is more binding than '$\cdot$' or '$+$', that is, '$*$' is always attached to the smallest possible scope (Λ, ϕ, a letter, or a parenthesized expression), and '$\cdot$' is more binding than '$+$'. The '$\cdot$' is always omitted. For example, $a + ba^*$ stands for $(a + (b \cdot (a)^*))$, and $(a + ba)^*a$ stands for $(((a + (b \cdot a)))^* \cdot a)$. Whenever there are two or more consecutive '$\cdot$', the parentheses are associated to the left, and the same is true for '$+$'. For example, aba stands for $((a \cdot b) \cdot a)$, and $a^*(aa + bb)^*b$ stands for $(((a)^* \cdot (((a \cdot a) + (b \cdot b)))^*) \cdot b)$.

EXAMPLE 1-1

Consider the following regular expressions over $\Sigma = \{a, b\}$.

R	$\tilde{R}$
ba^*	All words over Σ beginning with a b followed only by a's.
$a^*ba^*ba^*$	All words over Σ containing exactly two b's.
$(a + b)^*$	All words over Σ.
$(a + b)^*(aa + bb)(a + b)^*$	All words over Σ containing two consecutive a's or two consecutive b's.

$[aa + bb + (ab + ba)(aa + bb)^*(ab + ba)]^*$	All words over Σ containing an even number of a's and an even number of b's.
$(b + abb)^*$	All words over Σ in which every a is immediately followed by at least two b's.

□

From the definition of regular expressions, it is straightforward that *a set is regular over Σ if and only if it can be expressed by a regular expression over Σ*. Note that a regular set may be described by more than one regular expression. For example, the set of all words over $\Sigma = \{a, b\}$ with alternating a's and b's (starting and ending with b) can be described by the expression $b(ab)^*$ as well as by $(ba)^*b$. *Two regular expressions R_1 and R_2 over Σ are said to be equivalent* (*notation*: $R_1 = R_2$) *if and only if* $\tilde{R}_1 = \tilde{R}_2$; thus, $b(ab)^*$ and $(ba)^*b$ are equivalent. Later in this section we shall describe an algorithm to determine whether or not two given regular expressions are equivalent. In certain cases, the equivalence of two regular expressions can be shown by the use of known identities. Some of the more interesting identities are listed below. For simplicity, we let $R_1 \subseteq R_2$ stand for $\tilde{R}_1 \subseteq \tilde{R}_2$ and $w \in R$ stand for $w \in \tilde{R}$.

For any regular expressions R, S, and T over Σ:

1. $R + S = S + R$, $\quad R + \phi = \phi + R$, $\quad R + R = R$, $\quad (R + S) + T = R + (S + T)$.
2. $R\Lambda = \Lambda R = R$, $\quad R\phi = \phi R = \phi$, $\quad (RS)T = R(ST)$
   Note that generally $RS \neq SR$.
3. $R(S + T) = RS + RT$, $\quad (S + T)R = SR + TR$.
4. $R^* = R^*R^* = (R^*)^* = (\Lambda + R)^*$, $\quad \phi^* = \Lambda^* = \Lambda$.
5. $R^* = \Lambda + R + R^2 + \ldots + R^k + R^{k+1}R^*$ $(k \geq 0)$
   special case: $R^* = \Lambda + RR^*$.
6. $(R + S)^* = (R^* + S^*)^* = (R^*S^*)^* = (R^*S)^*R^* = R^*(SR^*)^*$
   Note that generally $(R + S)^* \neq R^* + S^*$.
7. $R^*R = RR^*$, $\quad R(SR)^* = (RS)^*R$.
8. $(R^*S)^* = \Lambda + (R + S)^*S$, $\quad (RS^*)^* = \Lambda + R(R + S)^*$.

9. (*Arden rule*) Suppose $\Lambda \notin S$; then†

$$R = SR + T \quad \text{if and only if} \quad R = S^*T$$

$$R = RS + T \quad \text{if and only if} \quad R = TS^*$$

Most of these identities can be proved by a general technique which is sometimes called *proof by reparsing*. Let us illustrate the technique by proving that $R(SR)^* = (RS)^*R$ (identity 7). Consider any word $w \in R(SR)^*$ that is $w = r_0(s_1r_1)(s_2r_2) \ldots (s_nr_n)$ for some $n \geq 0$, where each $r_i \in R$ and each $s_i \in S$. By reparsing the last expression (using the fact that concatenation is associative), we establish that $w = (r_0s_1)(r_1s_2) \ldots (r_{n-1}s_n)r_n$; therefore, $w \in (RS)^*R$. Since w is arbitrarily selected, this implies that $R(SR)^* \subseteq (RS)^*R$; similarly, we show $R(SR)^* \supseteq (RS)^*R$, and this establishes the identity.

Another common method for proving such identities is simply to use previously known identities. For example, $R = S^*T$ *implies* $R = SR + T$ (identity 9) since $R = S^*T \overset{(5)}{=} (\Lambda + SS^*)T \overset{(3)}{=} \Lambda T + SS^*T \overset{(2)}{=} T + SR \overset{(1)}{=} SR + T$.

Finally let us prove that *if* $\Lambda \notin S$ *and* $R = SR + T$, *then* $R = S^*T$. We note first that if $R = SR + T$ and we repeatedly replace R by $SR + T$, we obtain (after using identities 2 and 3 above)

$$R = SR + T = S^2R + (ST + T) = S^3R + (S^2T + ST + T) = \cdots$$

$$= S^{k+1}R + (S^kT + S^{k-1}T + \cdots + ST + T) \qquad \text{for } k \geq 0$$

First we show that $R \subseteq S^*T$. Let $x \in R$ and suppose $|x| = l$.‡ Then we consider the equality $R = S^{l+1}R + (S^lT + S^{l-1}T + \cdots + ST + T)$. Since $\Lambda \notin S$, every word in $S^{l+1}R$ must have at least $l + 1$ symbols; therefore $x \notin S^{l+1}R$. But since $x \in R$, it follows that $x \in (S^lT + S^{l-1}T + \ldots + ST + T)$; therefore, $x \in S^*T$. To show that $S^*T \subseteq R$, we let $x \in S^*T$. Then there must exist $i \geq 0$ such that $x \in S^iT$. However, the equality $R = S^{i+1}R + (S^iT + S^{i-1}T + \ldots + ST + T)$ implies that $R \supseteq S^iT$, therefore, $x \in R$.

† The condition $\Lambda \notin S$ is required in both cases only to show the implication from left to right.

‡ For any word x, $|x|$ indicates the number of symbols in x; in particular, $|\Lambda| = 0$. $|x|$ is sometimes called *the length of* x.

We shall now demonstrate the use of the above identities for proving equivalence of regular expressions.

EXAMPLE 1-2

We prove the following:

$$(b + aa^*b) + (b + aa^*b)(a + ba^*b)^*(a + ba^*b) = a^*b(a + ba^*b)^*$$

$$\begin{aligned}
&(b + aa^*b) + (b + aa^*b)(a + ba^*b)^*(a + ba^*b) \\
&\overset{(3)}{=} (b + aa^*b)[\Lambda + (a + ba^*b)^*(a + ba^*b)] \\
&\overset{(5)}{=} (b + aa^*b)(a + ba^*b)^* \\
&\overset{(3)}{=} (\Lambda + aa^*)b(a + ba^*b)^* \\
&\overset{(5)}{=} a^*b(a + ba^*b)^*
\end{aligned}$$

□

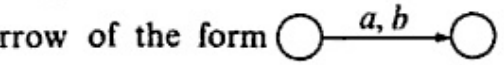
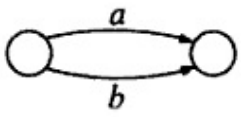
1-1.2 Finite Automata

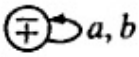
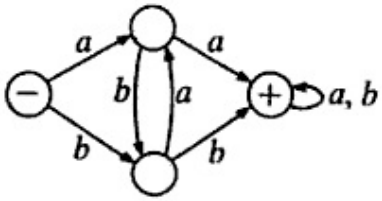
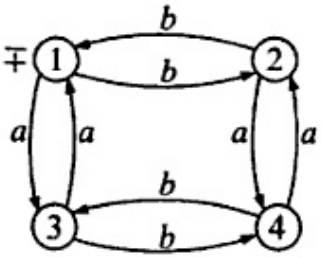
A *finite automaton A over* Σ, where $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, is a finite directed graph† in which every vertex has n arrows leading out from it, with each arrow labeled by a distinct σ_i $(1 \leqq i \leqq n)$. There is one vertex, labeled by a '$-$' sign, called the *initial vertex*, and a (possibly empty) set of vertices, labeled by a '$+$' sign, called the *final vertices* (the initial vertex may also be a final vertex). The vertices are sometimes called *states.*

For a word $w \in \Sigma^*$, a w *path* from vertex i to vertex j in A is a path from i to j such that the concatenation of the labels along this path form the word w. (Note that the path may intersect the same vertex more than once.) A word $w \in \Sigma^*$ is said to be *accepted* by the finite automaton A if the w path from the initial vertex leads to a final one. The empty word Λ is accepted by A if and only if the initial vertex is also final. The set of words accepted by a finite automaton A is denoted by $\tilde{A}$. Kleene's theorem (introduced in Sec. 1-1.4) implies that *a set is regular over* Σ *if and only if it is accepted by some finite automaton over* Σ.

† A *finite directed graph* consists of a finite set of elements (called *vertices*) and a finite set of ordered pairs (v, v') of vertices (called *arrows*). An arrow (v, v') is expressed as $\textcircled{v} \rightarrow \textcircled{v'}$. A finite sequence of (not necessarily distinct) vertices $v_1, v_2, \ldots, v_k$ is said to be a *path from* v_1 *to* v_k if each ordered pair (v_i, v_{i+1}), $1 \leqq i < k$, is an arrow of the graph.

EXAMPLE 1-3

Consider the following finite automata over $\Sigma = \{a, b\}$. We use the notation that an arrow of the form  stands for the two arrows 

A	$\tilde{A}$
	ϕ
	Σ^*
	All words over Σ containing two consecutive a's or two consecutive b's.
	All words over Σ containing an even number of a's and an even number of b's.

□

How do we know that the above $\tilde{A}$'s are correct? One possible way for giving an informal proof of $\tilde{A}$ is by associating with each vertex i a set S_i of all words w for which the w path from the initial vertex leads to i. Our choice of S_i's is validated by comparing each S_i with the S_j's of all vertices j leading by a single arrow to i. The union of all the S_i's for the final vertices is $\tilde{A}$. For example, the appropriate set of S_i's for the last automaton in Example 1-3 is

S_1: All words with an even number of a's and an even number of b's
S_2: All words with an even number of a's and an odd number of b's
S_3: All words with an odd number of a's and an even number of b's
S_4: All words with an odd number of a's and an odd number of b's

We can use the fact that every regular set is acceptable by some finite automaton to prove that certain sets are not regular. For example, we show that $\{a^n b^n | n \geqq 0\}$ *is not a regular set over* $\Sigma = \{a, b\}$. For suppose it is

a regular set; then there must exist a finite automaton A such that $\tilde{A} = \{a^n b^n | n \geqq 0\}$. Let us assume that A has N states, $N > 0$. Consider the word $a^N b^N$. Since the word is accepted by A and its length is greater than the number of vertices in A, it follows that there must exist in A at least one loop of a's or at least one loop of b's†, which enables A to accept the word $a^N b^N$. Suppose it is a loop of a's of length k, where $k > 0$. Then since we can go through the loop i times for any $i \geqq 0$, it follows that A must also accept all words of the form $a^{N+ik} b^N$ (for $i \geqq 0$), which contradicts the fact that $\tilde{A} = \{a^n b^n | n \geqq 0\}$.

1-1.3 Transition Graphs

We shall now introduce a generalization of the notion of finite automata, called *transition graphs*, by which it is much more convenient to express regular sets. We show, however, that although the class of finite automata is a proper subclass of the class of transition graphs, every regular set that is accepted by a transition graph is also accepted by some finite automaton.

A *transition graph* T *over* Σ is a finite directed graph in which every arrow is labeled by some word $w \in \Sigma^*$ (possibly the empty word Λ). There is at least one vertex, labeled by a $'-'$ sign (such vertices are called *initial vertices*), and a (possibly empty) set of vertices, labeled by a $'+'$ sign (called the *final vertices*). A vertex can be both initial and final.

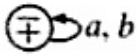
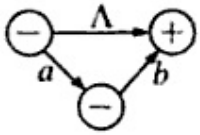
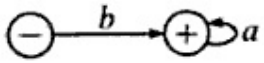
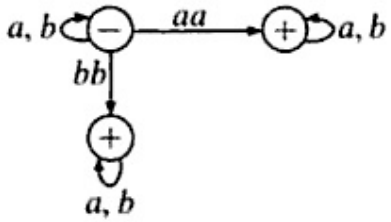
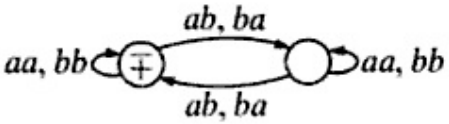
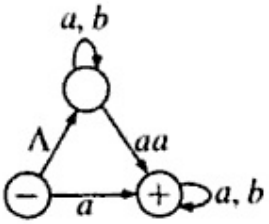
For a word $w \in \Sigma^*$, a w *path* from vertex i to vertex j in T is a finite path from i to j such that the concatenation of the labels on the arrows along this path form the word w (ignoring the Λ's, if any). A word $w \in \Sigma^*$ is said to be *accepted* by a transition graph T if there exists a w path from an initial vertex to a final one. The empty word Λ is accepted by T if there is a vertex in T which is both initial and final or if a Λ path leads from an initial vertex to a final one. The set of words accepted by a transition graph T is denoted by $\tilde{T}$. Kleene's theorem (introduced in Sec. 1-1.4) implies that *a set is regular over* Σ *if and only if it is accepted by some transition graph over* Σ.

Note that every finite automaton is a transition graph, but not vice versa. One of the key differences between the two is that a finite automaton is *deterministic* in the sense that for every word $w \in \Sigma^*$ and vertex i there is a unique w path leading from i. On the other hand, a transition graph is *nondeterministic* because it may have more than one w path leading from i (or none at all).

† A loop of σ's, where $\sigma \in \Sigma$, is a path in which the first vertex is identical to the last one and such that all its arrows are labeled by σ.

EXAMPLE 1-4

Consider the following transition graphs over $\Sigma = \{a, b\}$.

T	$\tilde{T}$
	ϕ
	$\{\Lambda\}$
	Σ^*
	$\{\Lambda, ab, b\}$
	All words over Σ beginning with a b followed only by a's
	All words over Σ containing two consecutive a's or two consecutive b's
	All words over Σ containing an even number of a's and an even number of b's
	All words over Σ either starting with a or containing aa

□

1-1.4 Kleene's Theorem

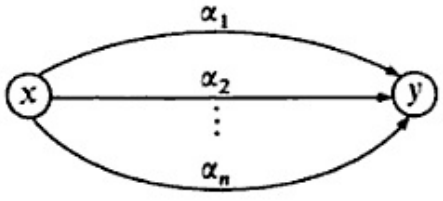
THEOREM 1-1 (Kleene). (1) *For every transition graph T over Σ there exists a regular expression R over Σ such that $\tilde{R} = \bar{T}$; (2) for every regular expression R over Σ there exists a finite automaton A over Σ such that $\bar{A} = \tilde{R}$.*

Because every finite automaton is a transition graph, it follows from the theorem that: (1) a set is regular over Σ if and only if it is accepted by some finite automaton over Σ, and (2) a set is regular over Σ if and only if it is accepted by some transition graph over Σ.

Proof (part 1). We describe briefly an algorithm for constructing for a given transition graph T a regular expression R such that $\tilde{R} = \bar{T}$. For the purpose of this proof, we introduce the notion of a *generalized transition graph*, which is a transition graph in which the arrows may be labeled by regular expressions rather than just by words.

First, add to the given transition graph T a new vertex, called x, and Λ arrows leading from x to all the initial vertices of T. Similarly, add another new vertex, called y, and Λ arrows leading from all the final vertices of T to y. Let x be the only initial vertex in the modified transition graph, and y the only final vertex.

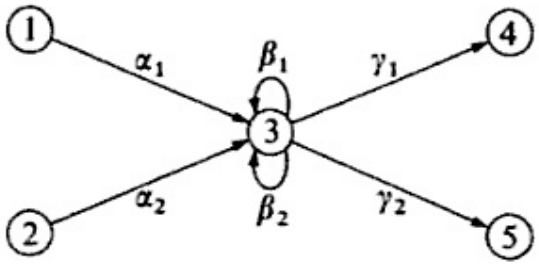
We now proceed step by step and eliminate all vertices of T until only the new vertices x and y are left. During the elimination process, the arrows and their labels are modified. The arrows may be labeled by regular expressions, i.e., we construct generalized transition graphs. The generalized transition graph which is constructed after each step still accepts the same set of words as the original T. The process terminates when we obtain a generalized transition graph with only two vertices, x and y, of the form:



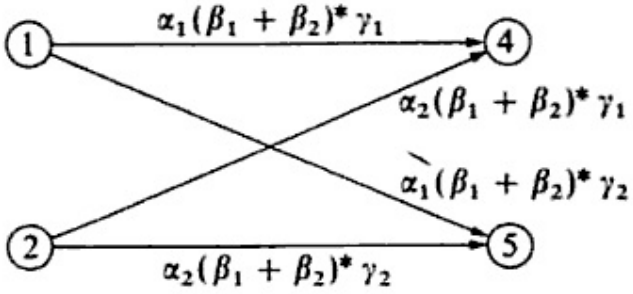
Then $\alpha_1 + \alpha_2 + \cdots + \alpha_n$ is the desired regular expression R.

The elimination process is straightforward. Suppose, for example, we want to eliminate vertex 3 in a generalized transition graph, where the

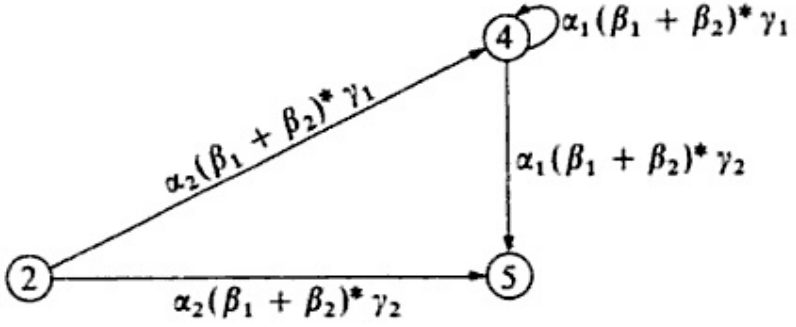
arrows leading to and from vertex 3 are of the form



($\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1$, and γ_2 are regular expressions.) To eliminate vertex 3 from the generalized transition graph, we replace that part by



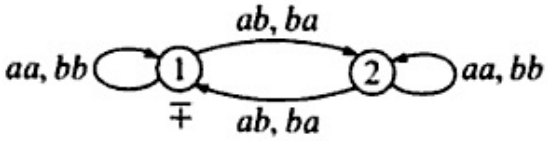
In the special case that vertices 1 and 4 are identical, we shall actually get



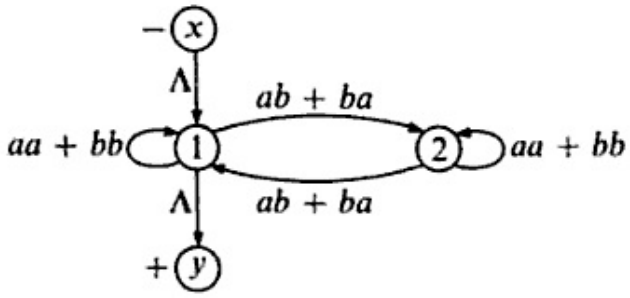
Let us illustrate the process before proceeding with the proof of part 2 of Theorem 1-1.

EXAMPLE 1-5

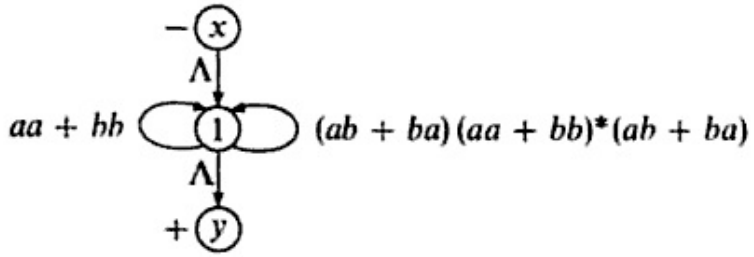
Consider the following transition graph T_0, which accepts the set of all words over $\Sigma = \{a, b\}$ with an even number of a's and an even number of b's.



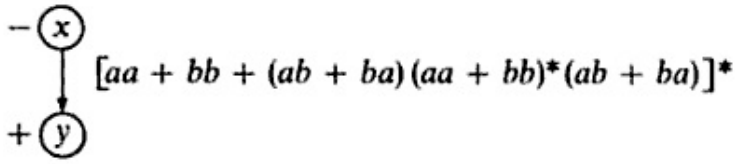
First we add the new vertices x and y. Since vertex 1 is both an initial and a final vertex of T_0, we add an Λ arrow leading from x to 1 and an Λ arrow leading from 1 to y. The corresponding generalized transition graph is



Eliminating vertex 2, we obtain



Finally, eliminating vertex 1, we obtain



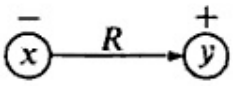
Thus the desired regular expression R_0 (that is, $\tilde{R}_0 = \tilde{T}_0$) is

$$[aa + bb + (ab + ba)(aa + bb)^*(ab + ba)]^*.$$

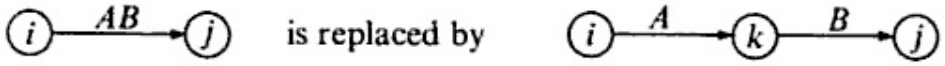
□

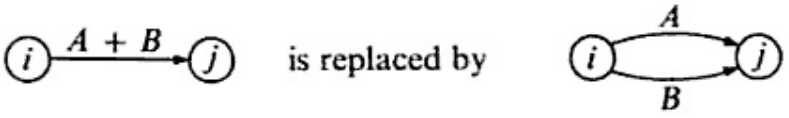
Proof (part 2) We now describe briefly an algorithm (known as the *subset method*) for constructing for a given regular expression R a finite automaton A such that $\tilde{A} = \tilde{R}$. We proceed in three steps.

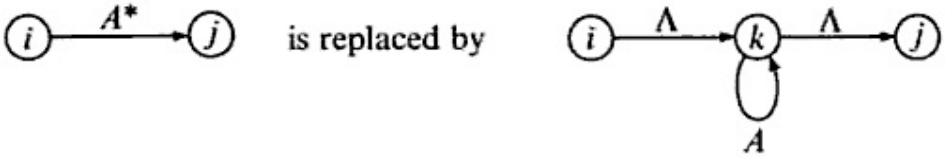
Step 1. First we construct a transition graph T such that $\tilde{T} = \tilde{R}$. We start with a generalized transition graph of the form



We successively split R by adding new vertices and arrows, until all arrows are labeled just by letters or Λ. The following rules are used:







Note that in the final transition graph, x is still the only initial vertex and y is the only final vertex. In practice, often an appropriate simple transition graph can be constructed in a straightforward way without using the above inductive steps.

Step 2. Let T be the transition graph generated in Step 1. Now we construct the *transition table* of T; for simplicity we assume that $\Sigma = \{a, b\}$.

Let M be any subset of vertices of T. For any word $w \in \Sigma^*$ we define M_w to be the subset of all vertices of T reachable by a w path from some vertex of M. For example, M_{ab} consists of all vertices of T reachable by an ab path from some vertex of M, or, equivalently, all vertices of T reachable by a b path from some vertex of M_a.

The transition table of T consists of three columns. The elements of the table are subsets of vertices of T (possibly the empty set). The subset in the upper-left-hand corner of the table (first row, column 1) is $\{x\}_\Lambda$, that is, the subset which consists of the initial vertex x and all vertices of T reachable by a Λ path from x. In general, for any row of the table, if the subset M is in column 1, then we add M_a (that is, the set of vertices of T reachable by an a path from some vertex of M) in column 2 and M_b (that is, the set of vertices of T reachable by a b path from some vertex of M) in column 3. If M_a does not occur previously in column 1, we place it in column 1 of the next row and repeat the process. We treat M_b similarly.

The process is terminated when there are no new subsets in columns 2 and 3 of the table that do not occur in column 1. The process must always terminate because all subsets in column 1 are distinct and there are only finitely many distinct subsets of the finite set of vertices of T.

Step 3. Finally we use the transition table to construct the desired finite automaton A. The finite automaton A is constructed as follows. For every subset M in column 1 of the table there corresponds a vertex $\overline{M}$ in A. From every vertex $\overline{M}$ in A there is an a arrow leading to vertex $\overline{M}_a$ and a b arrow leading to vertex $\overline{M}_b$. The vertex $\overline{\{x\}}_\Lambda$ (that is, the vertex corresponding to the subset in the upper-left-hand corner of the table) is the only initial vertex of A. A vertex $\overline{M}$ of A is final if and only if M contains a final vertex of T.

We leave it to the reader to verify that a word $w \in \Sigma^*$ is accepted by T if and only if it is accepted by A. The following example illustrates the method.

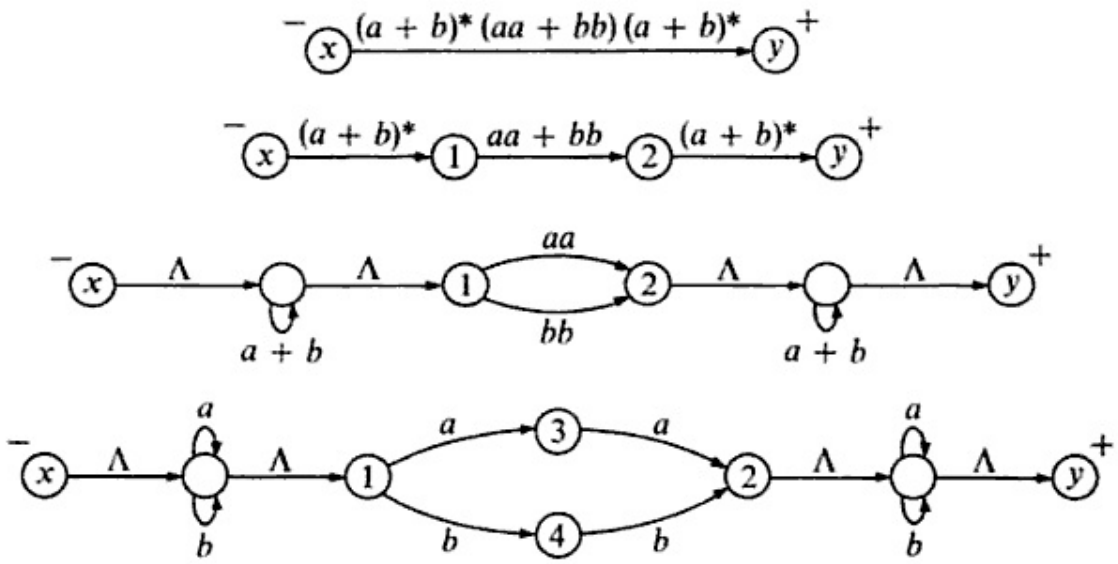
EXAMPLE 1-6

Consider the regular expression R

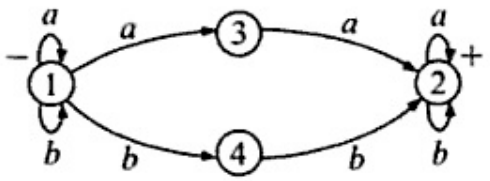
$$(a + b)^* (aa + bb) (a + b)^*$$

which describes the set of all words over $\Sigma = \{a, b\}$ containing either two consecutive a's or two consecutive b's. We proceed to construct a finite automaton A that will accept the same set of words.

Step 1. The corresponding transition graph T is constructed as follows:



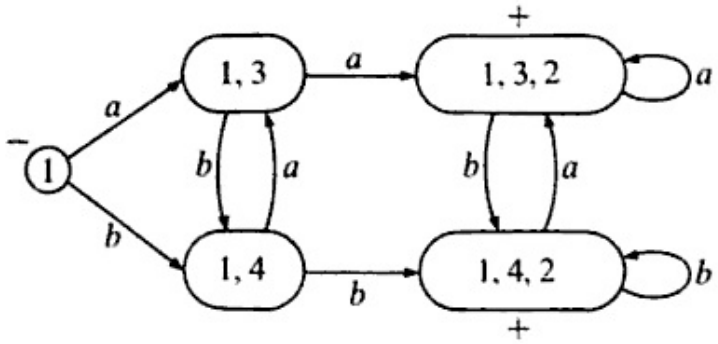
This is the desired transition graph. Clearly it can be simplified to



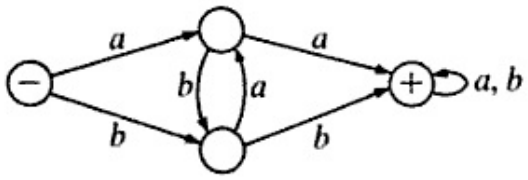
Step 2. The corresponding transition table is

M	M_a	M_b
$\{1\}$	$\{1, 3\}$	$\{1, 4\}$
$\{1, 3\}$	$\{1, 3, 2\}$	$\{1, 4\}$
$\{1, 4\}$	$\{1, 3\}$	$\{1, 4, 2\}$
$\{1, 3, 2\}$	$\{1, 3, 2\}$	$\{1, 4, 2\}$
$\{1, 4, 2\}$	$\{1, 3, 2\}$	$\{1, 4, 2\}$

Step 3. The desired finite automaton A is



Note that it is possible to simplify the finite automaton to obtain



□

1-1.5 The Equivalence Theorem

We shall conclude this section with an important theorem.

THEOREM 1-2 (Moore). *There is an algorithm to determine whether or not two finite automata A and A' over Σ are equivalent* (*that is, $\tilde{A} = \tilde{A}'$*).

Proof. Given two finite automata A and A' over Σ. Suppose for simplicity that $\Sigma = \{a, b\}$. First we rename the vertices of A and A' so that all vertices are distinct. Let x and x' be the initial vertices of A and A', respectively.

To decide whether or not A and A' are equivalent, we construct a *comparison table* which consists of three columns. The elements of the table are pairs of vertices (v, v'), where v is a vertex of A and v' is a vertex of A'. The pair in the upper-left-hand corner of the table (first row, column 1) is (x, x'). In general, for any row of the table, if the pair (v, v') is in column 1, then we add in column 2 the pair of vertices (v_a, v'_a), where the a arrow from v leads to v_a in A and the a arrow from v' leads to v'_a in A'. Similarly, in column 3 we add the pair of vertices (v_b, v'_b), where the b arrow from v leads to v_b in A and the b arrow from v' leads to v'_b in A'. If (v_a, v'_a) does not occur previously in column 1, we place it in column 1 of the next row and repeat the process; we treat (v_b, v'_b) similarly.

If we reach a pair (v, v') in the table for which v is a final vertex of A and v' is a nonfinal vertex of A' or vice versa, we stop the process: *A and A' are not equivalent.* Otherwise, the process is terminated when there are no new pairs in columns 2 and 3 of the table that do not occur in column 1.

In this case *A and A' are equivalent.* We leave it to the reader to verify these claims. The process must always terminate because all pairs in column 1 are distinct pairs and there are only finitely many distinct pairs of the vertices of A and A'.

Q.E.D.

EXAMPLE 1-7

1. Consider the following two finite automata A and A' over $\Sigma = \{a, b\}$ described in Fig. 1-1.

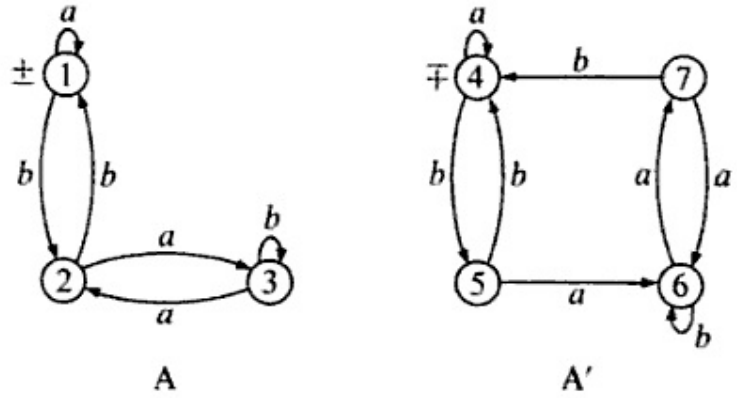


Figure 1-1 The finite automata A and A'.

They are equivalent because the corresponding comparison table is

(v, v')	(v_a, v'_a)	(v_b, v'_b)
(1, 4)	(1, 4)	(2, 5)
(2, 5)	(3, 6)	(1, 4)
(3, 6)	(2, 7)	(3, 6)
(2, 7)	(3, 6)	(1, 4)

There are no pairs in columns 2 and 3 that do not occur in column 1.

2. Consider the two finite automata B and B' described in Fig. 1-2.

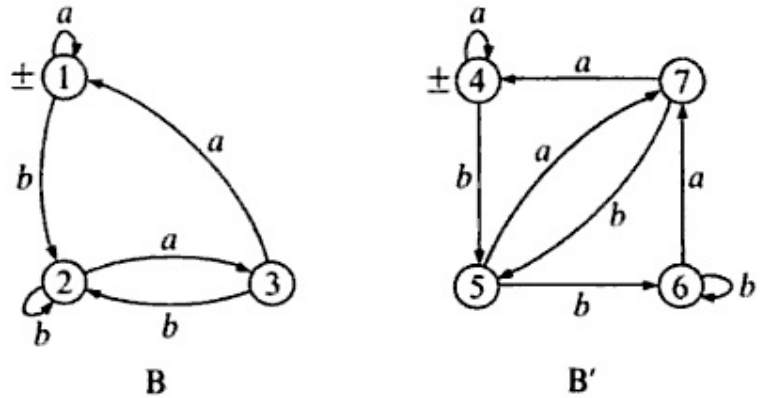


Figure 1-2 The finite automata B and B'.

They are equivalent because the corresponding comparison table is

(v, v')	(v_a, v'_a)	(v_b, v'_b)
(1, 4)	(1, 4)	(2, 5)
(2, 5)	(3, 7)	(2, 6)
(3, 7)	(1, 4)	(2, 5)
(2, 6)	(3, 7)	(2, 6)

Again, there are no pairs in columns 2 and 3 that do not occur in column 1.

3. Consider the two finite automata A and B' described in Fig. 1-3.

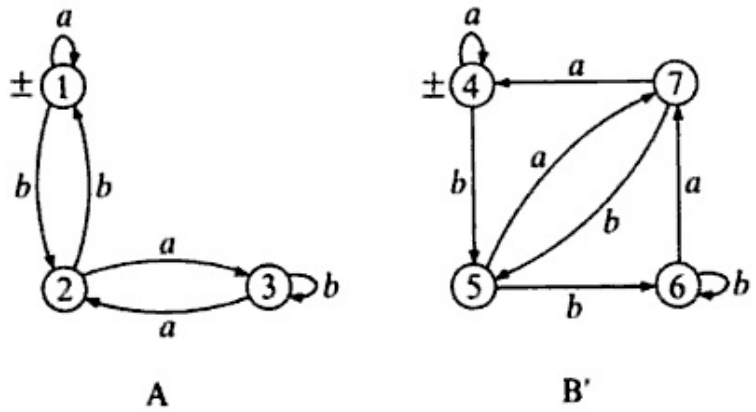


Figure 1-3 The finite automata A and B'.

They are *not* equivalent because the corresponding comparison table is

(v, v')	(v_a, v'_a)	(v_b, v'_b)
(1, 4)	(1, 4)	(2, 5)
(2, 5)	(3, 7)	(1, 6)

Note that 1 is a final vertex of A while 6 is a nonfinal vertex of B'. Since this pair was obtained by applying the letter b twice, the table actually shows us, not only that A and B' are not equivalent, but also that an appropriate counterexample is the word bb.

It can be shown that A accepts the set of all words which are the binary representation of natural numbers divisible by 3, where a stands for 0 and b stands for 1. (Such numbers with leading 0's on the left are also accepted.) The corresponding sets S_1, S_2, and S_3 contain all words representing binary numbers which, after division by 3, yield remainders 0, 1, and 2, respectively. The automaton B' accepts the set of all words which are the binary representation of natural numbers divisible by 4. The

corresponding sets S_4, S_5, S_6, and S_7 contain all words representing binary numbers which, after division by 4, yield remainders 0, 1, 3, and 2, respectively.

□

1-2 TURING MACHINES

In this section we define three classes of machines over a given alphabet Σ: *Turing machines*, *Post machines*, and *finite machines with pushdown stores*. Each of these machines can be applied to any word $w \in \Sigma^*$ as input. It can stop in two different ways—by accepting w or rejecting w—or it can loop forever. For such a machine M over Σ, we denote the set of all words over Σ accepted by M by $accept(M)$, the set of all words over Σ rejected by M by $reject(M)$, and the set of all words over Σ for which M loops by $loop(M)$. It is clear that for any given machine M over Σ, the three sets of words $accept(M)$, $reject(M)$, and $loop(M)$ are disjoint and that $accept(M) \cup reject(M) \cup loop(M) = \Sigma^*$.

Two such machines M_1 and M_2 over Σ are said to be *equivalent* if $accept(M_1) = accept(M_2)$ and, clearly, $reject(M_1) \cup loop(M_1) = reject(M_2) \cup loop(M_2)$. A class $\mathscr{M}_1$ of machines is said to have the *same power* as a class $\mathscr{M}_2$ of machines, $\mathscr{M}_1 = \mathscr{M}_2$, if for every machine in $\mathscr{M}_1$ there is an equivalent machine in $\mathscr{M}_2$, and vice versa. We also say that a class $\mathscr{M}_1$ of machines is *more powerful* than a class $\mathscr{M}_2$, $\mathscr{M}_1 > \mathscr{M}_2$, if for every machine in $\mathscr{M}_2$ there is an equivalent machine in $\mathscr{M}_1$, but not vice versa.

We shall show that the three classes of machines—Turing, Post and finite machines with two pushdown stores—have the same power. We finally emphasize that the introduction of a *nondeterministic mechanism* to these machines does not add anything to their power.

We make use of three basic functions defined over Σ^*:

$head(x)$:	gives the head (leftmost letter) of the word x
$tail(x)$:	gives the tail of the word x (that is, x with the leftmost letter removed)
$\sigma \cdot x$:	concatenates the letter σ and the word x.

For example, if $\Sigma = \{a, b\}$, then $head(abb) = a$, $tail(abb) = bb$, and $a \cdot bb = abb$. We agree that $head(\Lambda) = tail(\Lambda) = \Lambda$. Note that by definition $head(\sigma) = \sigma$, $tail(\sigma) = \Lambda$, and $\sigma \cdot \Lambda = \sigma$ for all $\sigma \in \Sigma$.

For simplicity, we present most of the results for the alphabet $\Sigma = \{a, b\}$; however, all the results hold as well for any alphabet with two or more letters.†

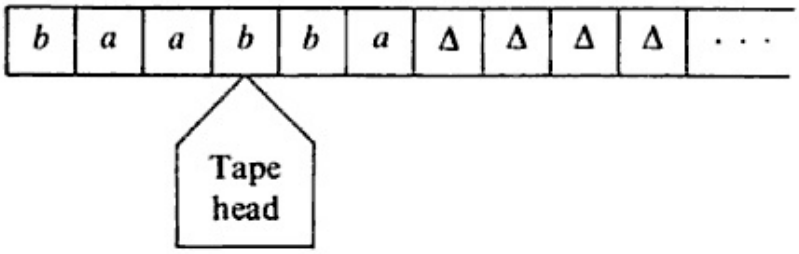
† In this chapter we never discuss the special case of an alphabet consisting of a single letter.

1-2.1 Turing Machines

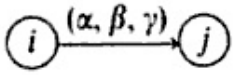
Specifications for Turing machines have been given in various ways in the literature. In our model, a *Turing machine M over an alphabet* Σ consists of three parts:

1. A *tape* which is divided into cells. The tape has a leftmost cell but is infinite to the right. Each cell of the tape holds exactly one tape symbol. The *tape symbols* include the letters of the given alphabet Σ, the letters of a finite auxiliary alphabet V, and the special blank symbol Δ. All symbols are distinct.

2. A *tape head* which scans one cell of the tape at a time. It can move; in each move the head prints a symbol on the tape cell scanned, replacing what was written there, and moves one cell left or right.



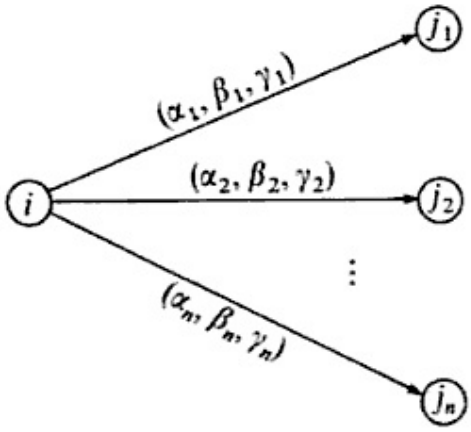
3. A *program* which is a finite directed graph (the vertices are called *states*). There is always one *start state*, denoted by START, and a (possibly empty) subset of *halt states*, denoted by HALT. Each arrow is of the form



where $\alpha \in \Sigma \cup V \cup \{\Delta\}$, $\beta \in \Sigma \cup V \cup \{\Delta\}$, and $\gamma \in \{L, R\}$. This indicates that, if during a computation we are in vertex (state) i and the tape head scans the symbol α, then we proceed to vertex (state) j while the tape head prints the symbol β in the scanned cell and then moves one cell left or right, depending whether γ is L or R. All arrows leading from the same vertex i must have distinct α's.

Initially, the given input word w over Σ is on the left-hand side of the tape, where the remaining infinity of cells hold the blank symbol Δ, and the tape head is scanning the leftmost cell. The execution proceeds as instructed by the program, beginning with the START state. If we eventually reach a HALT state, the computation is halted and we say that the

input word w is *accepted* by the Turing machine. If we reach a state i of the form



where the symbol scanned by the tape head is different from $\alpha_1, \alpha_2, \ldots,$ and α_n, there is no way to proceed with the computation. In this case therefore, the computation is halted and we say that the input word w is *rejected* by the Turing machine. The other case where the computation is halted and the input word is said to be *rejected* by the Turing machine is when the tape head scans the leftmost cell and is instructed to do a left move.

EXAMPLE 1-8

The Turing machine M_1 over $\Sigma = \{a, b\}$ (shown in Fig. 1-4) accepts every word of the form $a^n b^n$, $n \geqq 0$, and rejects all other words over Σ; that is, $accept(M_1) = \{a^n b^n | n \geqq 0\}$, $reject(M_1) = \Sigma^* - accept(M_1)$, and $loop(M_1) = \phi$. Here $V = \{A, B\}$. [Note that the specification of β and γ in (Δ, Δ, R) are irrelevant for the performance of this program.]

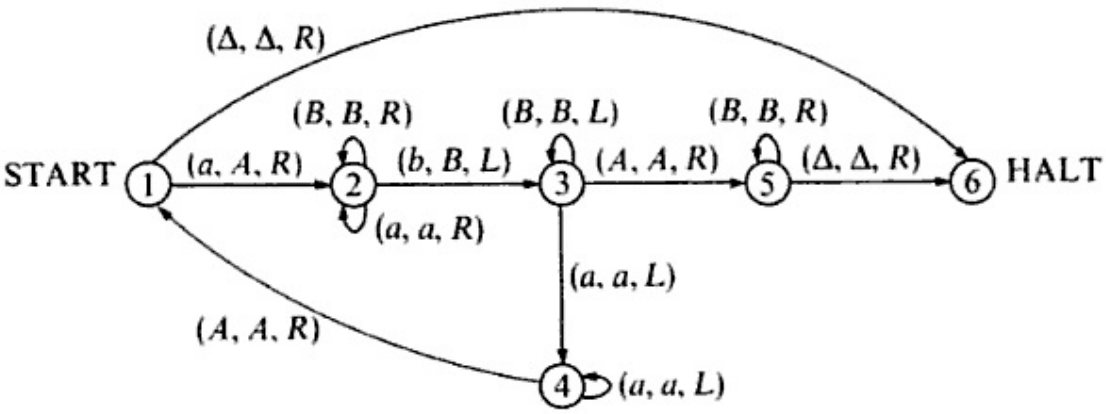
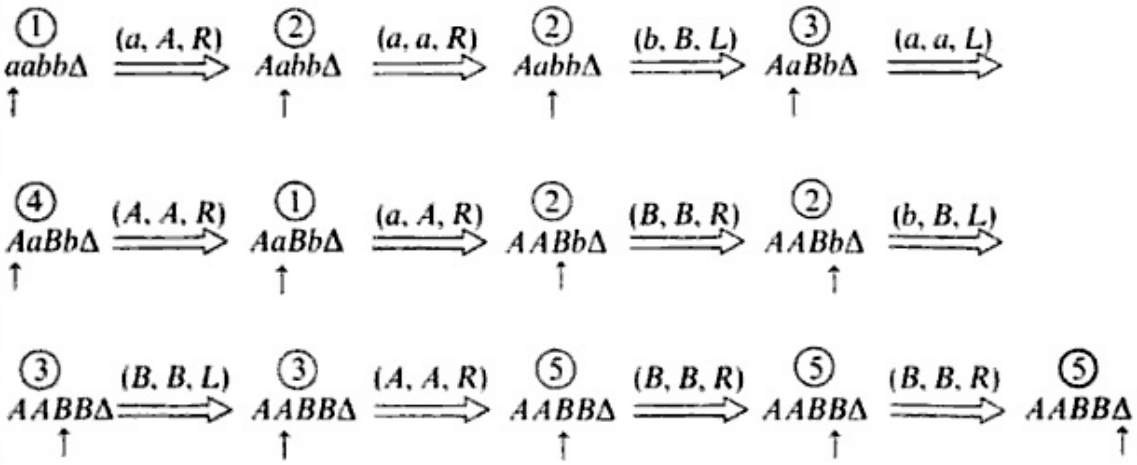


Figure 1-4 The Turing machine M_1 that accepts $\{a^n b^n | n \geqq 0\}$.

Starting from state 1, we move right. The leftmost occurrence of a is replaced by A, and then the leftmost occurrence of b is replaced by B; from state 3 we go back to the new leftmost occurrence of a and start again with state 1. This loop is repeated n times. When we reach state 5, all n a's have been replaced by A's and the leftmost n b's by B's. If the symbol to the right of the nth B is Δ (blank), the input word is accepted.

The following sequence illustrates how the Turing machine acts with input $aabb$:



Here, a string of the form $AaBb\Delta$ indicates the tape (ignoring the infinite sequence of blanks to the right), and the arrow indicates the current symbol scanned by the tape head. A labeled arrow such as $\overset{(a,A,R)}{\Longrightarrow}$ indicates which instruction of the program is performed, and a circled number such as ① indicates the current state.

Note that the above argument justifies only that every word of the form a^nb^n is acceptable by M_1. The other direction must be justified separately, that is, that every word acceptable by M_1 must be of the form a^nb^n.

□

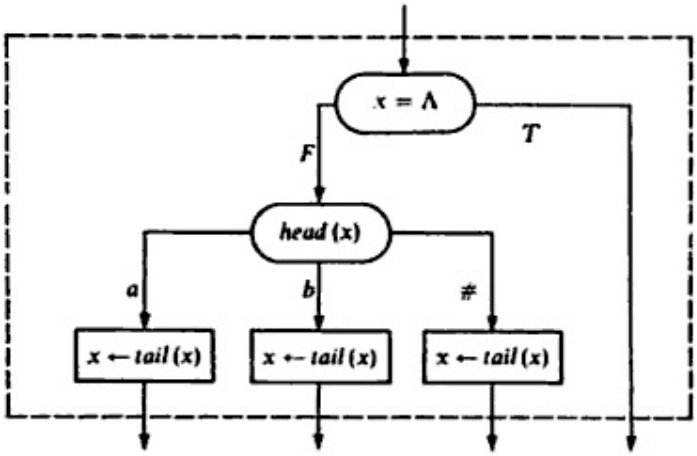
1-2.2 Post Machines

A *Post machine M over* $\Sigma = \{a, b\}$ is a *flow-diagram*† with one variable x, which may have as a value any word over $\{a, b, \#\}$, where $\#$ is a special auxiliary symbol. Each statement in the flow-diagram has one of the following forms:

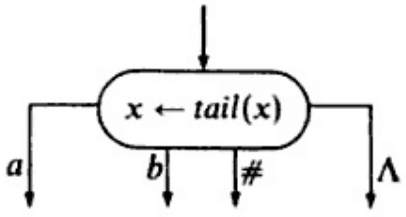
1. START statement (exactly one)

2. HALT statements

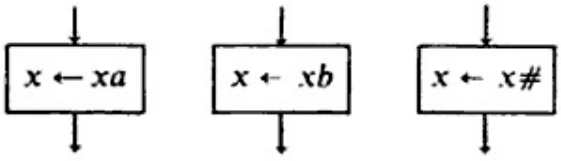
3. TEST statement



or in short‡



4. ASSIGNMENT statements



† A *flow-diagram* is a finite directed graph in which each vertex consists of a statement. We always allow the *JOIN statement* for combining two or more arrows into a single one.

‡Note that for a general n-letter alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, a TEST statement would have $n + 2$ exits labeled by $\sigma_1, \sigma_2, \ldots, \sigma_n, \#, \Lambda$.

Thus the TEST statement checks the leftmost letter of x, $head(x)$, and deletes it after making the decision. The only ASSIGNMENT statements allowed are to concatenate a letter (a, b, or $\#$) to the right of x.

A word w over Σ is said to be *accepted/rejected* by a Post machine M if the computation of M starting with input $x = w$ eventually reaches an ACCEPT/REJECT halt.

EXAMPLE 1-9

The Post machine M_2 over $\Sigma = \{a, b\}$ described in Fig. 1-5 accepts every word over Σ of the form $a^n b^n$, $n \geqq 0$, and rejects all other words over Σ; that is, $accept(M_2) = \{a^n b^n | n \geqq 0\}$, $reject(M_2) = \Sigma^* - accept(M_2)$, and $loop(M_2) = \phi$.

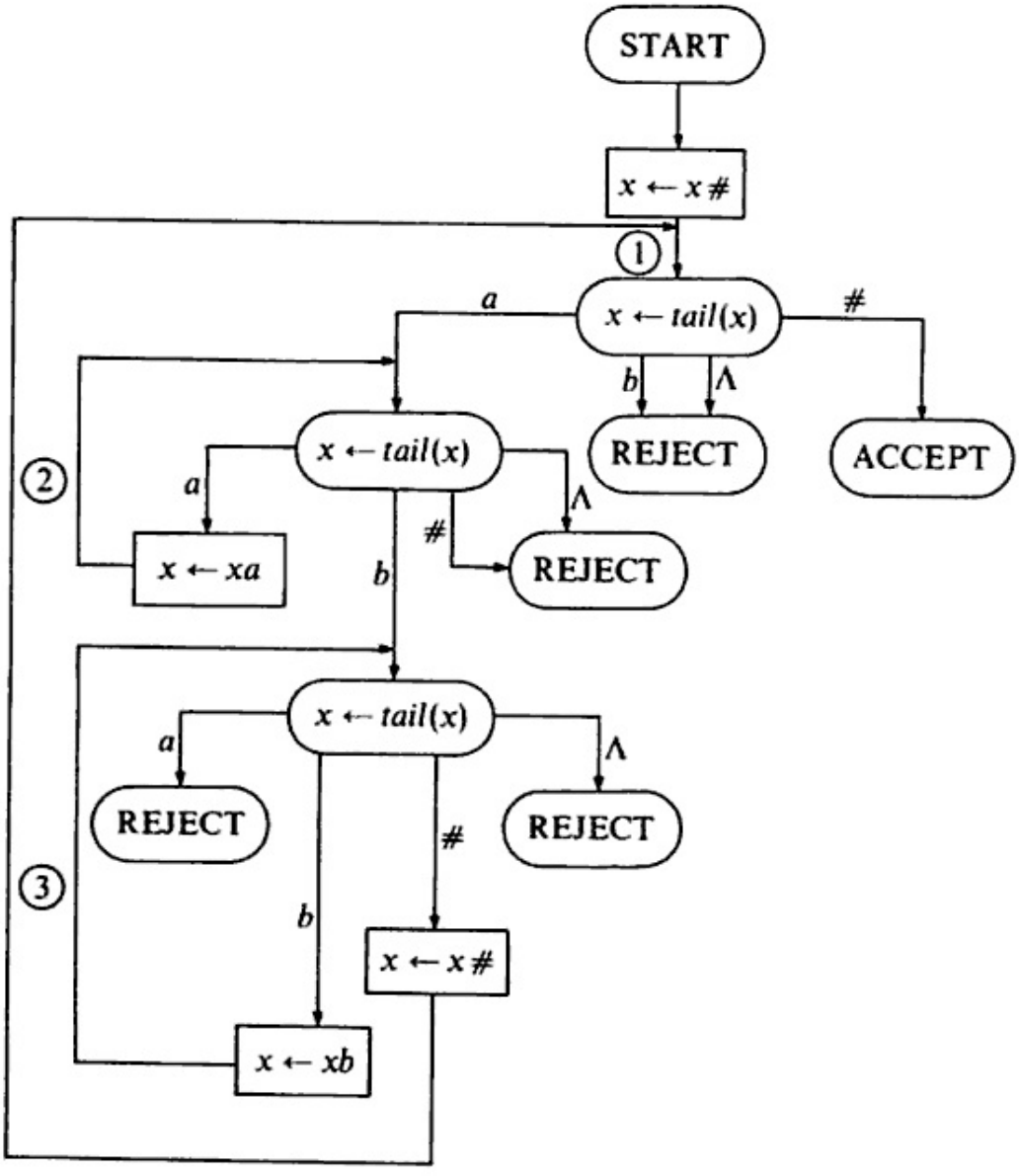


Figure 1-5 The Post Machine M_2 that accepts $\{a^n b^n | n \geqq 0\}$.

Suppose $x = a^n b^n$ for some $n \geqq 0$. The key point is that whenever we reach point 1, we have the word $a^i b^i \#$ in x, $0 \leqq i \leqq n$. Now if we find an a on the left of x, we go $i - 1$ times around loop 2 until we obtain $b^i \# a^{i-1}$; then we go $i - 1$ times around loop 3 until we obtain $\# a^{i-1} b^{i-1}$; finally we move the $\#$ symbol to obtain $a^{i-1} b^{i-1} \#$ and repeat the process from point 1. Note that there is only one ACCEPT statement, which corresponds to the case that we reach point 1 with only $\#$ in x (that is, $i = 0$).

□

EXAMPLE 1-10

Consider the Post machine M_3 over $\Sigma = \{a, b\}$ shown in Fig. 1-6. Here

$accept(M_3)$ = {all words over Σ with the same number of a's and b's}
$reject(M_3)$ = {all words over Σ where the difference between the number of a's and b's is 1}
$loop(M_3)$ = {all words over Σ where the difference between the number of a's and b's is more than 1}

For example, $ab \in accept(M_3)$, $aba \in reject(M_3)$, while $aaba \in loop(M_3)$.

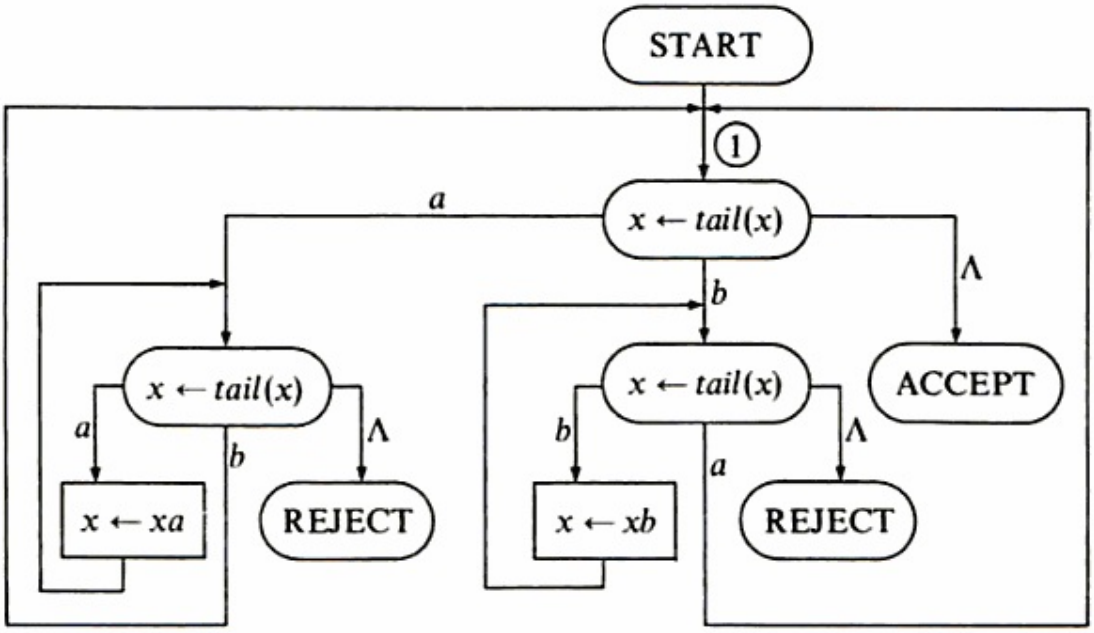


Figure 1-6 The Post machine M_3 that accepts all words with the same number of a's and b's.

Note that we have not used the special symbol $\#$, and therefore we ignored the $\#$ exit from the tests. The main test is at point 1: if $head(x) = a$, then we keep searching for a letter b to be eliminated, while if $head(x) = b$, then we keep searching for a letter a.

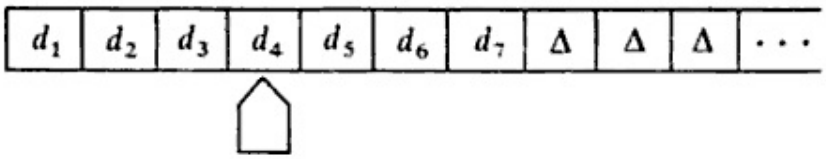
□

THEOREM 1-3 (Post) *The class of Post machines over* Σ *has the same power as the class of Turing machines over* Σ.

That is, for every Post machine over Σ there is an equivalent Turing machine over Σ and vice versa.

Proof. First we show that every Turing machine over $\Sigma = \{a, b\}$ can be simulated by an equivalent Post machine over $\Sigma = \{a, b\}$. The main idea behind this simulation is that the contents of the tape and the position of the tape head at any stage of the computation of the Turing machine are expressed as values of the variable x in the Post machine.

For example, if at some stage of the computation of the Turing machine the tape is of the form



where each $d_i \in \Sigma \cup V \cup \{\Delta\}$ and the tape head scans the symbol d_4, then this situation is expressed in the Post machine by

$$x = d_4 d_5 d_6 d_7 \# d_1 d_2 d_3$$

In other words, the infinite string of Δ's is ignored and the string $d_1 d_2 \ldots d_7$ is rotated in such a way that the leftmost symbol of x is the one scanned by the tape head of the Turing machine. The special symbol $\#$ is used to indicate the *breakpoint* of the string. Now, suppose we have $x = d_4 d_5 d_6 d_7 \# d_1 d_2 d_3$ and the next move of the Turing machine is (d_4, β, R); then the contents of x are changed by the Post machine to

$$x = d_5 d_6 d_7 \# d_1 d_2 d_3 \beta$$

If $x = d_4 d_5 d_6 d_7 \# d_1 d_2 d_3$ and the next move of the Turing machine is (d_4, β, L) then the contents of x are changed to

$$x = d_3 \beta d_5 d_6 d_7 \# d_1 d_2$$

However, there are two important special cases: The first case occurs when we have $x = d_7 \# d_1 d_2 \ldots d_6$ and the next move of the Turing ma-

chine is (d_7, β, R); then we have to change x by the Post machine to $x = \# d_1 d_2 \ldots d_6 \beta$ ($\#$ is the leftmost symbol of x). This means that the next symbol to be scanned by the tape head of the Turing machine is the first Δ (blank) to the right of d_7. Therefore in this case we change the value of x to $x = \Delta \# d_1 d_2 \ldots d_6 \beta$. The second case occurs when we reach a situation where $x = d_1 d_2 \ldots d_7 \#$ ($\#$ is the rightmost symbol of x) and the next move of the Turing machine is (d_1, β, L). This case happens when the tape head of the Turing machine scans the leftmost symbol of the tape and we are instructed to make a left move. Therefore, in this case we shall go to a *REJECT* halt in the Post machine.

Note that the Post machine obtained is over $\Sigma \cup V \cup \{\Delta\}$ rather than just over Σ. We can overcome this difficulty by using a standard encoding technique: If there are l symbols in $\Sigma \cup V \cup \{\Delta\}$, where $2^{n-1} < l \leqq 2^n$, then each one of these symbols is encoded as a word of length n over $\Sigma = \{a, b\}$. For example, if $\Sigma \cup V \cup \{\Delta\} = \{a', b', A, B, C, D, \Delta\}$, then each one of these symbols is encoded as a word of three a's and b's: $a' = aaa$, $b' = aab$, $A = aba$, $B = abb$, $C = baa$, $D = bab$, and $\Delta = bba$. Now, rather than looking, for example, for A as the leftmost letter of x, we shall look for the string aba on the left of x; similarly, rather than adding the letter A to the right of x, we shall add the string aba; and so forth.

The translation for any given Post machine over $\Sigma = \{a, b\}$ into an equivalent Turing machine over $\Sigma = \{a, b\}$ is much simpler. The current value of x during the computation of the Post machine, say, $x = d_1 d_2 d_3 d_4 \# d_5 d_6$, is expressed in the tape of the Turing machine as

$$\Delta\Delta \ldots \Delta d_1 d_2 d_3 d_4 \# d_5 d_6 \Delta\Delta \ldots$$

The two main operations of the Turing machine are either replacing d_1 by Δ (*deleting the leftmost symbol of* x) or replacing the Δ to the right of d_6 by a new letter (*adding a letter to the right of* x).†

Q.E.D.

† Note that the input word $w = \sigma_1 \sigma_2 \ldots \sigma_k$ must be stored initially in the tape of the Turing machine with an additional Δ to the left, that is, $\Delta\sigma_1\sigma_2 \ldots \sigma_k$. This is necessary in order to be able to reach σ_1 by left moves without encountering a *REJECT* halt.

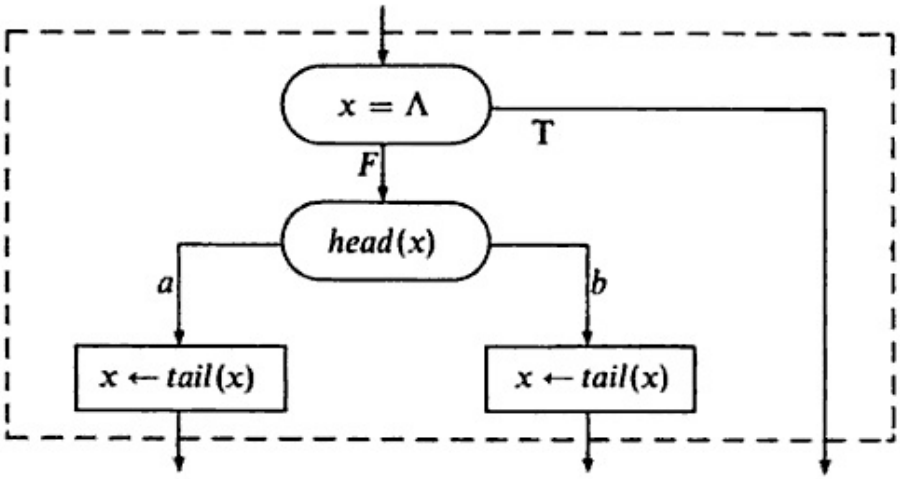
1-2.3 Finite Machines with Pushdown Stores

A *finite machine M over* $\Sigma = \{a, b\}$ is a flow-diagram with one variable x in which each statement is of one of the following forms. (The variable x may have as value any word over $\{a, b\}$.)

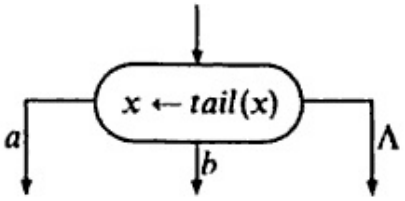
1. START statement (exactly one)

2. HALT statements

3. TEST statement

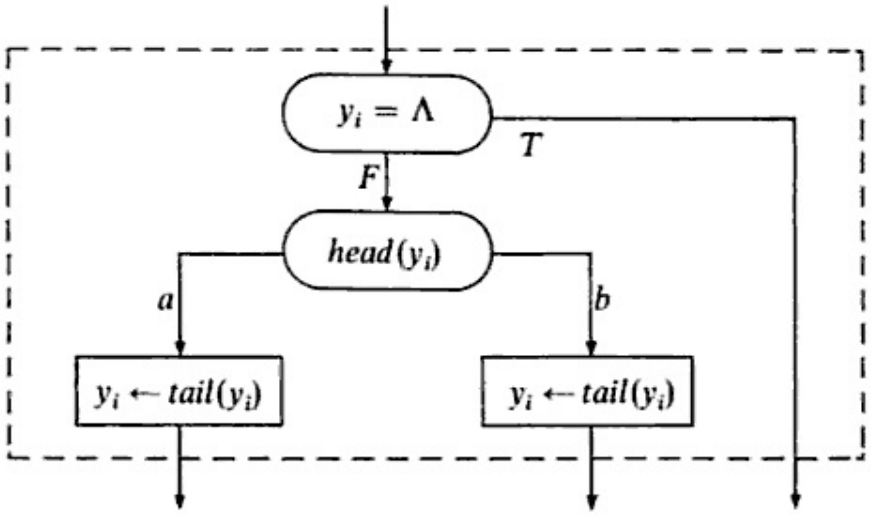


or for short

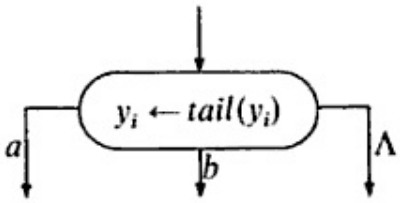


A variable y_i is said to be a *pushdown store over* $\Sigma = \{a, b\}$ if all the operations applied to y_i are of the form

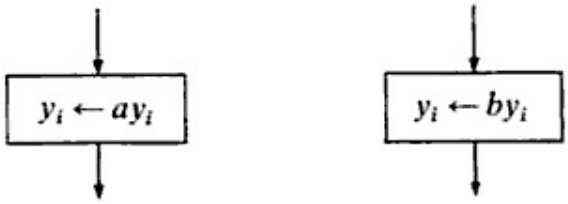
1. TEST y_i



or for short



2. ASSIGN TO y_i



Although the Post machines seem to be very "similar" to the finite machines with one pushdown store, there is a vast difference between the two. In a Post machine, not only may we read and erase letters on the left of x, but also we may write new letters on the *right* of x. Thus, in a Post machine, as the ends of x undergo modification, the information in x circulates slowly from right to left. In a pushdown store we may write only on the *left* of y_i. Thus, while a pushdown store manipulates the string in y_i in a "last-in–first-out" manner, a Post machine manipulates the string in x in a "first-in–first-out" manner.

A word w over Σ is said to be *accepted/rejected* by a finite machine M (with or without pushdown stores) over Σ if the computation of M starting with input $x = w$ eventually reaches an ACCEPT/REJECT halt. All the pushdown stores are assumed to contain the empty word Λ initially.

EXAMPLE 1-11

The finite machine M_4 with one pushdown store y, described in Fig. 1-7, accepts every word over Σ of the form $a^n b^n$, $n \geqq 0$, and rejects all other words over Σ; that is, $accept(M_4) = \{a^n b^n | n \geqq 0\}$, and $reject(M_4) = \Sigma^* - accept(M_4)$.

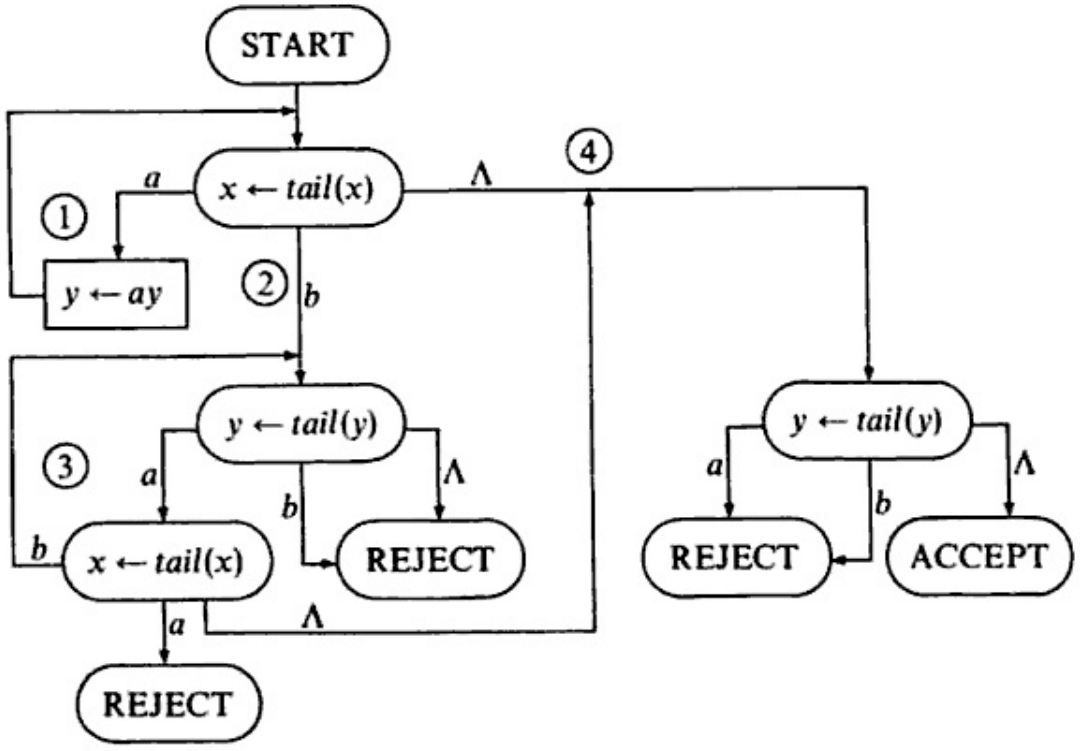


Figure 1-7 The finite machine M_4 with one pushdown store that accepts $\{a^n b^n | n \geqq 0\}$.

Suppose that $x = a^n b^n$. First we move the a's from x to y so that at point 2, a^n is stored in y while b^{n-1} is still in x. Then in loop 3 we eliminate the a's from y and the b's from x, comparing their numbers, until both x and y are empty at point 4.

□

EXAMPLE 1-12

The finite machine M_5 with two pushdown stores y_1 and y_2, described in Fig. 1-8, accepts every word of Σ of the form $a^n b^n a^n$, $n \geqq 0$, and rejects all other words over Σ; that is, $accept(M_5) = \{a^n b^n a^n \mid n \geqq 0\}$, and $reject(M_5) = \Sigma^* - accept(M_5)$.

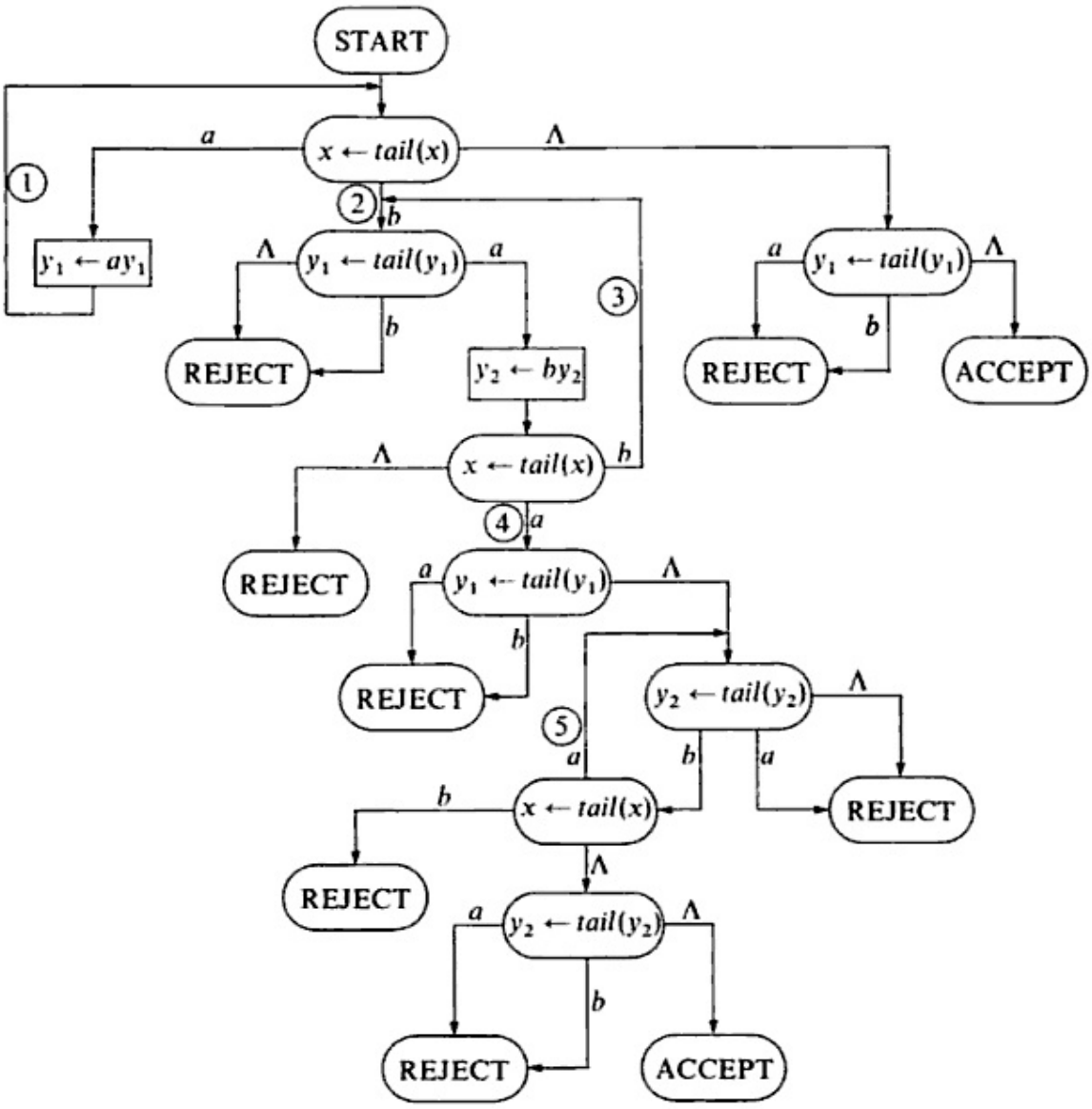


Figure 1-8 The finite machine M_5 with two pushdown stores that accepts $\{a^n b^n a^n \mid n \geqq 0\}$.

Suppose that $x = a^n b^n a^n$. First, in loop 1 we move the left a's from x to y_1 so that at point 2 we have a^n in y_1 and $b^{n-1}a^n$ in x. Then, in loop 3 we move the b's from x to y_2, comparing at the same time the number of a's in y_1 and the number of b's in x, so that at point 4 we have b^n in y_2 and a^{n-1} in x (y_1 is empty). Finally, in loop 5 we remove the b's from y_2 and the a's from x, comparing their numbers, until both x and y_2 are empty.

□

We are now interested in investigating whether or not the addition of pushdown stores really increases the power of the class of finite machines. We note that for a fixed alphabet Σ:

1. The class of finite machines (with no pushdown stores) has the same power as the class of finite automata introduced in Sec. 1-1.2. In other words, a set S over Σ is regular if and only if $S = accept(M)$ for some finite machine M over Σ with no pushdown stores.

2. The class of finite machines with one pushdown store is more powerful than the class of finite machines with no pushdown stores. For example, there is no finite machine over $\Sigma = \{a, b\}$ with no pushdown stores that is equivalent to the finite machine M_4 with one pushdown store described in Example 1-11.

3. The class of finite machines with two pushdown stores is more powerful than the class of finite machines with only one pushdown store. For example, there is no finite machine over $\Sigma = \{a, b\}$ with one pushdown store that is equivalent to the finite machine M_5 with two pushdown stores described in Example 1-12.

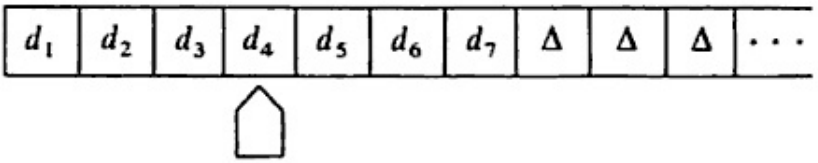
4. For $n \geqq 2$, the class of finite machines with n pushdown stores has exactly the same power as the class of finite machines with two pushdown stores.

It can be shown that the class of finite machines with two pushdown stores has exactly the same power as the class of Post machines, or equivalently, the class of Turing machines.

THEOREM 1-4 (Minsky). *The class of finite machines over Σ with two pushdown stores has the same power as the class of Turing machines over Σ.*

That is, for every finite machine over Σ with two pushdown stores, there is an equivalent Turing machine over Σ and vice versa.

Proof. For any given Turing machine over Σ, we can construct an equivalent finite machine over Σ with two pushdown stores y_1 and y_2. The finite machine simulates the operations of the Turing machine in such a way that at any stage of the computation y_1 has the contents of the tape to the left of the tape head (including the symbol being scanned) and y_2 has the contents of the tape to the right of the tape head (ignoring the infinite string of Δ's). For example, if the tape is



and the symbol being scanned is d_4, then

$$y_1 = d_4d_3d_2d_1 \qquad \text{and} \qquad y_2 = d_5d_6d_7$$

Note that the left part of the tape is stored in y_1 in reverse order so that the symbol being scanned by the tape head is the leftmost symbol of y_1.

Now, we can simulate the moves of the Turing machine by modifying the leftmost symbols of y_1 and y_2. For example, if the next move of the Turing machine is (d_4, β, R), then the contents of y_1 and y_2 will be changed to

$$y_1 = d_5\beta d_3d_2d_1 \qquad \text{and} \qquad y_2 = d_6d_7$$

If, instead, the next move is (d_4, β, L), then the contents of y_1 and y_2 will be changed to

$$y_1 = d_3d_2d_1. \qquad \text{and} \qquad y_2 = d_5d_6d_7$$

As in the proof of Theorem 1-3, there are two special cases that should be treated separately: The first case occurs when the tape head reaches d_7 (that is, $y_2 = \Lambda$) and is instructed to make a right move; and the second case occurs when the tape head reaches d_1 (that is, $y_1 = d_1$) and is instructed to make a left move.

Conversely, for every finite machine over Σ with two pushdown stores we can construct an equivalent Turing machine over Σ. The current value of y_1 is stored in the odd cells and the current value of y_2 is stored in the even cells of the tape (or vice versa). If the input word w is of length k, say, $w = \sigma_1\sigma_2 \ldots \sigma_k$, then the leftmost $k + 2$ cells of the tape are reserved to store the current value of x (with Δ on the left and the special symbol $\#$ on the right). For example, if the input word is $w = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_5$ and at some stage of the computation of the finite machine we have

$$x = \sigma_3\sigma_4\sigma_5$$
$$y_1 = d_1d_2d_3d_4$$
$$y_2 = e_1e_2$$

then the corresponding tape in the Turing machine is

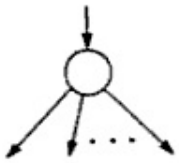
Δ	Δ	Δ	σ_3	σ_4	σ_5	#	d_1	e_1	d_2	e_2	d_3	Δ	d_4	Δ	Δ	Δ	$\cdots$

Now, the operations on y_1 and y_2 of the finite machine are simulated by applying the corresponding operations on the even and odd cells of the tape to the right of #.

Q.E.D.

1-2.4 Nondeterminism

An important generalization of the notion of machines discussed above is obtained by considering nondeterministic machines. A *nondeterministic machine* is a machine which may have in its flow-diagram *choice branches* of the form



That is, whenever we reach upon computation such a choice branch, we may choose *arbitrarily* any one of the possible exits and proceed with the computation as usual. For Turing machines we introduce nondeterminism, not by using choice branches, but just by removing the restriction that "all arrows leading from the same state i must have distinct α's."

A word $w \in \Sigma^*$ is said to be *accepted* by a nondeterministic Turing machine, Post machine, or finite machine M (with or without pushdown stores) over Σ if *there exists* a computation of M starting with input $x = w$ which eventually reaches an ACCEPT halt. If w is not accepted but there is a computation leading to a REJECT halt, then w is said to be *rejected*; otherwise, $w \in loop(M)$.

EXAMPLE 1-13

The nondeterministic finite machine M_6 with one pushdown store y, described in Fig. 1-9, accepts every word over $\Sigma = \{a, b\}$ of the form ww^R,

where w^R stands for the word w reversed, and rejects all other words over Σ; that is, $accept(M_6) = \{ww^R | w \in \{a, b\}^*\}$, and $reject(M_6) = \Sigma^* - accept(M_6)$. Thus, $accept(M_6)$ includes, for example, the words Λ, *aa*, *babbab*, *bbbabbabbb*, and so forth.

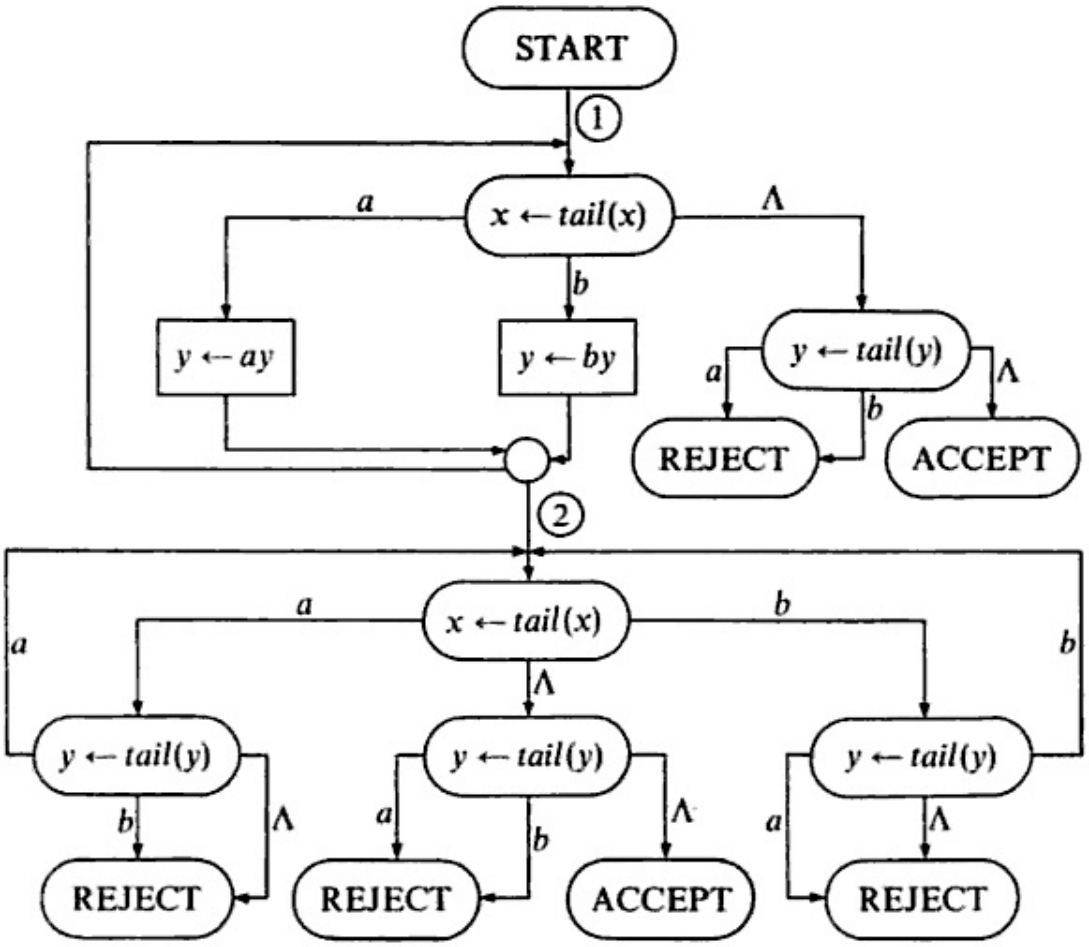


Figure 1-9 The nondeterministic finite machine M_6 that accepts $\{ww^R | w \in \{a, b\}^*\}$.

Suppose that $x = ww^R$. We begin to scan w at point 1 and scan w^R at point 2. However, there is one major difficulty: While we are scanning x from left to right, there is no way to recognize the *breakpoint*, i.e., the point when we finish scanning w and begin scanning w^R. We use a *choice branch* to overcome this difficulty. Whenever we reach the choice branch, we consider both cases at once: We may choose to go back to point 1 (that is, we are still scanning w), or we may choose to go to point 2 (that is, we are at the breakpoint) arbitrarily.

□

A natural question that may be asked is "Does the use of nondeterminism really increase the power of the class of machines?" It can be shown that *the classes of nondeterministic Turing machines, nondeterministic Post machines, and nondeterministic finite machines with two or more pushdown stores have the same power as the class of Turing machines.* The relation between the different classes of finite machines over the same alphabet Σ can be summarized as follows:

$$
\begin{array}{c}
\{\text{Finite automata}\} \\
\| \\
\{\text{Finite machines with no pushdown stores}\} \\
\| \\
\{\text{Nondeterministic finite machines with no pushdown stores}\} \\
\wedge \\
\{\text{Finite machines with one pushdown store}\} \\
\wedge \\
\{\text{Nondeterministic finite machines with one pushdown store}\} \\
\wedge \\
\{\text{Finite machines with two pushdown stores}\} \\
\| \\
\{\text{Nondeterministic finite machines with two pushdown stores}\} \\
\| \\
\{\text{Turing machines}\}
\end{array}
$$

Note the following:

1. There is no nondeterministic finite machine with no pushdown stores that is equivalent to the finite machine M_4 with one pushdown store (Example 1-11).
2. There is no finite machine with one pushdown store that is equivalent to the nondeterministic finite machine M_6 with one pushdown store (Example 1-13).
3. There is no nondeterministic finite machine with one pushdown store that is equivalent to the finite machine M_5 with two pushdown stores (Example 1-12).

1-3. TURING MACHINES AS ACCEPTORS

In this section we introduce two new classes of sets of words over an alphabet Σ: *recursively enumerable sets* and *recursive sets.* These are very rich classes

of sets; actually, the intention is to have these classes rich enough to include exactly those sets of words over Σ which are partially acceptable or acceptable by any computing device. Since by Church's thesis, a Turing machine is the most general computing device, it is quite natural that we define the two classes by means of Turing machines.

We define a set S of words over Σ to be recursively enumerable if there is a Turing machine M over Σ which accepts every word in S and either rejects or loops for every word in $\Sigma^* - S$; that is, $accept(M) = S$, and $reject(M) \cup loop(M) = \Sigma^* - S$. However, the main drawback is that if we apply the machine to some word w in $\Sigma^* - S$, there is no guarantee as to how the machine will behave: It may reject w or loop forever. Therefore we also introduce the class of recursive sets, which is a proper subclass of the class of recursively enumerable sets.

We define a set S of words over Σ to be recursive if there is a Turing machine M over Σ which accepts every word in S and rejects every word in $\Sigma^* - S$; that is, $accept(M) = S$, $reject(M) = \Sigma^* - S$, and $loop(M) = \phi$. There is a very important relation between the class of recursively enumerable sets and the class of recursive sets: *a set S over Σ is recursive if and only if both S and $\Sigma^* - S$ are recursively enumerable.*

1-3.1 Recursively Enumerable Sets

Alternative definitions for a set S of words over Σ to be *recursively enumerable* are:

1. There is a Turing machine which accepts every word in S and either rejects or loops for every word in $\Sigma^* - S$.
2. There is a Post machine which accepts every word in S and either rejects or loops for every word in $\Sigma^* - S$.
3. There is a finite machine with two pushdown stores which accepts every word in S and either rejects or loops for every word in $\Sigma^* - S$.

EXAMPLE 1-14

Examples 1-8, 1-10, and 1-12 imply that the following sets of words over $\Sigma = \{a, b\}$ are recursively enumerable: $\{a^n b^n | n \geqq 0\}$, {all words over Σ with the same number of a's and b's}, and $\{a^n b^n a^n | n \geqq 0\}$.

□

Although the class of recursively enumerable sets is very rich, *there exist sets of words over* $\Sigma = \{a, b\}$ *that are not recursively enumerable.* We now demonstrate such a set of words. All words in $\{a, b\}^*$ can be ordered by the *natural lexicographic* order

$$\Lambda, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, aaab, \ldots$$

that is, the words are ordered by length and lexicographically within each length. Thus it makes sense to talk about the jth string x_j in $\{a, b\}^*$. Every Turing machine over $\Sigma = \{a, b\}$ can be encoded (see Prob. 1-13) as a string in $\{a, b\}^*$ in such a way that each string in $\{a, b\}^*$ represents a Turing machine and vice versa. Therefore we can talk about the jth Turing machine T_j over $\Sigma = \{a, b\}$, that is, the one represented by the jth string in $\{a, b\}^*$,

Consider now the set

$$L_1 = \{x_j | x_j \text{ is not accepted by } T_j\}$$

Clearly, L_1 could not be accepted by any Turing machine. If it were, let T_j be a Turing machine accepting L_1. Then, by definition of L_1, $x_j \in L_1$ if and only if x_j is not accepted by T_j. But since T_j accepts L_1, we obtain the contradiction that x_j is accepted by T_j if and only if x_j is not accepted by T_j. Thus, there is no Turing machine accepting L_1; that is, L_1 *is not a recursively enumerable set.*

1-3.2 Recursive Sets

Alternative definitions for a set S of words over Σ to be *recursive* are:

1. There is a Turing machine which accepts every word in S and rejects every word in $\Sigma^* - S$.
2. There is a Post machine which accepts every word in S and rejects every word in $\Sigma^* - S$.
3. There is a finite machine with two pushdown stores which accepts every word in S and rejects every word in $\Sigma^* - S$.

EXAMPLE 1-15

Examples 1-8 and 1-12 imply that the following sets of words over $\Sigma = \{a, b\}$ are recursive: $\{a^n b^n | n \geqq 0\}$ and $\{a^n b^n a^n | n \geqq 0\}$.

□

The most important results regarding recursive sets are:

1. *If a set over* Σ *is recursive, then its complement* $\Sigma^* - S$ *is also recursive.*† If S is a recursive set over Σ, then there is a Post machine M which always halts and accepts all words in S and rejects all words in $\Sigma^* - S$. Construct a Post machine M' from M by interchanging the ACCEPT and REJECT halts. Clearly M' accepts all words in $\Sigma^* - S$ and rejects all words in S.

2. *A set* S *over* Σ *is recursive if and only if both* S *and* $\Sigma^* - S$ *are recursively enumerable.* (*a*) If S is recursive, then (by result 1 above) $\Sigma^* - S$ is recursive; and clearly if S and $\Sigma^* - S$ are recursive, then S and $\Sigma^* - S$ are recursively enumerable. (*b*) If S and $\Sigma^* - S$ are recursively enumerable, there must exist Turing machines M_1 and M_2 over Σ which accept S and $\Sigma^* - S$, respectively. M_1 and M_2 can be used to construct a new Turing machine M which simulates the computations of M_1 and M_2‡ in such a way that M accepts all words in S and rejects all words in $\Sigma^* - S$; that is, S is a recursive set. Note that the contents of the tape of M must be organized in such a way that at any given time it contains six pieces of information: the contents of the tapes of M_1 and M_2, the location of the tape heads of M_1 and M_2, and the current states of M_1 and M_2.

3. *The class of recursively enumerable sets over* Σ *properly includes the class of all recursive sets over* Σ. It is clear that every recursive set is recursively enumerable. We shall show that there exists a recursively enumerable set over Σ that is not recursive. For this purpose, let x_i denote the ith word in $\{a, b\}^*$ and T_i the ith Turing machine over $\Sigma = \{a, b\}$, and consider the set $L_2 = \{x_i | x_i \text{ is accepted by } T_i\}$. (*a*) L_2 is a recursively enumerable set because a Turing machine T can be constructed that will accept all words in L_2. For any given word $w \in \Sigma^*$, T will start to generate the words $x_1, x_2, x_3, \ldots$, and test each word generated until it finds the word $x_i = w$, thereby determining that w is the ith word in the enumeration. Then T generates T_i, the ith Turing machine, and simulates its operation. Therefore w is accepted by T if and only if $x_i = w$ is accepted by T_i. (*b*) L_2 is not a recursive set since $\Sigma^* - L_2 = L_1$ and L_1 is not recursively enumerable, as has been shown.

† However, there are recursively enumerable sets of words over Σ such that their complement is not recursively enumerable.

‡ The simulations of M_1 and M_2 are done "in parallel" (by applying alternately one move from each machine) so that neither requires the termination of the other in order to have its behavior completely simulated.

1-3.3 Formal Languages

A *type-0 grammar G over* Σ consists of the following:

1. A finite (nonempty) set V of distinct symbols (not in Σ), called *variables*, containing one distinguished symbol S, called the *start variable*.
2. A finite set P of *productions* in which each production is of the form

$$\alpha \to \beta \qquad \text{where } \alpha \in (V \cup \Sigma)^+ \text{ and } \beta \in (V \cup \Sigma)^*\dagger$$

A word $w \in \Sigma^*$ is said to be *generated* by G if there is a finite sequence of words over $V \cup \Sigma$

$$w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n \qquad n \geqq 1$$

such that w_0 is the start variable S, $w_n = w$, and w_{i+1} is obtained from w_i, $0 \leqq i < n$, by replacing some occurrence of a substring α (which is the left-hand side of some production of P) in w_i by the corresponding β (which is the right-hand side of that production).‡ The set of all words generated by G is called the *language generated by G*.

EXAMPLE 1-16

Let $\Sigma = \{a, b\}$. Consider the grammar G_1 where $V = \{S\}$ and $P = \{S \to aSb, S \to ab\}$. By applying the first production $n - 1$ times followed by an application of the second production, we have

$$S \Rightarrow aSb \Rightarrow a^2Sb^2 \Rightarrow a^3Sb^3 \Rightarrow \cdots \Rightarrow a^{n-1}Sb^{n-1} \Rightarrow a^nb^n$$

It is clear that these are the only words generated by the grammar. Thus the language generated by the grammar G_1 is

$$\{a^nb^n | n \geqq 1\}$$

□

EXAMPLE 1-17

Let $\Sigma = \{a, b\}$. Consider the grammar G_2, where $V = \{A, B, S\}$ and P consists of six productions:

1 $S \to aSBA$
2 $S \to abA$
3 $AB \to BA$
4 $bB \to bb$
5 $bA \to ba$
6 $aA \to aa$

† $(V \cup \Sigma)^*$ stands for all words over $V \cup \Sigma$ except Λ (the empty word); that is, $(V \cup \Sigma)^+ = (V \cup \Sigma)^* - \{\Lambda\}$.

‡ α is a substring in w, if and only if there exists words u, v (possibly empty) such that $uxv = w_i$; w_{i+1} is then the word $u\beta v$.

The language generated by the grammar G_2 is

$$\{a^n b^n a^n | n \geqq 1\}$$

To obtain $a^n b^n a^n$, for some $n \geqq 1$, first we apply $n-1$ times production 1 to obtain $a^{n-1}S(BA)^{n-1}$; then we use production 2 once to obtain $a^n bA(BA)^{n-1}$. Production 3 enables us to rearrange the B's and A's to obtain $a^n bB^{n-1}A^n$. Next we use $n-1$ times production 4 to obtain $a^n b^n A^n$; then we use production 5 once to obtain $a^n b^n aA^{n-1}$. Finally, we use $n-1$ times production 6 to obtain $a^n b^n a^n$.

Now, let us show that no other words over Σ can be generated by the grammar G_2. At the beginning we can apply only production 1 (and possibly production 3) until production 2 is applied the first time. At this point we have $a^n b$ followed by some string of n A's and $n-1$ B's (in any order). (Now we can apply productions 3 to 6, but in each step it is always true that the left part of the word consists of a's and b's while the right part consists of A's and B's.) Next we must apply $n-1$ times production 4 (and possibly production 3) to obtain the word $a^n b^n A^n$. The only other production we could possibly apply is production 5. However, in this case sooner or later we would have to eliminate aB in the word; but there is no such production in P, and so we must obtain the word $a^n b^n A^n$. The only way we can proceed now is by applying production 5 once and then applying $n-1$ times production 6 to obtain the word $a^n b^n a^n$.

□

A set of words over Σ is called *type-0 language* if it can be generated by some type-0 grammar over Σ. We have the following theorem.

THEOREM 1-5 (Chomsky). *A set of words over Σ is recursively enumerable if and only if it is a type-0 language.*

Certain restrictions can be made on the nature of the productions of a type-0 grammar to give three other types of grammars. A grammar is said to be of

1. *Type-1 (context-sensitive)* if for every production $\alpha \to \beta$ in P
$$|\alpha| \leqq |\beta| \dagger$$
2. *Type-2 (context-free)* if for every production $\alpha \to \beta$ in P
$$\alpha \in V \quad \text{and} \quad \beta \in (V \cup \Sigma)^+$$
3. *Type-3 (regular)* if every production in P is of the form
$$A \to \sigma B \quad \text{or} \quad A \to \sigma$$
where $A, B \in V$ and $\sigma \in \Sigma$.

† Recall that $|x|$ stands for the length of the string (i.e., the number of letters in x); in particular, $|\Lambda| = 0$.

In any one of the above grammars we allow also the use of a special production, $S \to \Lambda$, but only if S (the start symbol) does not occur in the right-hand side of any production in P.

Correspondingly we define a set of words over Σ to be a *language of type-i*, $1 \leqq i \leqq 3$ (or *context-sensitive*, *context-free*, and *regular* language, respectively), if it can be generated by some type-i grammar. The following table summarizes the inclusion relations between the classes of sets over Σ discussed so far.

$$
\begin{aligned}
\{\text{Type-0 languages}\} &= \{\text{recursively enumerable sets}\} \\
&= \{\text{all sets accepted by some finite machine with two pushdown stores}\} \\
&\cup \\
\{\text{Recursive sets}\} & \\
&\cup \\
\{\text{Type-1 languages}\} & \\
&\cup \\
\{\text{Type-2 languages}\} &= \{\text{all sets accepted by some nondeterministic finite machine with one pushdown store}\} \\
&\cup \\
\{\text{Type-3 languages}\} &= \{\text{regular sets}\} \\
&= \{\text{all sets accepted by some finite machine with no pushdown stores}\}
\end{aligned}
$$

All the above inclusions are actually proper since for $\Sigma = \{a, b\}$ [see, for example, Hopcroft and Ullman (1969)].

1. $L_2 = \{x_i | x_i \text{ is accepted by } T_i\}$ is a type-0 language but not a recursive set.
2. $\{x_i | x_i$ is not generated by the ith type-1 grammar$\}$† is a recursive set but not a type-1 language.
3. $\{a^n b^n a^n | n \geqq 1\}$ is a type-1 but not a type-2 language.
4. $\{a^n b^n | n \geqq 1\}$ is a type-2 but not a type-3 language.

1-4 TURING MACHINES AS GENERATORS

In this section we introduce a class of n-ary partial functions ($n \geqq 1$), called *Turing computable functions*, mapping n-tuples of words over Σ into

† Note that in order to carry out the enumeration of type-1 grammars over $\Sigma = \{a, b\}$, we must consider only type-1 grammars with variables from $V = \{v_1, v_2, v_3, \ldots\}$; that is, if the grammar has n variables, they must be $v_1, v_2, \ldots, v_n$.

words over Σ. This is a very rich class of functions; actually, the intention is to have this class rich enough to include exactly all computable n-ary partial functions mapping n-tuples of words over Σ into words over Σ. Again, since, by Church's thesis, a Turing machine is the most general computing device, it is quite natural that we define the class by means of Turing machines.

A partial function† $f(x_1, \ldots, x_n)$, $n \geqq 1$, mapping n-tuples of words over $\Sigma = \{a, b\}$ into words over Σ, is said to be *Turing computable* if there is a Turing machine M over $\{a, b, *\}$ which behaves as follows: For every n-tuple $(w_1, \ldots, w_n)$ of words over Σ, M takes the string $w_1 * \ldots * w_n$ as input and

1. If $f(w_1, \ldots, w_n)$ is undefined, then M will loop forever.
2. If $f(w_1, \ldots, w_n)$ is defined, then M will eventually halt (either rejecting or accepting the input) with a tape containing the value of $f(w_1, \ldots, w_n)$ *followed only by* Δ's.

EXAMPLE 1-18

The Turing machine M_7 over $\{a, b, *\}$, described in Fig. 1-10, computes the concatenation function over $\Sigma = \{a, b\}$: It takes a pair of words (w_1, w_2) as input and yields the word $w_1 w_2$ as output. For example, if $w_1 = abaa$

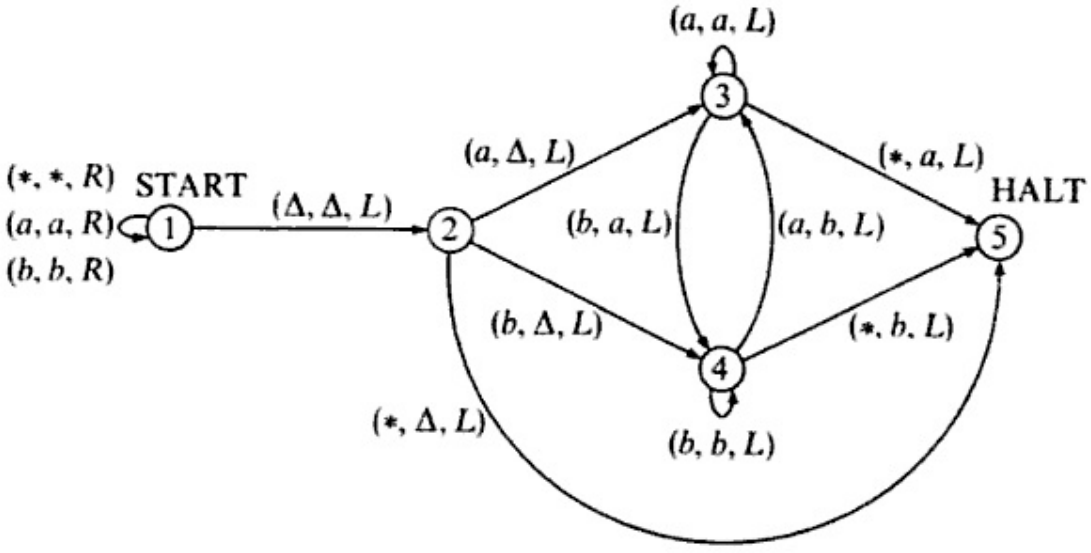


Figure 1-10 The Turing machine M_7 that computes the concatenation function.

† The words *partial function* indicate that $f(w_1, \ldots, w_n)$ may be undefined for some n-tuple of words $(w_1, \ldots, w_n)$ over Σ, in contrast with a *total function* for which $f(w_1, \ldots, w_n)$ is defined (and yields a word in Σ) for all possible n-tuples of words over Σ.

and $w_2 = bbab$, then the input and output tapes look like this:

Input tape	a	b	a	a	$*$	b	b	a	b	Δ	Δ	Δ $\cdots$
Output tape	a	b	a	a	b	b	a	b	Δ	Δ	Δ	Δ $\cdots$

We start with state 1 and scan the input tape moving right until we reach the first Δ symbol (state 2). Now we move left until we reach the $*$ symbol (state 5). While we are moving to the left, we read each symbol on the tape cell, remember it, move one cell to the left, and print it on the new cell. We are at state 3 whenever we have to remember the symbol a, and at state 4 whenever we have to remember the symbol b.

□

In this section we give an alternative definition of the class of Turing computable functions; that is, we show that a function is Turing computable if and only if it is a *partial recursive* function. First we shall discuss the class of *primitive recursive* functions, which is a proper subclass of the class of partial recursive functions. Virtually all functions of practical value are primitive recursive.

1-4.1 Primitive Recursive Functions

Every n-ary ($n \geqq 1$) primitive recursive function is a total function mapping n-tuples of words over Σ into words over Σ, but not every such total function is a primitive recursive function. The class of all primitive recursive functions over an alphabet $\Sigma = \{a, b\}$ is defined as follows:

1. *Base functions:*

   $nil(x) = \Lambda$ (that is, for every argument $x \in \Sigma^*$, the value of $nil(x)$ is Λ, the empty word)

   $consa(x) = ax$

   $consb(x) = bx$

   These are primitive recursive functions over Σ.

2. *Composition:*

   If $h, g_1, \ldots, g_m$ are primitive recursive functions over Σ, then so is the n-ary function f defined by

   $$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

(In each g_i some of the arguments may be absent.)

3. *Primitive recursion:*
   If g, h_1, and h_2 are primitive recursive functions over Σ, then so is the n-ary function f defined by
   (*a*) If $n = 1$,

$$\begin{aligned} f(\Lambda) &= w \qquad \text{(any word } w \text{ of } \Sigma^*) \\ f(ax) &= h_1(x, f(x)) \\ f(bx) &= h_2(x, f(x)) \end{aligned}$$

   (In h_1 and h_2 some of the arguments may be absent.)
   (*b*) If $n \geqq 2$,

$$\begin{aligned} f(\Lambda, x_2, \ldots, x_n) &= g(x_2, \ldots, x_n) \\ f(ax_1, x_2, \ldots, x_n) &= h_1(x_1, x_2, \ldots, x_n, f(x_1, x_2, \ldots, x_n)) \\ f(bx_1, x_2, \ldots, x_n) &= h_2(x_1, x_2, \ldots, x_n, f(x_1, x_2, \ldots, x_n)) \end{aligned}$$

   (In h_1 and h_2 some of the arguments may be absent.)

EXAMPLE 1-19

The following are primitive recursive functions over $\Sigma = \{a, b\}$.

1. The constant functions a and b, where $a(x) = a$ and $b(x) = b$ for all x, are primitive recursive because, by composition,

$$\begin{aligned} a(x) &= consa(nil(x)) \\ b(x) &= consb(nil(x)) \end{aligned}$$

2. The identity function *iden*, where $iden(x) = x$ for all x, is primitive recursive because by primitive recursion (case $n = 1$),

$$\begin{aligned} iden(\Lambda) &= \Lambda \\ iden(ax) &= consa(x) \\ iden(bx) &= consb(x) \end{aligned}$$

3. The concatenation function *append*, where $append(x_1, x_2) = x_1x_2$ for all x_1 and x_2 [that is, $append(x_1, x_2)$ concatenates the words x_1 and x_2 into one word x_1x_2] is primitive recursive because, by primitive recursion (case $n = 2$),

$$\begin{aligned} append(\Lambda, x_2) &= iden(x_2) \\ append(ax_1, x_2) &= consa(append(x_1, x_2)) \\ append(bx_1, x_2) &= consb(append(x_1, x_2)) \end{aligned}$$

4. The *head* and *tail* functions, where $head(x)$ = "the leftmost letter of x" and $tail(x)$ = "the word x after removing the leftmost letter" [$head(\Lambda) =$

Λ and $tail(\Lambda) = \Lambda$], are primitive recursive because, by primitive recursion (case $n = 1$),

$$\begin{aligned} head(\Lambda) &= \Lambda & tail(\Lambda) &= \Lambda \\ head(ax) &= a(x) & tail(ax) &= iden(x) \\ head(bx) &= b(x) & tail(bx) &= iden(x) \end{aligned}$$

5. The *reverse* function, where $reverse(x)$ = "the letters of x in reversed order," is a primitive recursive because, by primitive recursion (case $n = 1$),

$$\begin{aligned} reverse(\Lambda) &= \Lambda \\ reverse(ax) &= append(reverse(x), a(x)) \\ reverse(bx) &= append(reverse(x), b(x)) \end{aligned}$$

6. The *right* and *left* functions, where $right(x)$ = "the rightmost letter of x" and $left(x)$ = "the word x after removing the rightmost letter" [$right(\Lambda) = left(\Lambda) = \Lambda$], are primitive recursive because, by composition,

$$right(x) = head(reverse(x))$$

$$left(x) = reverse(tail(reverse(x)))$$

7. The next-string function $next(x)$ = "the next word in the natural lexicographic order" is primitive recursive because

$$next(x) = reverse(nextl(reverse(x)))$$

where

$$\begin{aligned} nextl(\Lambda) &= a \\ nextl(ax) &= consb(x) \\ nextl(bx) &= consa(nextl(x)) \end{aligned}$$

8. The conditional function $cond(x_1, x_2, x_3)$ = "if $x_1 \neq \Lambda$ then x_2 else x_3" is primitive recursive because, by primitive recursion (case $n = 3$),

$$\begin{aligned} cond(\Lambda, x_2, x_3) &= iden(x_3) \\ cond(ax_1, x_2, x_3) &= iden(x_2) \\ cond(bx_1, x_2, x_3) &= iden(x_2) \end{aligned}$$

9. The predecessor function $pred(x)$ = "the predecessor of x in the natural lexicographic ordering" [$pred(\Lambda) = \Lambda$] is primitive recursive because

$$pred(x) = reverse(predl(reverse(x)))$$

where

$$\begin{aligned} predl(\Lambda) &= \Lambda \\ predl(ax) &= cond(x, consb(predl(x)), nil(x)) \\ predl(bx) &= consa(x) \end{aligned}$$

□

For abbreviation, from now on we shall write Λ, x, a, and b where $nil(x)$, $iden(x)$, $a(x)$, and $b(x)$, respectively, are required.

So far we have defined the class of primitive recursive functions over Σ; we proceed to define the class of primitive recursive predicates. A total predicate† $p(x_1, \ldots, x_n)$, $n \geqq 1$, mapping n-tuples of words over Σ into $\{true, false\}$ is said to be a *primitive recursive predicate* if the function defined by

$$\begin{array}{lll} \Lambda & \text{if} & p(x_1, \ldots, x_n) = false \\ a & \text{if} & p(x_1, \ldots, x_n) = true \end{array}$$

is primitive recursive. This function is called the *characteristic function of* p and is denoted by $f_p(x_1, \ldots, x_n)$.

EXAMPLE 1-20

1. The predicates *true* and *false*, where $true(x) = true$ and $false(x) = false$ for all x, are primitive recursive predicates because their characteristic functions $a(x)$ and $nil(x)$ are primitive recursive functions.

2. The predicate *null*, where $null(x) =$ "is $x = \Lambda$?" is a primitive recursive predicate because its characteristic function f is primitive recursive:

$$f(\Lambda) = a \qquad \text{and} \qquad f(ax) = f(bx) = \Lambda$$

3. The predicate $eqa(x) =$ "is $x = a$?" is a primitive recursive predicate because its characteristic function is

$$f(x) = cond(x, cond(pred(x), \Lambda, a), \Lambda)$$

The predicate $eqb(x) =$ "is $x = b$?" is also a primitive recursive predicate because

$$eqb(x) = eqa(pred(x))$$

□

We shall now introduce several interesting results related to primitive recursive functions and primitive recursive predicates. We let $\bar{x}$ stand for $(x_1, \ldots, x_n)$.

1. *If-then-else*:
If $p(\bar{x})$ is a primitive recursive predicate and $f(\bar{x})$ and $g(\bar{x})$ are primitive recursive functions, then the function

† A predicate is actually a function but one which yields for every n-tuple of words over Σ not a word over Σ, but *true* or *false*.

$$\textit{if } p(\bar{x}) \textit{ then } f(\bar{x}) \textit{ else } g(\bar{x}) \qquad \text{that is,} \quad \begin{cases} f(\bar{x}) & \text{if} \quad p(\bar{x}) = \textit{true} \\ g(\bar{x}) & \text{if} \quad p(\bar{x}) = \textit{false} \end{cases}$$

is primitive recursive.

In general, if $p_1(\bar{x}), p_2(\bar{x}), \ldots, p_{m-1}(\bar{x})$ are primitive recursive predicates, and $g_1(\bar{x}), g_2(\bar{x}), \ldots, g_m(\bar{x})$ are primitive recursive functions, then the function

$$\begin{aligned} &\textit{if } p_1(\bar{x}) \textit{ then } g_1(\bar{x}) \\ &\quad \textit{else if } p_2(\bar{x}) \textit{ then } g_2(\bar{x}) \\ &\qquad \textit{else } \ldots \textit{ if } p_{m-1}(\bar{x}) \textit{ then } g_{m-1}(\bar{x}) \\ &\qquad\quad \textit{else } g_m(\bar{x}) \end{aligned}$$

is primitive recursive. Note that the function is identical to

$$\begin{aligned} &\textit{cond}(f_{p_1}(\bar{x}), g_1(\bar{x}), \textit{cond}(f_{p_2}(\bar{x}), g_2(\bar{x}), \ldots, \\ &\qquad \textit{cond}(f_{p_{m-1}}(\bar{x}), g_{m-1}(\bar{x}), g_m(\bar{x})) \ldots)) \end{aligned}$$

where $f_{p_i}(\bar{x})$ is the characteristic function of p_i for $1 \leqq i \leqq m-1$.

2. *Propositional connectives*:

If $p(\bar{x})$, $q(\bar{x})$, and $r(\bar{x})$ are primitive recursive predicates, then so are the following predicates:

$$\sim p(\bar{x}) \qquad \begin{cases} \textit{true} & \text{if } p(\bar{x}) = \textit{false} \\ \textit{false} & \text{otherwise} \end{cases}$$

$$p(\bar{x}) \vee q(\bar{x}) \qquad \begin{cases} \textit{true} & \text{if } p(\bar{x}) = \textit{true}, \text{ or } q(\bar{x}) = \textit{true}, \text{ or both} \\ \textit{false} & \text{otherwise} \end{cases}$$

$$p(\bar{x}) \wedge q(\bar{x}) \qquad \begin{cases} \textit{true} & \text{if } p(\bar{x}) = q(\bar{x}) = \textit{true} \\ \textit{false} & \text{otherwise} \end{cases}$$

$$p(\bar{x}) \equiv q(\bar{x}) \qquad \begin{cases} \textit{true} & \text{if } p(\bar{x}) = q(\bar{x}) = \textit{true} \text{ or } p(\bar{x}) = q(\bar{x}) = \textit{false} \\ \textit{false} & \text{otherwise} \end{cases}$$

The characteristic functions of the first two predicates are

$$f_{\sim p}(\bar{x}) \qquad \textit{if null}(f_p(\bar{x})) \textit{ then } a \textit{ else } \Lambda$$

$$f_{p \vee q}(\bar{x}) \qquad \textit{if null}(\textit{append}(f_p(\bar{x}), f_q(\bar{x}))) \textit{ then } \Lambda \textit{ else } a$$

Therefore $\sim$ p and p $\vee$ q are primitive recursive predicates. The other two predicates are also primitive recursive since

$$\begin{aligned} p(\bar{x}) \wedge q(\bar{x}) \quad &\text{is} \quad \sim((\sim p(\bar{x})) \vee (\sim q(\bar{x}))) \\ p(\bar{x}) \equiv q(\bar{x}) \quad &\text{is} \quad (p(\bar{x}) \vee \sim q(\bar{x})) \wedge (q(\bar{x}) \vee \sim p(\bar{x})) \end{aligned}$$

Three very useful primitive recursive functions are *code*, *decode*, and *sub*, where $code(x) = a^i$ and $decode(a^i) = x$ iff x is the ith word in the natural lexicographic ordering ($i \geqq 0$) and $sub(a^i, a^j) = a^{i-j}$ (if $i \geqq j$). In all other cases *decode* and *sub* are defined (arbitrarily) to be Λ. In Prob. 1-17 the reader is asked to prove that these functions are indeed primitive recursive. We shall now illustrate two applications of these functions.

3. *Bounded minimization*:

If the predicate $p(\bar{x}, y)$ is primitive recursive, then so is the function $f(\bar{x}, z)$ which is defined as follows:

The value of $f(\bar{x}, z)$ is the *first* word y of Σ^* in the natural lexicographic order (that is, Λ, a, b, aa, ab, ba, bb, aaa, . . .) such that $p(\bar{x}, y) =$ *true* or $y = z$. For all y' preceding y ($y \neq \Lambda$) in this order, $p(\bar{x}, y') =$ *false* and $y' \neq z$.

Note that the test $y = z$ ensures that the function obtained is always defined; that is, f is a total function. For brevity, we shall denote this definition by

$$f(\bar{x}, z) = h(\Lambda, \bar{x}, z) \text{ where}$$
$$h(y, \bar{x}, z) \Leftarrow \text{if } p(\bar{x}, y) \vee y = z \text{ then } y \text{ else } h(next(y), \bar{x}, z)$$

$f(\bar{x}, z)$ is primitive recursive because

$$f(\bar{x}, z) = g(code(z), \bar{x}, code(z))$$

where

$$g(\Lambda, \bar{x}, t) = decode(t)$$
$$g(as, \bar{x}, t) = \text{if } p(\bar{x}, decode(sub(t, as))) \text{ then } decode(sub(t, as)) \text{ else } g(s, \bar{x}, t)$$

$g(bs, \bar{x}, t)$ can be defined arbitrarily.

4. *Equality predicate*:

The equality predicate $equal(x, y) =$ "is $x = y$?" is a primitive recursive predicate because it can be expressed as

$$equal(x, y) = null(append(sub(code(x), code(y)), sub(code(y), code(x))))$$

1-4.2 Partial Recursive Functions

The class of all partial recursive functions over an alphabet $\Sigma = \{a, b\}$ is defined as follows:

1. *Base functions*:
   $nil(x)$, $consa(x)$ and $consb(x)$ are partial recursive functions over Σ.

2. *Composition*:
   If $h, g_1, \ldots, g_m$ are partial recursive functions over Σ, then so is the n-ary function f defined by composition.†
3. *Primitive recursion*:
   If g, h_1, and h_2 are partial recursive functions over Σ, then so is the n-ary function f defined by primitive recursion. For $n = 1$ we allow f to be defined also as

$$f(\Lambda) = \text{undefined}$$
$$f(ax) = h_1(x, f(x))$$
$$f(bx) = h_2(x, f(x))$$

   (In h_1 and h_2 some of the arguments may be absent.)
4. *(Unbounded) minimization*:
   If p is a partial recursive predicate,‡ then the function f defined by

$$f(\bar{x}) = h(\bar{x}, \Lambda) \text{ where}$$
$$h(\bar{x}, y) \Leftarrow \text{if } p(\bar{x}, y) \text{ then } y \text{ else } h(\bar{x}, next(y))$$

   is partial recursive. In other words, the value of $f(\bar{x})$ is the *first* word y of Σ^* in the natural lexicographic order (that is, $\Lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots$) such that $p(\bar{x}, y) = true$; for all y' preceding y in this order $p(\bar{x}, y') = false$. In all other cases, that is, if $p(\bar{x}, y) = false$ for all $y \in \Sigma^*$, or $p(\bar{x}, y)$ is undefined for some y and for all y' preceding y in this order $p(\bar{x}, y') = false$, then the value of $f(\bar{x})$ is undefined.

EXAMPLE 1-21

1. The *uhead* and *utail* functions, where $uhead(x)$ = "the leftmost letter of a nonempty word x" and $utail(x)$ = "the tail of a nonempty word x obtained by removing the leftmost letter" [$uhead(\Lambda) = utail(\Lambda) =$ undefined], are partial recursive functions because they can be defined by primitive recursion:

$$uhead(\Lambda) = \text{undefined} \qquad utail(\Lambda) = \text{undefined}$$
$$uhead(ax) = a \qquad utail(ax) = x$$
$$uhead(bx) = b \qquad utail(bx) = x$$

† Since a partial recursive function may be undefined for some arguments, a composition of partial recursive functions may require the value of such a function for undefined arguments. We agree that any partial function is undefined whenever at least one of its arguments is undefined.

‡ That is the characteristic function $f_p(\bar{x})$ is a partial recursive function, where $f_p(\bar{x})$ is defined as

Λ	if $p(\bar{x}) = false$
a	if $p(\bar{x}) = true$
undefined	if $p(\bar{x})$ is undefined

2. The function *undef*(x) which is undefined for every string x is a partial recursive function because it can be defined by minimization with p being *false*(x) or alternatively using primitive recursion:

$$undef(\Lambda) = \text{undefined}$$
$$undef(ax) = undef(bx) = undef(x)$$

3. The function *half*, where $half(x) = y$ iff $append(y, y) = x$ [$half(x)$ is undefined if no such y exists], is a partial recursive function because, by minimization,

$$half(x) = h(x, \Lambda) \quad where$$
$$h(x, y) \Leftarrow if\ equal(append(y, y), x)\ then\ y\ else\ h(x, next(y))$$

□

One very surprising result related to partial recursive functions is that every partial recursive function can be obtained by applying the minimization rule *at most once*, and even in this case it suffices to use a predicate p which is primitive recursive (not necessarily partial recursive). Our interest in the class of partial recursive functions is due to the following important result (which we shall present without proof).

THEOREM 1-6 (Kleene). *An n-ary partial function mapping n-tuples of words over* Σ *into words over* Σ *is partial recursive if and only if it is Turing computable.*

A partial recursive function that is total (defined for all arguments) is called a *total recursive function.* The relation between the various classes of functions over Σ (that is, mapping n-tuples of words over Σ into words over Σ) can be summarized as follows:

$$\{\text{All functions over } \Sigma\}$$
$$\cup$$
$$\{\text{Partial recursive functions}\} = \{\text{Turing computable functions}\}$$
$$\cup$$
$$\{\text{Total recursive functions}\}$$
$$\cup$$
$$\{\text{Primitive recursive functions}\}$$

All the above inclusions are proper inclusions. To show that the top inclusion is proper we consider the set $L_1 = \{x_i | x_i \text{ is not accepted by } T_i\}$. Since this set is not recursively enumerable (see Sec. 1-3.1), it follows that any function, which is defined for every $x \in L_1$ and undefined for every $x \notin L_1$,

is not Turing computable; therefore, by Theorem 1-6, this function is not partially recursive.

It is also straightforward that the class of partial recursive functions properly includes the class of all total recursive functions (see Example 1-21).

Ackermann's function $A(x)$ over Σ is the classic example of a total recursive function that is not primitive recursive. $A(x)$ is equal to $f(x, x)$, where f is defined recursively as follows [see, for example, Hermes (1967)]:

$$f(x_1, x_2) = \begin{cases} ax_2 & \text{if } x_1 = \Lambda \\ f(tail(x_1), a) & \text{if } x_1 \neq \Lambda \text{ and } x_2 = \Lambda \\ f(tail(x_1), f(x_1, tail(x_2))) & \text{if } x_1 \neq \Lambda \text{ and } x_2 \neq \Lambda \end{cases}$$

The main point in the proof that $A(x)$ is not a primitive recursive function is that for every unary primitive recursive function g there exists a word $w \in \Sigma^*$ such that $A(w)$ yields a longer word than that of $g(w)$, which implies that $A(x)$ cannot be a primitive recursive function.

1-5 TURING MACHINES AS ALGORITHMS

In mathematics we often consider classes of *yes/no problems*, i.e., problems for which the answer is always either "yes" or "no," and investigate whether or not there is any algorithm (computing device) for solving (or at least partially solving) all the problems in the class. By Church's thesis, a Turing machine can be considered to be the most general possible computing device, and this suggests formalizing the common vague notion of "solvability" of classes of yes/no problems by means of Turing machines.

We say that a class of yes/no problems is *solvable* (*decidable*) if there is some fixed algorithm (Turing machine) which for any problem in the class as input will determine the solution to the problem; i.e., the algorithm *always halts* with a correct "yes" or "no" answer. (A "yes" answer is obtained by reaching an ACCEPT halt, while a "no" answer is obtained by reaching a REJECT halt.) If no such algorithm (Turing machine) exists, we say that the class of problems is *unsolvable* (*undecidable*). Note, however, that the unsolvability of a class of problems does not mean that we cannot determine the answer to a specific problem in the class by a Turing machine.

An important task of theoreticians in computer science is to indicate unsolvable classes of yes/no problems to prevent computer scientists from looking for nonexisting algorithms. One of the most famous unsolvable classes of yes/no problems is *Hilbert's tenth problem*. In 1901 Hilbert†

† David Hilbert, "Mathematical Problems," *Bull. Am. Math. Soc.*, **8**: 437–479 (1901).

listed a group of problems which were to stand as a challenge to future generations of mathematicians. The tenth problem in this group is to find an algorithm that takes an arbitrary polynomial equation $P(x_1, \ldots, x_n) = 0$ with integer coefficients as input and determines whether or not the equation has a solution in integers. This was an open problem for about seventy years. In 1970, Matijasevič† showed that there cannot exist such an algorithm, in other words, that *Hilbert's tenth problem is unsolvable.*

Unfortunately many interesting classes of yes/no problems are unsolvable. This suggests introducing a weaker notion of algorithms which will only *partially* solve the problems in the class in the sense that for any problem in the class as input: if the answer to the problem is "yes," the algorithm will eventually halt with a "yes" answer, but if the answer is "no," the algorithm may supply no answer at all! More precisely, a class of yes/no problems is said to be *partially solvable* (*partially decidable*) if there is an algorithm (Turing machine) that will take any problem in the class as input and (1) if the answer to the problem is "yes," the algorithm will eventually reach an ACCEPT halt, but (2) if the answer to the problem is "no," the algorithm either reaches a REJECT halt or never halts. This definition of solvability is clearly weaker than the original one in the sense that if a class of yes/no problems is solvable, then it is also partially solvable. However, this weaker notion of solvability is very important because many unsolvable classes of problems can be shown to be partially solvable.

At this point the reader will probably wonder how we can give a yes/no problem as input to an algorithm (Turing machine). This is purely an encoding problem. We shall be very vague regarding this problem and comment only that we restrict our yes/no problems to those that can be represented by some suitable encoding as words over $\Sigma = \{a, b\}$.

1-5.1 Solvability of Classes of Yes/No Problems

There are several interesting classes of yes/no problems related to the material discussed previously in this chapter which are solvable. For example:

1. *The equivalence problem of finite automata is solvable*; that is, there is an algorithm that takes any pair of finite automata A and A' over $\Sigma = \{a, b\}$ as input and determines whether or not A is equivalent to A'.

2. *The word problem of context-sensitive grammars is solvable*; that is, there is an algorithm that takes any context-sensitive grammar G over

† Ju. V. Matijasevič, "Enumerable Sets are Diophantine," *Sov. Math. Dokl.*, 11(2):354–358 (1970).

$\Sigma = \{a, b\}$ and any word $w \in \Sigma^*$ as input and determines whether or not w is in the language generated by G.

In the following sections we present several classes of yes/no problems which are unsolvable. The first and most important unsolvable class of problems is known as the *halting problem of Turing machines.* It can be shown that there is no algorithm that takes an arbitrary Turing machine M over $\Sigma = \{a, b\}$ and an arbitrary word $w \in \Sigma^*$ as input and always determines whether or not M would halt if it were given input w. This result is proved later by contradiction showing that if there did exist such an algorithm, then we could conclude that there exists a Turing machine B over $\Sigma = \{a, b\}$ such that B halts for input $d(B)$ if and only if B does not halt for input $d(B)$, where $d(B) \in \Sigma^*$ is the encoded description of B as a word over Σ.

Using the unsolvability of the halting problem of Turing machines, we can prove many other classes of problems to be unsolvable by reducing the halting problem of Turing machines to those classes. More precisely, we say that a class $\mathscr{P}$ of yes/no problems is *reducible* to another class $\mathscr{P}'$ of yes/no problems if there is a Turing machine that takes any problem $P \in \mathscr{P}$ as input and yields some problem $P' \in \mathscr{P}'$ as output such that the answer to problem P is "yes" if and only if the answer to problem P' is "yes." We call such a Turing machine a *reduction Turing machine from* $\mathscr{P}$ *to* $\mathscr{P}'$.

Now, suppose we show that $\mathscr{P}$ is reducible to $\mathscr{P}'$. Then, if $\mathscr{P}'$ is solvable, we can combine the reduction Turing machine from $\mathscr{P}$ to $\mathscr{P}'$ and the algorithm for solving $\mathscr{P}'$ to form an algorithm for solving $\mathscr{P}$. Thus, *if* $\mathscr{P}$ *is reducible to* $\mathscr{P}'$ *and* $\mathscr{P}'$ *is solvable, so is* $\mathscr{P}$; in other words, if $\mathscr{P}$ *is reducible to* $\mathscr{P}'$ *and* $\mathscr{P}$ *is unsolvable, so is* $\mathscr{P}'$. In particular, since we already know that the halting problem (HP) of Turing machines is unsolvable, it follows that for every class $\mathscr{P}'$ of yes/no problems, if HP is reducible to $\mathscr{P}'$, then $\mathscr{P}'$ is unsolvable.

Using this reduction technique, we show two classes of yes/no problems to be unsolvable:

1. *The word problem of semi-Thue systems is unsolvable.*
2. *The Post correspondence problem is unsolvable.*

Note that if $\mathscr{P}$ is a class of yes/no problems and $\mathscr{P}'$ is a subclass of $\mathscr{P}$ (that is, every problem in $\mathscr{P}'$ is a problem in $\mathscr{P}$), then it is clear that if $\mathscr{P}$ is solvable, so is $\mathscr{P}'$, and if $\mathscr{P}'$ is unsolvable, so is $\mathscr{P}$.

1-5.2 The Halting Problem of Turing Machines

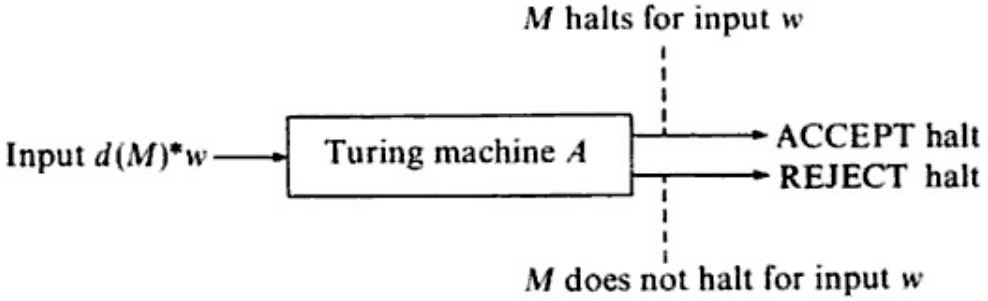
The HP of Turing machines over $\Sigma = \{a, b\}$ is actually a class of yes/no problems and can be stated as follows: Given an arbitrary Turing machine M over $\Sigma = \{a, b\}$ and an arbitrary word $w \in \Sigma^*$, does M halt for input w? We have the following theorem.

THEOREM 1-7 (Turing). *The halting problem of Turing machines is unsolvable.*

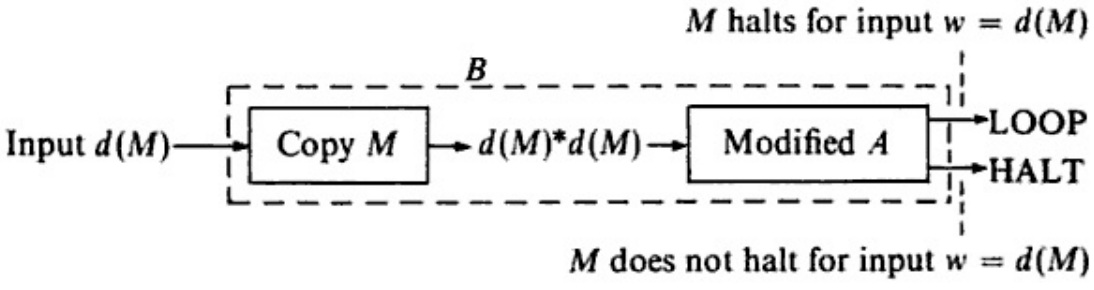
That is, there is no algorithm that takes an arbitrary Turing machine M over $\Sigma = \{a, b\}$ and word $w \in \Sigma^*$ as input and determines whether or not M halts for input w.

Proof (Minsky). Again we shall be vague regarding the encoding problem. We denote simply by $d(M)$ and $d(M) * w$ the encoded description of M and (M, w), respectively, as strings over $\Sigma = \{a, b\}$.

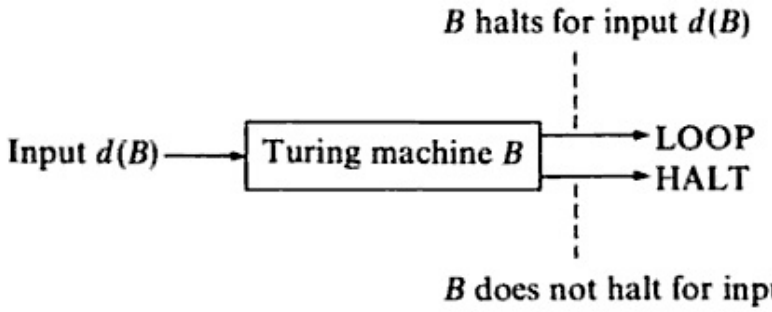
The proof is by contradiction. Suppose there exists such an algorithm (Turing machine), say, A. Then for every input $d(M) * w$ to A, we have: If M halts for input w, then A reaches an ACCEPT halt; if M does not halt for input w, then A reaches a REJECT halt. That is,



Now we can construct the Turing machine B which takes $d(M)$ as input and proceeds as follows: First, it copies the input to obtain $d(M) * d(M)$, and then it applies the Turing machine A on $d(M) * d(M)$ as input, with one modification—whenever A is supposed to reach an ACCEPT halt, instead B will loop forever. Considering the original behavior of A, we obtain



The above discussion holds true for an arbitrary Turing machine M over $\Sigma = \{a, b\}$. Since B itself is such a Turing machine, we let $M = B$; then, replacing B for M in the above figure, we obtain



Thus, B halts for input $d(B)$ if and only if B does not halt for input $d(B)$. Contradiction.

Q.E.D.

There are two interesting variations of the halting problem of Turing machines:

1. *The empty-word halting problem of Turing machines is unsolvable.* That is, there is no algorithm that takes an arbitrary Turing machinge M over $\Sigma = \{a, b\}$ as input and determines whether or not M halts for input Λ (the empty word).

2. *The uniform halting problem of Turing machines is unsolvable.* That is, there is no algorithm that takes an arbitrary Turing machine M over $\Sigma = \{a, b\}$ as input and determines whether or not M halts for *every* input.

Both results can be proved easily by the reduction technique.

1. We show that the halting problem of Turing machines is reducible to the *empty-word halting problem of Turing machines.* In this case the reduction algorithm takes any Turing machine M over $\Sigma = \{a, b\}$ and any word $w \in \Sigma^*$ as input and yields a Turing machine M' over $\Sigma = \{a, b\}$ which first generates the word w onto the tape and then applies M to it. Thus, M halts for input w if and only if M' halts for input Λ (the empty word).

2. We show that the halting problem of Turing machines is reducible to the *uniform halting problem of Turing machines.* In this case the reduction algorithm takes any Turing machine M over $\Sigma = \{a, b\}$ and any word $w \in \Sigma^*$ as input and yields a Turing machine M' over $\Sigma = \{a, b\}$ which first clears the input tape to Δ's, then generates the word w onto the tape, and finally applies M. Thus, M halts for input w if and only if M' halts for every input.

From the discussions in previous sections it is clear that one can show the following (by using either the proof of Theorem 1-7 or the reduction technique).

1. *The halting problem of Post machines is unsolvable.* That is, there is no algorithm that takes an arbitrary Post machine M over $\Sigma = \{a, b\}$ and word $w \in \Sigma^*$ as input and determines whether or not M halts for input w.

2. *The halting problem of finite machines with two pushdown stores is unsolvable.* That is, there is no algorithm that takes an arbitrary finite machine M over $\Sigma = \{a, b\}$ with two pushdown stores and word $w \in \Sigma^*$ as input and determines whether or not M halts for input w.

3. *The totality problem of partial recursive functions is unsolvable.* That is, there is no algorithm that takes a definition of a partial recursive function f over $\Sigma = \{a, b\}$ as input and determines whether or not f is total.

4. *The recursiveness problem of type-0 grammars is unsolvable.* That is, there is no algorithm that takes an arbitrary type-0 grammar G over $\Sigma = \{a, b\}$ as input and determines whether or not the language generated by G is recursive.

1-5.3 The Word Problem of Semi-Thue Systems

A *semi-Thue system* S *over* $\Sigma = \{a, b\}$ consists of a set of k, $k \geqq 1$, ordered pairs (α_i, β_i) of words over Σ; that is,

$$S = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_k, \beta_k)\}$$

For two words $x, y \in \Sigma^*$, we say that y *is derivable from* x *in* S if there exists a sequence of words over Σ

$$w_0, w_1, w_2, \ldots, w_n \qquad n \geqq 0$$

such that w_0 is x, w_n is y and w_{i+1} is obtained from w_i, $0 \leqq i < n$, by replacing some occurrence of a substring α_j (which is the left-hand side of some pair in S) in w_i by the corresponding β_j (which is the right-hand side of that pair).

The *word problem* (*WP*) *of semi-Thue systems over* $\Sigma = \{a, b\}$ is a class of yes/no problems which can be stated as follows: Given an arbitrary semi-Thue system S over $\Sigma = \{a, b\}$ and two arbitrary words $x, y \in \Sigma^*$, is y derivable from x in S?

EXAMPLE 1-22

Consider the semi-Thue system $S = \{(ba, ab), (aab, \Lambda)\}$ over $\Sigma = \{a, b\}$. Then Λ is derivable from $x \in \Sigma^*$ in S if and only if x has twice as many a's

as it has b's; for example, Λ is derivable from *baaaba* but not from *baaab*.

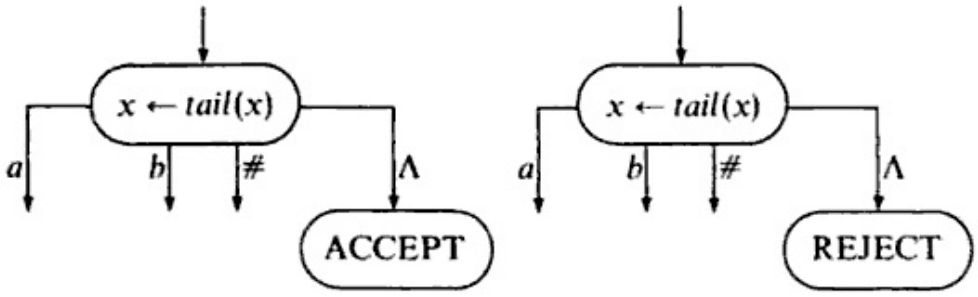
□

We have the following theorem.

THEOREM 1-8 (Post). *The word problem of semi-Thue systems over* $\Sigma = \{a, b\}$ *is unsolvable.*

That is, there is no algorithm that takes an arbitrary semi-Thue system S over $\Sigma = \{a, b\}$ and words $x, y \in \Sigma^*$ as input and determines whether or not y is derivable from x in S.

Proof. We show that the HP of Post machines is reducible to the WP of semi-Thue systems. For an arbitrary Post machine M over $\Sigma = \{a, b\}$ and input word $w \in \Sigma^*$, we construct a semi-Thue system S over $\Sigma = \{a, b\}$ and a pair of words $x, y \in \Sigma^*$ such that M halts for input w if and only if y is derivable from x in S. In this proof we make use of the fact that we can consider, without loss of generality, only Post machines where TEST statements must have the form



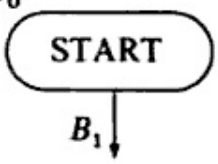
and there are no other HALT statements. That is, for every Post machine over Σ there exists an equivalent Post machine over Σ having only such TEST statements.

Suppose the Post machine M has m TEST and ASSIGNMENT statements labeled by $B_1, B_2, \ldots, B_m$. Let B_0 be the START statement. Suppose also that $w = \sigma_1\sigma_2 \ldots \sigma_n$. Then take

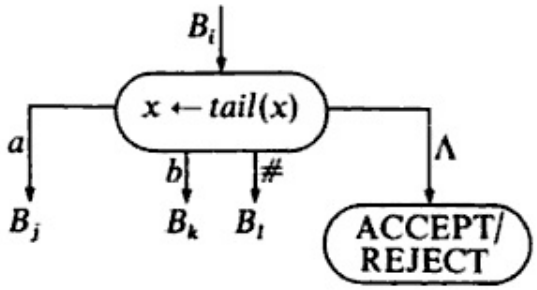
$$x = B_0 \vdash \sigma_1\sigma_2 \ldots \sigma_n \dashv$$
$$y = \Lambda$$

The list of ordered pairs of S is constructed as follows:

1. B_0 yields (B_0, B_1)

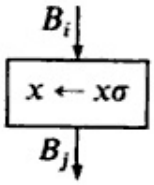


2.



yields $\begin{cases} (B_i \vdash a, B_j \vdash) \\ (B_i \vdash b, B_k \vdash) \\ (B_i \vdash \#, B_l \vdash) \\ (B_i \vdash \dashv, \Lambda) \end{cases}$

3.



yields $(\dashv B_i, \sigma \dashv B_j)$

where $\sigma \in \{a, b, \#\}$.

4. Cycling: $\begin{cases} (\sigma B_i, B_i\sigma) \\ (B_i\sigma, \sigma B_i) \end{cases}$

for all $\sigma \in \{a, b, \#, \vdash, \dashv\}$ and $1 \leqq i \leqq m$.

It is straightforward to see that M halts for input w if and only if y is derivable from x in S.

Note that the alphabet of the semi-Thue system S that we have constructed is actually $\{a, b, \#, \vdash, \dashv\} \cup \{B_0, B_1, \ldots, B_m\}$. However, with "suitable" encoding one can construct a semi-Thue system S' over $\Sigma = \{a, b\}$ and words $x', y' \in \Sigma^*$ such that M halts for input w if and only if y' is derivable from x' in S'.

Q.E.D.

1-5.4 Post Correspondence Problem

A *Post system* S *over* $\Sigma = \{a, b\}$ consists of a set of k, $k \geqq 1$, ordered pairs (α_i, β_i) of words over Σ; that is,

$$S = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_k, \beta_k)\}$$

A *solution to a Post system* is a nonempty sequence of integers $i_1, i_2, \ldots, i_m$

$(1 \leqq i_j \leqq k)$ such that

$$\alpha_{i_1}\alpha_{i_2} \cdots \alpha_{i_m} = \beta_{i_1}\beta_{i_2} \cdots \beta_{i_m}$$

The *Post correspondence problem* (*PCP*) *over* $\Sigma = \{a, b\}$ is a class of yes/no problems which can be stated as follows: Given an arbitrary Post system over $\Sigma = \{a, b\}$, does it have a solution?

EXAMPLE 1-23

1. Let $\Sigma = \{a, b\}$. The Post system

$$S = \{(b, bbb), (babbb, ba), (ba, a)\}$$

has a solution $i_1 = 2$, $i_2 = 1$, $i_3 = 1$, and $i_4 = 3$ (that is, $m = 4$) since

$$\underbrace{b\ a\ b\ b\ b\ b\ b\ a}_{\alpha_2\ \alpha_1\alpha_1\ \alpha_3} = \underbrace{b\ a\ b\ b\ b\ b\ b\ a}_{\beta_2\ \beta_1\ \beta_1\ \beta_3}$$

2. Let $\Sigma = \{a, b\}$. The Post system

$$S = \{(ab, abb), (b, ba), (b, bb)\}$$

has no solution. In each pair α_i is shorter than β_i; therefore, for any nonempty sequence of integers the string of α_i's will be shorter than the corresponding string of β_i's. □

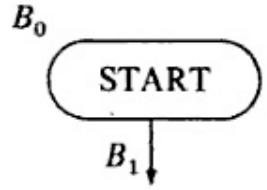
We have the following theorem.

THEOREM 1-9 (Post). *The Post correspondence problem over* $\Sigma = \{a, b\}$ *is unsolvable.*

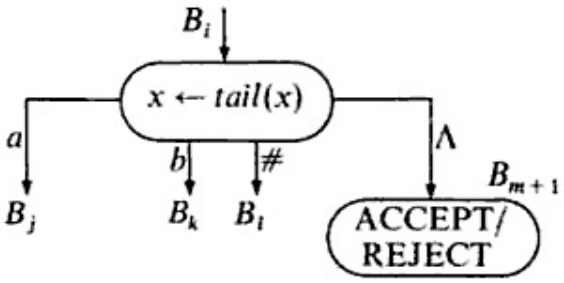
That is, there is no algorithm that takes arbitrary Post system over $\Sigma = \{a, b\}$ as input and determines whether or not it has a solution.

Proof (Scott). We show that the HP of Post machines is reducible to the PCP. For an arbitrary Post machine M over $\Sigma = \{a, b\}$ and input word $w \in \Sigma^*$, we construct a Post system S over $\Sigma = \{a, b\}$ such that M halts for input w if and only if S has a solution.

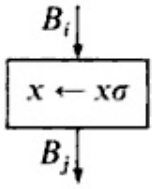
Suppose the Post machine M has m TEST and ASSIGNMENT statements labeled by $B_1, B_2, \ldots, B_m$ (see the proof of Theorem 1-8). Let B_0 be the START statement, and let B_{m+1} stand for any ACCEPT or REJECT halt. Suppose also that $w = \sigma_1\sigma_2 \cdots \sigma_n$. Then the corresponding Post system S over the alphabet $\{a, b, \#, *, B_0, \ldots, B_{m+1}\}$ is

1. 

yields $(B_0, B_0*\sigma_1*\sigma_2* \ldots *\sigma_n*B_1*)$

2. 

yields $\begin{cases}(*B_i*a, B_j*)\\(*B_i*b, B_k*)\\(*B_i*\#, B_l*)\\(*B_i*B_{m+1}, B_{m+1})\end{cases}$

3. 

yields $(*B_i, \sigma*B_j*)$

where $\sigma \in \{a, b, \#\}$.

4. Cycling: $(*\sigma, \sigma*)$ for all $\sigma \in \{a, b, \#\}$.

It should now be verified that a concatenation of words from the first members of these pairs can equal a corresponding concatenation from the second members if and only if it transcribes word by word a terminating computation of the Post machine with input w. The terminating computation is described as a sequence of B_i's. Between two successive B_i's and B_j's, we have a string over $\{a, b, \#\}$ that represents the current value of x after the execution of B_i and before the execution of B_j. All letters are separated by $*$; the role of the $*$'s is to ensure that the current value of x will be copied between the B_i's.

Q.E.D.

The unsolvability of the Post correspondence problem is very often used to prove the unsolvability of other classes of yes/no problems. In particular, the Post correspondence problem can be reduced to many classes of yes/no problems in formal language theory, concluding, therefore, that they are unsolvable. For example, the following can be shown.

1. *The equivalence problem of context-free (type-2) grammars is unsolvable.* That is, there is no algorithm that takes any two context-free grammars over $\Sigma = \{a, b\}$ as input and determines whether or not they generate the same language. Note that the equivalence problem of regular (type-3) grammars is solvable.

2. *The emptiness problem of context-sensitive (type-1) grammars is unsolvable.* That is, there is no algorithm that takes any context-sensitive grammar G over $\Sigma = \{a, b\}$ as input and determines whether or not the language generated by G is empty. Note that the emptiness problem of context-free (type-2) grammars is solvable.

3. *The word problem of type-0 grammars is unsolvable.* That is, there is no algorithm that takes any type-0 grammar G over $\Sigma = \{a, b\}$ and word $w \in \Sigma^*$ as input and determines whether or not w is in the language of G. Note that the word problem of context-sensitive (type-1) grammars is solvable.

EXAMPLE 1-24 (Floyd)

A finite (nonempty) set of 3×3 matrices $\{M_i\}$ over the integers is said to be *mortal* with respect to $\langle j_1, j_2 \rangle$, $1 \leqq j_1, j_2 \leqq 3$, if there is a finite product of members of the set, $M = M_{i_1} \cdot M_{i_2} \cdot \ldots \cdot M_{i_k}$, such that the $\langle j_1, j_2 \rangle$ element of M is 0. We shall show that *the mortality problem of matrices is unsolvable*; that is, there is no algorithm that takes a finite set of 3×3 integer matrices $\{M_i\}$ and a pair $\langle j_1, j_2 \rangle$, $1 \leqq j_1, j_2 \leqq 3$, as input and decides whether or not $\{M_i\}$ is mortal with respect to $\langle j_1, j_2 \rangle$.

We prove the unsolvability of the mortality problem by reducing the Post correspondence problem to the mortality problem; i.e., for an arbitrary Post system S over Σ we construct a set of 3×3 integer matrices $\{M_i\}$ in such a way that S has a solution if and only if $\{M_i\}$ is mortal with respect to $\langle 3, 2 \rangle$. The idea is to construct for a given pair of words (u, v) over Σ a 3×3 integer matrix $M(u, v)$ in such a way that

1. The $\langle 3, 2 \rangle$ element of $M(u, v)$ is 0 if and only if $u = v$.
2. For all words $u_1, u_2, v_1, v_2 \in \Sigma^*$

$$M(u_1, v_1) \cdot M(u_2, v_2) = M(u_1 u_2, v_1 v_2)$$

Now, for a given Post system $S = \{(\alpha_i, \beta_i)\}$ over Σ, we construct a set of 3×3 integer matrices $\{M_i\}$ where each $M_i = M(\alpha_i, \beta_i)$. Then from properties 1 and 2 above, it is clear that S has a solution $i_1, i_2, \ldots, i_k$ if and only if the $\langle 3, 2 \rangle$ element of $M_{i_1} \cdot M_{i_2} \cdot \ldots \cdot M_{i_k}$ is 0.

To define the matrix $M(u, v)$ we use two functions c (code) and el (exponent of length) mapping words over Σ into integers:

$c(x) = i$ iff x is the ith word in the natural lexicographic ordering
$el(x) = n^{|x|}$ where $|x|$ is the length of x and n is the number of letters in Σ

It is easy to prove that

$$c(xy) = c(x) \cdot el(y) + c(y)$$
$$el(xy) = el(x) \cdot el(y)$$

Now, for a pair (u, v) of words over Σ, we define

$$M(u, v) = \begin{bmatrix} el(u) & el(v) - el(u) & 0 \\ 0 & el(v) & 0 \\ c(u) & c(v) - c(u) & 1 \end{bmatrix}$$

$M(u, v)$ has the desired properties:

1. The $\langle 3, 2 \rangle$ element of $M(u, v)$ is 0 iff $u = v$ since $c(v) - c(u) = 0$ iff $u = v$.

2. For all words $u_1, u_2, v_1, v_2 \in \Sigma^*$, $M(u_1, v_1) \cdot M(u_2, v_2) = M(u_1u_2, v_1v_2)$ since

$$M(u_1, v_1) \cdot M(u_2, v_2)$$
$$= \begin{bmatrix} el(u_1) & el(v_1) - el(u_1) & 0 \\ 0 & el(v_1) & 0 \\ c(u_1) & c(v_1) - c(u_1) & 1 \end{bmatrix} \cdot \begin{bmatrix} el(u_2) & el(v_2) - el(u_2) & 0 \\ 0 & el(v_2) & 0 \\ c(u_2) & c(v_2) - c(u_2) & 1 \end{bmatrix}$$

$$= \begin{bmatrix} el(u_1)\,el(u_2) & el(v_1)\,el(v_2) - el(u_1)\,el(u_2) & 0 \\ 0 & el(v_1)\,el(v_2) & 0 \\ c(u_1)\,el(u_2) + c(u_2) & [c(v_1)\,el(v_2) + c(v_2)] - [c(u_1)\,el(u_2) + c(u_2)] & 1 \end{bmatrix}$$

$$= M(u_1u_2, v_1v_2)$$

□

1-5.5 Partial Solvability of Classes of Yes/No Problems

It is quite interesting to note that although the classes of yes/no problems discussed in the previous sections are unsolvable, many of them are partially solvable. For example, the *halting problem of Turing machines is partially solvable.* That is, there is an algorithm (Turing machine) that takes any Turing machine M over $\Sigma = \{a, b\}$ and any word $w \in \Sigma^*$ as input and

(1) if M halts for w, the algorithm will eventually reach an ACCEPT halt, but (2) if M does not halt for w, the algorithm either reaches a REJECT halt or never halts. The standard proof of this result proceeds by constructing an algorithm (Turing machine) that takes an arbitrary Turing machine M over $\Sigma = \{a, b\}$ and a word $w \in \Sigma^*$ as input and simulates the behavior of M for input w. We call such a Turing machine a *universal Turing machine.* We can think of a universal Turing machine as a general-purpose computer which is powerful enough to simulate any computer, including itself.

The relationship between the solvability and partial solvability of a class $\mathscr{P}$ of yes/no problems is best described by means of the complement class $\bar{\mathscr{P}}$, that is, the class of yes/no problems obtained from $\mathscr{P}$ just by taking the negation of each problem in the class. Then the key result can be stated as follows: *$\mathscr{P}$ is solvable if and only if both $\mathscr{P}$ and $\bar{\mathscr{P}}$ are partially solvable.* It is clear that if $\mathscr{P}$ is solvable, then so is $\bar{\mathscr{P}}$ (interchange all the ACCEPT and REJECT halts in the given algorithm), and therefore if $\mathscr{P}$ is solvable, both $\mathscr{P}$ and $\bar{\mathscr{P}}$ are partially solvable. To show the result in the other direction, let us assume that algorithms A and $\bar{A}$ partially solve $\mathscr{P}$ and $\bar{\mathscr{P}}$, respectively. Then construct an algorithm B which simulates the operations of both A and $\bar{A}$.† Whenever A reaches an ACCEPT halt, so will B; but whenever $\bar{A}$ reaches an ACCEPT halt, B is modified to reach a REJECT halt. Thus, for every problem of $\mathscr{P}$ as input, if the answer to the problem is "yes," B will reach an ACCEPT halt and if the answer to the problem is "no," B will reach a REJECT halt; in other words, B solves the problems in $\mathscr{P}$.

Since the halting problem of Turing machines is unsolvable but is partially solvable, an important consequence of the above result is that its complement is not partially solvable; i.e., the *nonhalting problem of Turing machines is not partially solvable.* That is, there is no algorithm (Turing machine) that will take any Turing machine M over $\Sigma = \{a, b\}$ and word $w \in \Sigma^*$ as input and (1) if M *does not halt* for w, the algorithm will reach an ACCEPT halt, but (2) if M halts for w, the algorithm either reaches a REJECT halt or never halts.

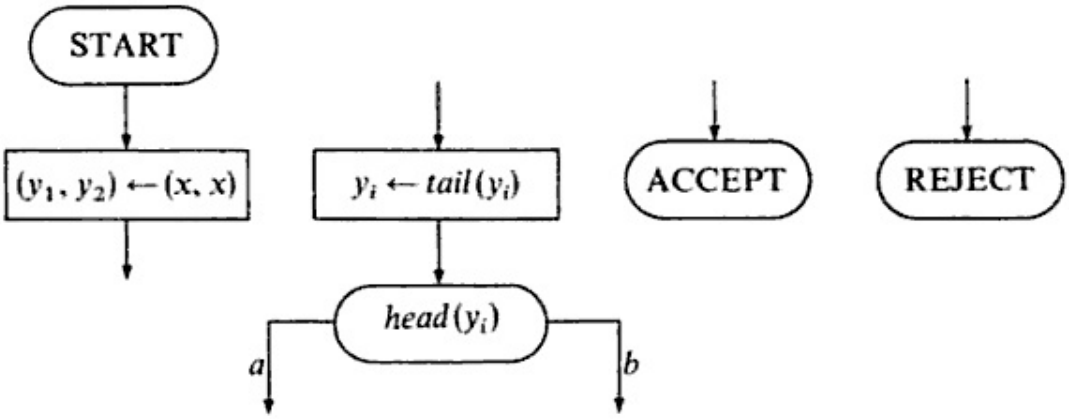
The reduction technique for solvability can be used for partial solvability as well because we have that *if a class $\mathscr{P}$ of yes/no problems is reducible to $\mathscr{P}'$ and $\mathscr{P}'$ is partially solvable, then so is $\mathscr{P}$*; in other words, *if a class $\mathscr{P}$ of yes/no problems is reducible to $\mathscr{P}'$ and $\mathscr{P}$ is not partially solvable, then neither is $\mathscr{P}'$.* We shall illustrate this technique with one example. (An important

† The simulations of A and $\bar{A}$ are done "in parallel" so that neither requires the termination of the other in order to have its behavior completely simulated.

application of the results presented in this example is discussed in Chap. 4, Sec. 4-2.1.)

EXAMPLE 1-25 (Paterson)

Let $\Sigma = \{a, b\}$. A *two-registered finite automaton* A over Σ is a machine with domain Σ^∞ (the set of all *infinite* tapes of letters from Σ) and is described by a flow-diagram such that each statement in it is of one of the following forms:



For $i = 1, 2$

where, for every $\sigma \in \Sigma$ and $w \in \Sigma^\infty$, $tail(\sigma w) = w$ and $head(\sigma w) = \sigma$. Note that the first letter of x is removed by the first application of *tail*.

For each input tape $\xi \in \Sigma^\infty$ for x, we distinguish between three possible computations: (1) If ACCEPT is reached, the tape is said to be *accepted*; (2) if REJECT is reached, the tape is said to be *rejected*; (3) otherwise, the automaton A is said to *loop* on the tape. Thus, a two-registered finite automaton A determines a threefold partition of Σ^∞ into $accept(A)$, $reject(A)$, and $loop(A)$, the classes of tapes of Σ^∞ which A accepts, rejects, and on which it loops, respectively.

We show the following results to be true.

1. *The nonacceptance problem of two-registered finite automata over* $\Sigma = \{a, b\}$ *is not partially solvable.* That is, there is no algorithm that takes any two-registered finite automaton A over $\Sigma = \{a, b\}$ as input and (1) if $accept(A) = \phi$, the algorithm will reach an ACCEPT halt, but (2) if $accept(A) \neq \phi$, the algorithm either reaches a REJECT halt or never halts.

2. *The looping problem of two-registered finite automata over* $\Sigma = \{a, b\}$ *is not partially solvable.* That is, there is no algorithm that takes any two-registered finite automaton A over $\Sigma = \{a, b\}$ as input and (1) if $loop(A) \neq \phi$,

the algorithm will reach an ACCEPT halt, but (2) if $loop(A) = \phi$, the algorithm either reaches a REJECT halt or never halts.

Both results are proved by reducing the nonhalting problem of Turing machines to the nonacceptance and the looping problems of two-registered finite automata.

Given a Turing machine M and an input word w over $\Sigma = \{a, b\}$, there is an effective method [see Paterson (1968) and Luckham, Park, and Paterson (1970)] for constructing a two-registered finite automaton $A(M, w)$ over Σ which checks whether or not its input tape $\xi \in \Sigma^\infty$ describes a computation of M starting with input w. In other words, A accepts a tape $\xi \in \Sigma^\infty$ if and only if ξ has a finite initial segment which describes a complete computation of M starting with w; A loops on the tape if and only if ξ describes a nonterminating computation of M starting with w; otherwise, A rejects the tape ξ. Roughly speaking, this is done by assuming ξ to be of the form $\alpha_0\alpha_1\alpha_2\alpha_3 \ldots$, where each $\alpha_i \in \Sigma^*$ indicates the *configuration* of M (contents of tape, position of reading head, and current state) after the ith step of the computation. The checking is done by letting y_2 "read" along the string α_i while y_1 "verifies" that the following string α_{i+1} is indeed the next configuration of the computation of M starting with w.

Then, the automaton $A(M, w)$ has the following property: M fails to halt starting with input w if and only if $accept(A) = \phi$, or, equivalently, if and only if $loop(A) \neq \phi$. Now, since the nonhalting problem of Turing machines is not partially solvable, neither are the nonacceptance and looping problems of two-registered finite automata.

□

Bibliographic Remarks

Three excellent books on computability are those of Minsky (1967), Hopcroft and Ullman (1969), and Kain (1972). There are several more advanced books such as those of Davis (1958), Hermes (1965), and Rogers (1967). Our presentation benefitted from the excellent papers of Shepherdson and Sturgis (1963) and Scott (1967).

The notion of finite state devices is attributed to McCulloch and Pitts (1943). An important paper discussing this topic is the one by Rabin and Scott (1959). The notion of regular expressions and their relation to regular sets (Theorem 1-1) is due to Kleene (1956). The equivalence of regular sets (Theorem 1-2) was first discussed by Moore (1956). An excellent exposition of regular sets is given by Ginzburg (1968) and Salomaa (1969) [see also the selected papers in Moore (1964)].

The basic notion of computability by Turing machines appears in

Turing's pioneer paper (1936). Our Post machine (Theorem 1-3) is very similar to Post's normal system (1936). However, it was not defined by Post, but by Arbib (1963) and independently by Shepherdson and Sturgis (1963). The power of finite machines with two pushdown stores (Theorem 1-4) was first discovered by Minsky (1961) [see also Evey (1963)]. Formal languages are discussed in detail by Hopcroft and Ullman (1969). The relation between recursively enumerable sets and type-0 languages (Theorem 1-5) was first given by Chomsky (1959). The notion of partial recursive functions (Theorem 1-6) was first introduced by Kleene (1936).

The unsolvability of the halting problem of Turing machines (Theorem 1-7) is given in Turing (1936); our proof is based on the exposition of Minsky (1967). The unsolvability of the semi-Thue system (Theorem 1-8) is due to Post (1947). Post's correspondence problem (Theorem 1-9) was first formulated and shown to be unsolvable by Post (1946); our proof is due to Scott (1967).

REFERENCES

Arbib, M. A. (1963): "Monogenic Normal Systems are Universal," *J. Aust. Math. Soc.*, **3**:301–306.

Chomsky, N. (1959): "On Certain Formal Properties of Grammars," *Inf. & Control*, **2**(2):137–167.

Church, A. (1936): "An Unsolvable Problem of Elementary Number Theory," *Am. J. Math.*, **58**:345–363.

Davis, M. (1958): *Computability and Unsolvability*, McGraw-Hill Book Company, New York.

——— (1965): *The Undecidable*, Raven Press, Hewlett, N.Y.

Evey, R. J. (1963): "The Theory and Applications of Pushdown Store Machines," Ph.D. thesis, Harvard University, Cambridge, Mass.

Ginzburg, A. (1968): *Algebraic Theory of Automata*, Academic Press, Inc., N.Y.

Hermes, H. (1965): *Enumerability, Decidability, Computability*, Academic Press, Inc., New York.

Hopcroft, J. E., and J. D. Ullman (1969): *Formal Languages and their Relation to Automata*, Addison-Wesley Publishing Company, Inc., Reading, Mass.

Kain, R. Y. (1972): *Automata Theory, Machines and Languages*, McGraw-Hill Book Company, New York.

Kleene, S. C. (1936): "General Recursive Functions of Natural Numbers," *Mathematisch Ann.*, **112**: 727–742.

——— (1956): "Representation of Events in Nerve Nets and Finite Automata," in C. E. Shannon and J. McCarthy (eds.), *Automata Studies*, pp. 3–42, Princeton University Press, Princeton, N.J.

Luckham, D. C., D. M. R. Park, and M. S. Paterson (1970): "On Formalized Computer Programs," *J. CSS*, **4**(3): 220–249.

McCulloch, W. S. and W. Pitts (1943): "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bull. Math. Biophys.*, **5**: 115–133.

Minsky, M. L. (1961): "Recursive Unsolvability of Post's Problem of 'Tag' and Other Topics in Theory of Turing Machines," *Ann. Math.*, **74**(3): 437–454.

——— (1967): *Computation: Finite and Infinite Machines*, Prentice-Hall, Inc., Englewood Cliffs, N.J.

Moore, E. F. (1956): "Gedanken Experiments on Sequential Machines," in C. E. Shannon and J. McCarthy (eds.), *Automata Studies*, pp. 129–153, Princeton University Press, Princeton, N.J.

——— (1964): *Sequential Machines: Selected Papers*, Addison-Wesley Publishing Company, Inc., Reading, Mass.

Paterson, M. S. (1968): "Program Schemata," in D. Michie (ed.), *Machine Intelligence 3*, pp. 19–31, Edinburgh University Press, Edinburgh.

Post, E. L. (1936): "Finite Combinatory Processes—Formulation, I," *Symb. Logic*, **1**:103–105.

——— (1946): "A Variant of a Recursively Unsolvable Problem," *Bull. Am. Math. Soc.*, **52**(4): 264–268.

——— (1947): "Recursive Unsolvability of a Problem of Thue," *Symb. Logic*, **12**:1–11.

Rabin, M. and D. Scott (1959): "Finite Automata and Their Decision Problems," *IBM J. Res. & Dev.*, **3**(2): 114–125. [Reprinted in Moore (1964).]

Rogers, H. (1967): *Theory of Recursive Functions and Effective Computability*, McGraw-Hill Book Company, New York.

Salomaa, A. (1969): *Theory of Automata*, Pergamon Press, New York.

Scott, D. (1967): "Some Definitional Suggestions for Automata Theory," *J. CSS*, **1**(2): 187–212.

Shepherdson, J. C. and H. E. Sturgis (1963): "Computability of Recursive Functions," *J. ACM*, **10**:217–255.

Turing, A. M. (1936): "On Computable Numbers, With an Application to the Entscheidungsproblem," *Proc. Lond. Math. Soc., Ser. 2*, **42**:230–265; correction, *ibid.*, **43**:544–546 (1937).

PROBLEMS

Prob. 1-1 Prove the following identities for any regular expressions R and S over Σ:

(*a*) $R^* = R^*R^* = (R^*)^* = (\Lambda + R)^* = \Lambda + RR^*$

(*b*) $(R + S)^* = (R^* + S^*)^* = (R^*S^*)^* = (R^*S)^* R^* = R^*(SR^*)^*$

(*c*) $R^*R = RR^*$

(*d*) $(R^*S)^* = \Lambda + (R + S)^* S$

Prob. 1-2 Let $E(R_1, \ldots, R_n)$ stand for any regular expression over Σ containing regular expressions $R_1, \ldots, R_n$ in addition to '*', '·' and '+'. Prove the following:

(*a*) $E(R_1, \ldots, R_n) + (R_1 + \cdots + R_n)^* = (R_1 + \cdots + R_n)^*$

(*b*) $(R_1 + \cdots + R_n + E(R_1, \ldots, R_n))^* = (R_1 + \cdots + R_n)^*$

Prob. 1-3 Use the identities given in Sec. 1-1.1 to prove the following:

(*a*) $\Lambda + b^*(abb)^*\,[b^*(abb)^*]^* = (b + abb)^*$

(*b*) $(ba)^* b + (ba)^* (bb + a)\,[a + b(ba)^* (bb + a)]^* b(ba)^* b =$
$[ba + (bb + a)a^*b]^*b$
Hint: Show $[a + b(ba)^* (bb + a)]^* b = a^*b\,[(ba)^* (bb + a)\, a^*b]^*$

(*c*) $(a + b + aa)^* = (a + b)^*$
Hint: Show $(a + b + aa)^* = [(a + b)^* + aa]^* = (a + b)^*$.

(*d*) $[(b^*a)^* ab^*]^* = \Lambda + a(a + b)^* + (a + b)^* aa(a + b)^*$
Hint: Show $[a + b + (a + b)^* aa]^* = (a + b)^*$.

***Prob. 1-4** [One-dimensional checkers (Reynolds)]. Consider a one-dimensional (infinite) checkerboard where each square contains the letter a or b, for example,

· · ·	*a*	*a*	*b*	*a*	*a*	*b*	*b*	*b*	*a*	*a*	*b*	*a*	*a*	*a*	· · ·

There is always some word, starting and ending with b, (*baabbbaab* in the example above) with an infinite string of a's in both directions. A *move* in the game consists of jumping a b over a neighbor b to an a square, as follows:

Before move	After move
$\overset{\frown}{b\ b\ a}$	$a\ a\ b$
or	
$\overset{\frown}{a\ b\ b}$	$b\ a\ a$

An *initial board* consists of a word w (starting and ending with b). w is said to be a *winning word* if there is a finite sequence of moves that will lead to a *final board* consisting of one b (and the rest a's).

(a) Express the set of all winning words as a regular expression over $\{a, b\}$.
Hint: Consider the final board and play *backwards*.

(b) Prove your result.

***Prob. 1-5** [Regular equations (Arden†)].

(a) Let r be a variable, and let S and T denote regular expressions over Σ. Consider the equation

$$r = Sr + T$$

A set $X \subseteq \Sigma^*$ is said to be a *fixpoint* (*solution*) of the equation if $X = \tilde{S}X \cup \tilde{T}$. In general, an equation may have several fixpoints. A fixpoint X of the equation is said to be a *least fixpoint* if for every fixpoint Y of the equation, $X \subseteq Y$. Clearly, there can be at most one least fixpoint.

(i) Show that if $\Lambda \notin \tilde{S}$ there is a unique fixpoint given by $\widetilde{S^*T}$.

(ii) Show that every equation has a (unique) least fixpoint given by $\widetilde{S^*T}$.

(iii) Find the fixpoints of the equation $r = (ba)^* r + b$ over $\Sigma = \{a, b\}$.

(b) The above definition can be extended to systems of equations

$$r_i = S_{i1}r_1 + S_{i2}r_2 + \cdots + S_{in}r_n + T_i$$

where $1 \leqq i \leqq n$. An n-tuple of sets $\overline{X} = (X_1, \ldots, X_n)$ is said to be a *fixpoint of the system* if

$$X_i = \tilde{S}_{i1}X_1 \cup \tilde{S}_{i2}X_2 \cup \ldots \cup \tilde{S}_{in}X_n \cup \tilde{T}_i$$

for $1 \leqq i \leqq n$. A fixpoint $\overline{X} = (X_1, \ldots, X_n)$ is said to be a *least fixpoint of the system* if for every fixpoint $\overline{Y} = (Y_1, \ldots, Y_n)$, $X_i \subseteq Y_i$ for $1 \leqq i \leqq n$.

(i) Show that if $\Lambda \notin \tilde{S}_{11} \cup \tilde{S}_{12} \cup \ldots \cup \tilde{S}_{nn}$, then there is a unique fixpoint for which each X_i is regular.

(ii) Use this result to prove that for every finite automaton A over Σ there exists a regular expression R such that $\tilde{R} = \tilde{A}$.

† D. N. Arden, "Delayed logic and finite state machines," in *Theory of Computing Machine Design*, pp. 1–35, University of Michigan Press, Ann Arbor, 1960.

(iii) Show that every system of equations has a (unique) least fixpoint $\bar{X} = (X_1, \ldots, X_n)$ such that each X_i, $1 \leqq i \leqq n$, is regular.

(iv) Find the least fixpoint of the system

$$r_1 = (a + b)\, r_1 + (bb)^* r_2 + b$$
$$r_2 = (aa)^* r_1 + (a + b)\, r_2 + a$$

Prob. 1-6 Write finite automata that accept the following regular expressions:

(*a*) $\Lambda + a(a + b)^* + (a + b)^* aa(a + b)^*$

(*b*) $[ba + (a + bb)\, a^*b]^*$

(*c*) $(ba)^*\,(ab)^*\,(aa + bb)^*$

Prob. 1-7 Suppose that L is a regular set over Σ. Prove that the following are also regular sets:

(*a*) $rev(L) = \{w | w \text{ reversed is in } L\}$

(*b*) $init(L) = \{w_1 | w_1 w_2 \in L \text{ for some } w_2 \in \Sigma^*\}$

(*c*) $comp(L) = \{w | w \in \Sigma^* \text{ and } w \notin L\}$

(*d*) $lefthalf(L) = \{w_1 | w_1 w_2 \in L \text{ for some } w_2 \in \Sigma^* \text{ where } |w_1| = |w_2|\}$

(*e*) $righthalf(L) = \{w_2 | w_1 w_2 \in L \text{ for some } w_1 \in \Sigma^* \text{ where } |w_1| = |w_2|\}$

Suppose L_1, L_2 are regular sets over Σ. Prove that the following is also a regular set.

(*f*) $and(L_1, L_2) = \{w | w \in L_1 \text{ and } w \in L_2\}$

Prob. 1-8 Find regular expressions, finite automata, and simple transition graphs representing the following regular sets over $\Sigma = \{a, b\}$:

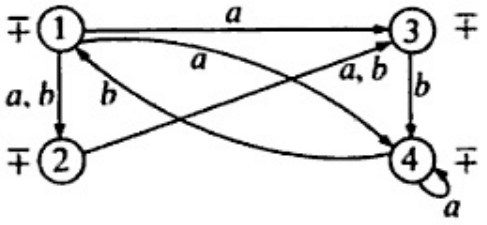
(*a*) All words in Σ^* such that every a has a b immediately to its right

(*b*) All words in Σ^* that do not contain two consecutive b's

Prob. 1-9

(*a*) Describe an algorithm to determine whether or not an arbitrary transition graph T accepts every word $w \in \Sigma^*$, that is, $\tilde{T} = \Sigma^*$.

(*b*) Does the following transition graph over $\Sigma = \{a, b\}$ accept Σ^*?



Prob. 1-10

(*a*) Consider the set L of words over $\Sigma = \{a, b\}$

$$L = \{a^n b a^n | n \geqq 0\}$$

(i) Construct a Turing machine M_1, Post machine M_2, and finite machine M_3 (with one pushdown store) such that $accept(M_i) = L$ and $reject(M_i) = \Sigma^* - L$ for $1 \leqq i \leqq 3$.

(ii) Prove that there is no finite machine M with no pushdown stores such that $accept(M) = L$ and $reject(M) = \Sigma^* - L$.

(*b*) Do the same for $L = \{w | w$ consists of twice as many a's as b's$\}$.

Prob. 1-11 We define two different classes of boolean expressions.

(*a*) A *boolean expression* (be) is defined recursively as follows:

(i) T and F are be's.

(ii) If A and B are be's, so are $A \vee B$ and $A \wedge B$ (no parentheses!).

The computation rules of be's are

$$T \wedge T = T \qquad T \wedge F = F \wedge T = F \wedge F = F$$
$$F \vee F = F \qquad F \vee T = T \vee F = T \vee T = T$$

We assume that $\wedge$ is more binding than $\vee$. Construct a Turing machine M_1, Post machine M_2, and finite machine M_3 (with no pushdown stores) over $\Sigma = \{T, F, \wedge, \vee\}$ such that

$$accept(M_i) = \text{all be's with value } T$$
$$reject(M_i) = \text{all be's with value } F$$
$$loop(M_i) = \text{all words over } \Sigma \text{ which are not be's}$$

for $1 \leqq i \leqq 3$.

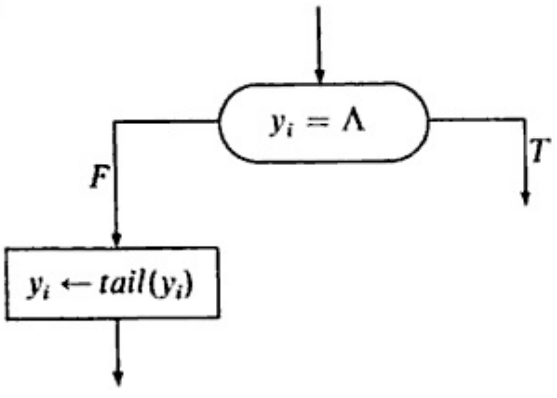
(*b*) Similarly construct a Turing machine M_1, Post machine M_2, and finite machine M_3 (with one pushdown store) over $\Sigma = \{T, F, \wedge, \vee, (,)\}$ for the following class of boolean expressions (be's):

(i) T and F are be's.

(ii) If A and B are be's, so are $A \vee B$ and $A \wedge B$.

(iii) If A is a be, so is (A).

***Prob. 1-12** (Finite machines with counters). A variable y_i is said to be a *counter over* $\Sigma = \{a, b\}$ if all the operations applied to y_i are of the form



and