2?/6-3-76
20 cy DNTLS

UCID-17045

Lawrence Livermore Laboratory

INTRODUCTION TO THE THEORY OF MACHINES AND LANGUAGES

Patrick P. Weidhaas

MASTER

April 1, 1976

Prepared for U.S. Energy Research & Development Administration under contract No. W-7405-Eng-48.

PREFACE

This set of notes grew out of an insane desire to close a gap which I felt existed in my education as a programmer. Like many other programmers, I initially received an education in mathematics when "computer science" was barely beginning to be recognized as an independent scientific discipline. Completion of my mathematical studies was followed by a transition into the new and vast (and more lucrative) world of computer-programming. Here, while reading computer-related literature, I frequently stumbled over such bizarre concepts as "Turing machines," "context-free grammars," and "finite-state automata" -- concepts with which I simply could not associate any meaning. Their persistent recurrence in the literature increased my feeling of ignorance until I finally reached a point of decision. I decided to systematically study the appropriate theory.

So, in the spring of 1973 I got the books by Minsky and Kaine (mentioned in the bibliography at the end of these notes). Fortunately, both books did an excellent job of pointing out the relevant concepts and ideas without losing themselves in complicated and irrelevant details. However, to enhance my understanding I copied down many of the more important parts of the theory in my own words. After three months I suddenly discovered that I had written an extensive set of notes, over 200 handwritten pages. Looking back over the notes, I felt that the material they contained would lend itself quite well to a course whose intent would be to "close the educational gap" experienced by other programmers who had started out like myself in a non-computer-science field.

As a result, in the winter quarter 1976 I taught a course entitled "Introduction to thc Theory of Machines and Languages" as part of the continuing education program at LLL. I used Minsky's book as a text, and also handed out my notes which I followed closely. The 22 lectures were recorded on videotape and can be presented again at any time.

[illegible]

THIS PAGE

WAS INTENTIONALLY

LEFT BLANK

CONTENTS

INTRODUCTION TO THE THEORY OF MACHINES AND LANGUAGES

ABSTRACT

This text is intended to be an elementary "guided tour" through some basic concepts of modern computer science. Various models of computing machines and formal languages are studied in detail. Discussions center around questions such as, "What is the scope of problems that can or cannot be solved by computers?"

REVIEW OF SET THEORY

Intuitively, a *set* S is simply a collection of different objects. The objects in this collection are called the *elements* or *members* of the set S.

Some Terminology

Examples:

(1) A = the set of all CDC-7600's at LLL in January 1976. A is a set containing three elements, the R- , S- , and U-machines.

(2) B = the set of all stars in the entire universe.

(3) C = the set of all integers greater than 1000.

(4) D = the set of all programmers who have never been frustrated.

A is an example of a *finite set*, containing a finite (3) number of elements. According to current astronomical knowledge, B is presumably finite, even though it contains an incredibly large number of elements. C is an example of an *infinite set*, which contains infinitely many elements. D is called the *empty set*, containing no elements at all.

Let us fix some mathematical terminology: If a set contains the objects a,b,c,and d, we can write S = {a,b,c,d} = {c,a,d,b}, i.e., the order of the elements in S is immaterial. Frequently sets are defined by saying that a set S contains all elements with a given property P; in symbols: *S = {x / x has property P}* (read S equals the set of elements x where x has the property P). For example, set C above could be defined by C = {x / x is an integer > 1000}. Similarly, B = {z / z is a star in the universe},

A = {y / y is a 7600 computer at LLL}.

If an element or object a, belongs to a set S, we write $a \in S$. Otherwise, $a \notin S$. Thus, $5000 \notin B$, but $5000 \in C$. Since D is the empty set (which is denoted by $\emptyset$), there exists no object x with $x \in D$, or stated differently, for all objects, x, $x \notin D$. In fact, the empty set can be defined as: $\emptyset = \{x / x \neq x\}$.

Just as numbers can be compared by their size ($5 < 7$, etc.), sets may (sometimes) be compared: given two sets A and B, we call *A a subset of B* if $x \in A$ implies $x \in B$. If A is a subset of B, we write $A \subset B$. Clearly, if $A \subset B$ and also $B \subset A$, then $A = B$ (just as for numbers: $x \leq y$ and $y \leq x$ implies $x = y$). The empty set is a subset of any set: $\emptyset \subset A$, for *all* sets A.

Examples:

(5) Let $E = \{x / x$ is integer and $x > 3000\}$, then $x \in E$ implies $x > 3000 > 1000$, so $x \in C$. Thus E is a subset of C: $E \subset C$.

(6) $F = \{y / y$ is a star less than 10^6 light years away from us$\}$. Clearly, $y \in F$ implies $y \in B$, so $F \subset B$.

(7) $G = \{t / t$ is a computer at LLL$\}$. Clearly, $t \in A$ implies $t \in G$, so $A \subset G$.

Set Operations

Now that we have defined a new class of objects called "sets," we proceed just as in elementary algebra where, having defined numbers, one studies algebraic operations (addition, multiplication, etc.) that can be performed with numbers. We'll do the same with sets, and it'll turn out that our set operations will even have many of the familiar properties that algebraic operations have.

Union of Sets

Given two sets A and B, their *union* is the set $A \cup B = \{x / x \in A$ or $x \in B\}$.

Intersection of Sets

The *intersection* of two sets A and B is the set $A \cap B = \{y / y \in A$ and $y \in B\}$.

Difference of Sets

The *difference* of two sets A and B is the set $A \setminus B = \{z \,/\, z \in A$ and $z \notin B\}$.

Example:

(8) Let $A = \{1,2,3,4\}$ and $B = \{2,4,6,8\}$:

$A \cup B = \{1,2,3,4,6,8\}$,

$A \cap B = \{2,4\}$,

$A \setminus B = \{1,3\}$,

$B \setminus A = \{6,8\}$.

Properties of Set Operations

The following eleven properties follow directly from the definitions:

(P_1) $(A \cup B) \cup C = A \cup (B \cup C)$
(P_2) $(A \cap B) \cap C = A \cap (B \cap C)$ } Associativity

(P_3) $A \cup B = B \cup A$
(P_4) $A \cap B = B \cap A$ } Commutativity

(P_5) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
(P_6) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ } Distributivity

(P_7) $A \cup A = A \cap A = A$

(P_8) $A \cup \emptyset = A$

(P_9) $A \cap \emptyset = \emptyset$

(P_{10}) $A \setminus A = \emptyset$

(P_{11}) $A \setminus \emptyset = A.$

The next two properties have important applications in Boolean algebra and are known as the *rules of De Morgan:*

(P_{12}) $A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$

(P_{13}) $A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$

More Set Constructions

The Power Set

Given a set S there is an important construction which associates a new set with S. This new set is called the *power set* of S: $P(S) = \{A / A \subset S\}$. Thus, the *power-set P(S) is the set of all subsets of S* (including $\emptyset$ and S itself as subsets, of course). For example, if $S = \{x / x \text{ integer}, 0 < x < 4\}$, then $S = \{1,2,3\}$ and $P(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, S\}$, so P(S) is a set with $8 = 2^3$ elements. In general, if S is a finite set of n elements, then P(S) is a finite set of 2^n elements. Note that $\emptyset$ has zero elements, but $P(\emptyset) = \{\emptyset\}$ contains 1 element.

The Cartesian Product

Given two sets A and B, we define a new set A x B, called the *Cartesian product* of A and B, as the set of all ordered pairs (a,b) with $a \in A$, $b \in B$: $A \times B = \{(a,b) / a \in A, b \in B\}$. Here, an *ordered pair* (x,y) is *not* the set $\{x,y\}$ but rather the set $\{\{x\}, \{x,y\}\}$. Thus $(x,y) = (\bar{x},\bar{y})$ if and only if $x = \bar{x}$, $y = \bar{y}$. In particular $(x,y) \neq (y,x)$ unless $x = y$.

Functions

Mapping from One Set to Another

Of fundamental importance not only to set theory but to all of higher mathematics is the concept of a function or map from one set to another.

Given two sets A and B, a *function f from A to B* (in symbols $f : A \to B$) is simply a rule which associates to any element $a \in A$ a *unique* element $b \in B$. This element b is denoted $b = f(a)$. All functions (e.g., $f(x) = x^2$, $f(x) = \sqrt{|\cos x|}$, etc.) that occur in calculus are examples of mappings from the set $\mathbf{R}$ of real numbers into the (same) set $\mathbf{R}$, i.e., $f : \mathbf{R} \to \mathbf{R}$.

Examples:

(9) Let $A = \{x / x \text{ is an employee at LLL}\}$
and $B = \{y / y \text{ integer}, y > 0\}$.
Define $f : A \to B$ by associating to every employee $x \in A$ the number of years he has worked at LLL ($= f(x)$).

(10) Let S be any set, and define a function $f: P(S) \rightarrow P(S)$ by $f(x) = S \setminus x$ for every $x \in P(S)$. This function simply associates with every subset $x \subset S$ its complement $S \setminus x$. In particular: $f(\emptyset) = S$, $f(S) = \emptyset$.

(11) Given two sets A and B, define two functions $P_1: A \times B \rightarrow A$ and $P_2: A \times B \rightarrow B$ by $P_1(a,b) = a$ and $P_2(a,b) = b$, for all $a \in A$, $b \in B$. These two functions are called *projections*.

Given a function f from some subset of a set A into the set B, we define the *domain of f* (D_f) by $D_f = \{a \in A \: / \: f(a) \text{ is defined}\}$. The *range of f* ($R_f$) is the set $R_f = \{b \in B \: / \: b = f(a) \text{ for some } a \in A\}$. Thus, the range of f is exactly the "image" of D_f under the map f.

A function $f : A \rightarrow B$ is called *surjective* (or *onto* B) if and only if $R_f = B$, i.e., if every element $b \in B$ is of the form $b = f(a)$.

A function $f : A \rightarrow B$ is called *injective* (or *one-to-one*) if $f(a) = f(a')$ implies $a = a'$, i.e., if different elements of A are mapped into different elements of B.

A function $f : A \rightarrow B$ that is both injective and surjective is called an *isomorphism* between A and B (or an *equivalence* between A and B).

Thus, the sets $A = \{5,7,500,-2\}$ and $B = \{X,Y,Z,\Omega\}$ are equivalent, since an isomorphism is given by $f : A \rightarrow B$ with $f(5) = X$, $f(7) = Y$, $f(500) = Z$, $f(-2) = \Omega$.

Cardinality of a Set

Obviously, two finite sets are equivalent if and only if they contain an identical number of elements. If we denote the number of elements of a finite set A by $|A|$, then A and B are equivalent if and only if $|A| = |B|$. The number $|A|$ is called the *cardinality* of A. The importance of the concept of an isomorphism between sets lies in the fact that it has led to a generalization of the cardinality to *infinite* sets! The idea is very simple as the following consideration shows. A child who cannot count yet can still find out whether two finite sets have the same cardinality (e.g., whether the number of apples on the table is equal to the number of oranges) by simply associating to each object of the first set a unique object of the second set (i.e., by pairing the oranges and apples). If the objects can be paired in this manner without any "left-overs," then the two sets must have had the same cardinality. By dropping the finiteness restriction we arrive at the following definition.

Definition 1:

Two (finite or infinite) sets have the same *cardinality*, if there is an isomorphism between them, i.e., if they can be paired in a one-to-one fashion.

Examples:

(12) Let $\mathbf{Z}^+ = \{n \,/\, n \text{ integer}, n > 0\}$ be the set of all positive integers. Let $SQ = \{x \,/\, x \text{ is integer of form } y^2\}$ be the set of all square numbers. Thus, $\mathbf{Z}^+ = \{1,2,3,4,\ldots\}$
$SQ = \{1,4,9,16,\ldots\} \subset \mathbf{Z}^+$.
The function $f\colon \mathbf{Z}^+ \to SQ$ given by $f(n) = n^2$ is clearly an equivalence; thus, $\mathbf{Z}^+$ and SQ have the same (infinite) cardinality. Similarly, $\mathbf{Z}^+$ and $CU = \{1,8,27,64,\ldots\}$ have the same cardinality ($f(n) = n^3$).

(13) The set $\mathbf{Z}^+$ of all positive integers is of the same cardinality as the set of all integers $\mathbf{Z} = \{\ldots,-2,-1,0,1,2,3,\ldots\}$. The proof is left as an exercise.

Note that in these examples a set (e.g., $\mathbf{Z}^+$) has the same cardinality as a proper subset (e.g., CU). This is a property that distinguishes infinite from finite sets.

Definition 2:

A set S is called *countable* if it has the same cardinality as $\mathbf{Z}^{|}$.

Examples of countable sets are the sets $\mathbf{Z}$ of *all* integers; SQ of all square numbers, CU of all cubes, $\mathbf{Z}^{ev}$ of all even integers, and $\mathbf{Z}^{odd}$ of all odd integers. The set Q of all *rational* numbers (fractions) is also countable.

The question now arises, "Are there infinite sets that have a higher cardinality than the countable ones?" The answer is "yes," as indicated in the following theorem.

Theorem 1:

Given an infinite set S, the power set P(S) has higher cardinality than S.

Proof:

We'll show that the assumption that S and P(S) have the same cardinality leads to a contradiction. Assume that there is an equivalence, $f : S \to P(S)$, which is "onto" and one-to-one. Define a subset $A \subset S$ as follows:

$$A = \{x \,/\, x \in S,\ x \notin f(x)\}.$$

Thus, A simply contains all those objects of S that do not belong to their associated subset f(x). Therefore, A as a subset of S is an element of P(S): $A \in P(S)$; and since f is "onto," there is an element $z \in S$ with $f(z) = A$.

We now ask whether $z \in A$ or $z \notin A$. If $z \in A$, then by definition of A, $z \notin f(z)$. But $f(z) = A$, so $z \in A$ implies $z \notin A$ - nonsense! However, if $z \notin A$, then $z \notin f(z) = A$, so $z \notin A$ implies $z \in A$ - again nonsense! The assumption has led to an impossible situation, and must be abandoned. Q.E.D.

The theorem we have just proved displays one of the most remarkable discoveries of modern mathematics, namely that there exist different (in fact, infinitely many) *infinite* cardinalities.

The simple and intuitive concept of one-to-one correspondence allows us to distinguish between different "sizes of infinity." Starting with an infinite set, say $\mathbf{Z}^+$, the set of all positive integers, we form the power-set $P(\mathbf{Z}^+)$, the set of all subsets of $\mathbf{Z}^+$. From what we have just shown, $P(\mathbf{Z}^+)$, has a higher cardinality than $\mathbf{Z}^+$. Next we can form the set $P(P(\mathbf{Z}^+))$, which is of higher cardinality than $P(\mathbf{Z}^+)$, $P(P(P(\mathbf{Z}^+)))$ is of higher cardinality than $P(P(\mathbf{Z}^+))$, etc. We can continue this process forever, and as a result obtain infinitely many sets of *different infinite cardinalities*.

Let us denote the (finite or infinite) cardinality of a set S by $|S|$. Then, one writes for the cardinality of P(S):

$$|P(S)| = 2^{|S|}.$$

For finite sets S we already know that if S has n elements, i.e., $|S| = n$, then $|P(S)| = 2^n$. Thus, we simply extend this fact to infinite cardinalities. One denotes the cardinality of the countable sets by $\aleph_0$ ($\aleph$ is *Aleph*, the first letter in the Hebraic alphabet). Thus, $|\mathbf{Z}| = |\mathbf{Z}^+| = \aleph_0$.

The set of all real numbers (including all integers, fractions, irrational numbers like $\sqrt{2}$, and transcendental numbers like π or e) is denoted by **R**, and its cardinality is denoted by $c = |\mathbf{R}|$. One can show that **R** is in one-to-one correspondence with $P(\mathbf{Z}^+)$, i.e., $|\mathbf{R}| = 2^{|\mathbf{Z}^+|}$, or in our new notation: $c = 2^{\aleph_0}$. This means that the set of all real numbers is of *higher cardinality* than the set of integers or rational numbers. Any set S which is in one-to-one correspondence with **R** (so that $|S| = c$) is called a *continuum*. In particular, we know from analytic geometry that the set of all real numbers is in one-to-one correspondence with all points on an infinitely long line. Thus, the set of all points on a line forms a continuum.

A famous problem posed at the beginning of this century by Cantor, the founder of set theory, was solved only a few years ago. It is the *continuum problem*.

Continuum Problem:

Are there sets S such that $\aleph_0 < |S| < 2^{\aleph_0}$? In other words, are there sets having higher cardinality than the integers but lower cardinality than the real numbers?

The interesting and unsuspecting answer to this question is that neither the existence nor the nonexistence of such sets can be proved using the axioms of set-theory. Either assertion is independent of the set-theoretic axioms, and could therefore be appended as one further axiom.

Equivalence Relations

Our final discussion will center around the concept of equivalence relations.

Definition 3:

Let A be a set (nonempty). A relation "$\sim$" which holds between certain pairs (a,b) of elements of A is called an *equivalence relation* if it satisfies the following three properties:

(a) $a \sim a \quad \forall\, a \in A$ (reflexive),*

(b) $a \sim b$ implies $b \sim a \quad \forall\, a,b \in A$ (symmetric),

(c) $a \sim b$ and $b \sim c$ implies $a \sim c \quad \forall\, a, b, c \in A$ (transitive).

*The symbol $\forall$ is read "for all" (see Appendix 3).

Examples:

(14) Let A be the set of all human beings, and let "$\sim$" be the relation "is blood-related to." Thus, if a and b are human beings, then $a \sim b$ means that a is blood-related to b. Clearly, $a \sim a$ (each person is blood-related to himself); $a \sim b$ implies $b \sim a$; and $a \sim b$ and $b \sim c$ implies $a \sim c$. Thus, the above relation is an equivalence relation.

(15) Let **Z** be the set of all integers, and let "$\sim$" be the relation "has the same remainder after division by 3 as." Thus, e.g., $5 \sim 11$, since both numbers have the same remainder (2) after division by 3. It is easy to verify that "$\sim$" is an equivalence relation. In this example, instead of writing $a \sim b$, one writes $a \equiv b \pmod 3$, and says that *a is congruent to b modulo 3*, which means that a and b leave the same remainder when divided by 3, or, $a - b$ is a multiple of 3.

(16) Let A be the set of all human beings as in (14), and let "$\sim$" be the relation "is a friend of." Thus, $a \sim b$ means "a is a friend of b." Properties (a) and (b) of an equivalence relation are obviously satisfied; however, property (c) is certainly not true, since a might be a friend of b, and b might also be a friend of c, but a and c might not be friends at all (if they always were, social life would be quite a bit simpler, but also more boring). Thus, "$\sim$" is *not* an equivalence relation.

Given an equivalence relation on a set A, we shall collect all elements that are equivalent to each other into a subset. If $a \in A$, then we'll denote the subset of all elements $b \in A$ with $a \sim b$ by $[a]$, so $[a] = \{b \,/\, b \in A, a \sim b\}$. Since "$\sim$" is an equivalence relation, $[a] = [b]$ if and only if $a \sim b$; otherwise $[a] \cap [b] = \emptyset$ (if $a \not\sim b$). So, the set A is subdivided into mutually disjoint subsets $[a]$; each such subset is called an *equivalence class*.

Definition 4:

Given a set A and an equivalence relation "$\sim$", we define the *quotient-set* $A/\sim$ as the set of all equivalence classes: $A/\sim \; = \{[a] \,/\, a \in A\}$.

In example (15) above, where "$\sim$" is the relation "has the same remainder after division by 3 as," there exist three mutually disjoint equivalence classes:

$[\emptyset]$ = set of all integers whose remainder is $\emptyset$,

$[1]$ = set of all integers whose remainder is 1,

$[2]$ = set of all integers whose remainder is 2.

Thus:

$[\emptyset] = \{\emptyset,\pm3,\pm6,\pm9,\pm12,\dots\}$,

$[1] = \{\dots-8,-5,-2,1,4,7,10,\dots\}$,

$[2] = \{\dots,-7,-4,-1,2,5,8,11,\dots\}$.

Hence $\mathbf{Z}/\sim \; = \{[\emptyset], [1],[2]\}$ has three elements. Since the equivalence relation depended on the number 3, we write $\mathbf{Z}/\sim \; = \mathbf{Z}_3$.

(17) Our final example of an equivalence relation has many important applications. We consider two sets A and B, and a function $f\colon A \to B$. Let "$\sim$" be the following relation in A: $a \sim b$ means "$f(a) = f(b)$." Clearly, $a \sim a$ since $f(a) = f(a)$; $a \sim b$ implies $b \sim a$; and $a \sim b$, $b \sim c$ implies $a \sim c$, since $f(a) = f(b)$, $f(b) = f(c)$ implies $f(a) = f(c)$. Thus, "$\sim$" is an equivalence relation in A. We can therefore form the quotient set $A/\sim$ of all equivalence classes in A.

Proposition 1:

There is a one-to-one correspondence between $A/\sim$ and the range of f, R_f, i.e., $|A/\sim| = |R_f|$.

Proof:

We define a function $\rho\colon \; A/\sim \; \to R_f \subset B$ by: $\rho([a]) = f(a)$ for all $[a] \in A/\sim$. First we must show that ρ is "well-defined," i.e., that ρ is independent of the element $a' \in [a]$. Thus, let $a' \in [a]$. Then $a' \sim a$, i.e., $f(a') = f(a)$ by definition of "$\sim$." Then $\rho([a']) = f(a') = f(a) = \rho([a])$, so ρ is independent of the choice of $a' \in [a]$. Clearly, ρ is surjective, since every $b \in R_f$ has the form $b = f(a)$ for some $a \in A$, hence, $b = \rho([a])$. Next suppose, $\rho([a]) = \rho([b])$. Then $f(a) = f(b)$ by the definition of ρ. But $f(a) = f(b)$ means "$a \sim b$," so $[a] = [b]$. This shows that ρ is injective. Since ρ is injective and surjective, it is one-to-one.

Given A and "$\sim$," there is an obvious function $\psi : A \to A/\sim$, namely, $\psi(a) = [a]$. Clearly, $\psi(a) = \psi(b)$ if and only $a \sim b$. Given $f : A \to B$, we showed above that there is an equivalence relation "$\sim$" in A and a function $\rho : A/\sim \to R_f \subset B$. Composing this function with $\psi : A \to A/\sim$, we obtain:

$$A \overset{\psi}{\to} A/\sim \overset{\rho}{\to} B.$$

It is easily seen that $f = \rho \cdot \psi$, so that the triangle of functions shown in Fig. 1 "commutes." Thus, every function $f : A \to B$ can be split up into a composition of a *surjective function* (ψ), *followed by an injective function* (ρ). In modern algebra to yield important results, this fact is exploited quite frequently.

$$\begin{tikzcd}
A \arrow[rr, "f"] \arrow[dr, "\psi"'] & & B \\
& A/\sim \arrow[ur, "\rho"'] &
\end{tikzcd}$$

Fig. 1. Function diagram.

<u>Exercises</u>

(1) If X is a set, let P(X) be the set of all subsets of X, i.e., the power set of X. Which of the following 3 equations are incorrect, and which would be the correct statements?

(a) $P(A \cup B) = P(A) \cup P(B)$

(b) $P(A \cap B) = P(A) \cap P(B)$

(c) $P(A \setminus B) = P(A) \setminus P(B)$

(2) Show that

(a) $A \setminus B = A \setminus (A \cap B)$

(b) $A \setminus B = A \leftrightarrow A \cap B = \emptyset$.

(3) Let $\mathbf{Z}^+ = \{1,2,3,\ldots\}$ and $\mathbf{Z} = \{\ldots,-2,-1,0,1,2,3,\ldots\}$. Find a function $f : \mathbf{Z}^+ \to \mathbf{Z}$ which is one-to-one and onto.

(4) Suppose, A and B are nonempty sets, and $f : A \to B$ and $g : B \to A$ are functions. Let $1_A : A \to A$ and $1_B : B \to B$ be the "identity-functions," i.e.,

$1_A(a) = a$, for all $a \in A$
$1_B(b) = b$, for all $b \in B$

What can be said about f and g, if

(a) $g \cdot f = 1_A$, i.e., $g(f(a)) = a$, for all $a \in A$?
(b) $g \cdot f = 1_A$ *and* $f \cdot g = 1_B$?

(5) Give a short (3 lines) proof that $P(A) = P(B)$ implies $A = B$.

THEORY OF LANGUAGES

In every language there is a distinction between the *structure* of that language and the *meaning* of its words and sentences. Consequently, linguists distinguish between the *syntax*, which deals with the structure only, and the *semantics*, which deals with the meaning of the words. In the following we shall be concerned only with the syntax, which lends itself relatively easily to mathematical investigation.

Formal Definition of Languages

Consider the following example of a sentence in the English language:

$$\text{"}\underbrace{\text{The man}}_{\text{noun phrase}}\ \underbrace{\text{gave me a book}}_{\text{verb phrase}}\text{."}$$

As indicated, this sentence consists of two parts, the noun phrase (NP) and the verb phrase (VP), in that order. If we use the symbol S to denote the entire sentence, we could symbolically write

$$S \rightarrow (NP)\ (VP),$$

meaning, "We can start with a noun phrase, and follow it with a verb phrase to obtain a sentence."

In our example, the noun phrase can be further partitioned into two parts: an article (A), which is the word "the," and a noun (N), the word "man." Using the above symbolism we can write

$$(NP) \rightarrow AN,$$

meaning, "We can start with an article, and follow it with a noun to obtain a noun phrase." Similarly we can partition the verb phrase into a verb (gave), followed by a noun phrase (me) and another noun phrase (a book), i.e.,

$$(VP) \rightarrow V(NP)(NP).$$

Now, the noun phrase "me" consists of a single pronoun (PR), so we also have

$$(NP) \rightarrow (PR).$$

Finally, for our particular example we have

$$
\begin{aligned}
PR &\to \text{me}\\
A &\to \text{the}\\
A &\to \text{a}\\
N &\to \text{man}\\
N &\to \text{book}\\
V &\to \text{gave}
\end{aligned}
$$

The symbolic rules of the form $x \to y$, which specify how sentences are formed, are called *productions* or *rewriting rules*. To arrive at a sentence S, we must start with a rewriting rule of the form $S \to y$.

Let us use our above rules to derive the original sentence:

$$
\begin{aligned}
S &\to (NP)(VP) \to AN(VP) \to A(\text{man})(VP)\\
&\to A(\text{man})V(NP)(NP) \to A(\text{man})V(PR)(NP)\\
&\to A(\text{man})V(\text{me})(NP) \to A(\text{man})(\text{gave})(\text{me})(NP)\\
&\to A(\text{man})(\text{gave})(\text{me})AN \to (\text{The})(\text{man})(\text{gave})(\text{me})AN\\
&\to (\text{The})(\text{man})(\text{gave})(\text{me})A(\text{book})\\
&\qquad \to (\text{The})(\text{man})(\text{gave})(\text{me})(\text{a})(\text{book})
\end{aligned}
$$

Using the same rewriting rules in another order we could have obtained sentences like, "A man gave me the book," or "The man gave me the book," or "The book gave me a man," etc.

Looking back at this example, we could think of the (English) language as being comprised of words (the, or, gave, man, etc.), and of a "grammar" reflected by the rewriting rules. This consideration leads straight to the mathematical formulation of languages as follows.

First we must specify a *finite* set of symbols Σ. Let us denote the elements (symbols) of Σ by $x_1, x_2, \ldots, y_1, y_2, \ldots$. Then a *word* or a *string over* Σ *of length n* is a finite sequence $x_1x_2x_3\ldots x_n$, where $x_i \in \Sigma$, $n \geq 0$. If $n = 0$, we denote the (empty) string of length 0 by ε. If $|\Sigma|$ is the number of elements in Σ, then the number of words of length 1 is $|\Sigma|$, while in general, the number of words of length n is $|\Sigma|^n$ $(n = 0,1,2,\ldots)$. If w is a word of length n, write $|w| = n$. Now, two words w, v are *equal* if $|w| = |v|$ and both contain the same symbols in the same order.

If $w = x_1x_2 \ldots x_m$ and $v = y_1y_2 \ldots y_n$ are two words of length m and n, respectively, then wv is a new word of length m+n obtained by putting the symbols of v behind those of w:

$$wv = x_1x_2 \ldots x_m y_1 y_2 \ldots y_n.$$

Obviously, $|wv| = |w| + |v|$ for all strings v, w. Also, for three words, u, v, w:

$$(uv)w = u(vw) = uvw$$

and $\varepsilon w = w\varepsilon = w$, if ε is the empty word.

The above concatenation of words is called the *product* of words.

Now let Σ^* be the set of all finite words over Σ. In contrast to Σ the set Σ^* is an *infinite* set.

Definition 5:

The set Σ^* together with the above multiplication is called the *free semigroup generated by* Σ.

Definition 6:

A *language over* Σ is a subset of Σ^*.

This definition of languages is much too general. To obtain a more interesting definition we have to recognize the grammar as a main part of the language. We can use our intuition to describe a grammar (as a set of rewriting rules).

Definition 7:

A *phrase-structure grammar* (or simply, *grammar*) is a quadruple

$$G = (V, \Sigma, P, \sigma),$$

where:

V is a finite set (the *vocabulary*).

Σ is a subset of V, $\Sigma \subset V$ (the set of *terminals*).

P is a finite set of ordered pairs (u,v) such that u is a nonempty word of $(V\backslash\Sigma)^*$ and $v \in V^*$ (empty or not). Instead of $(u,v) \in P$, we write $u \to v$, so that P is the set of *productions* or *rewriting rules*.

σ is the *start variable* (corresponding to our sentence symbol S). $\sigma \in V \setminus \Sigma$, so $V \setminus \Sigma \neq \emptyset$.

We can interpret the various quantities in this definition as follows (using our example): Σ, the set of terminals, contains English words like "the," "man," "book," "me," etc. V, the vocabulary, contains Σ and $V \setminus \Sigma$, the set of variables or nonterminals consisting of (NP), (VP), A, N, etc. P, the set of rewriting rules, contains rules like (NP)(VP) $\to$ AN(NP)(NP), where the left side must be a nonempty string of variables and/or terminals, while the right side may be any string of terminals and variables, including the empty string ε.

Now suppose that w and w' are in V^*. If there are strings $w_1, w_2 \in V^*$, and if $u \to v$ is in P such that $w = w_1uw_2$ and $w' = w_1vw_2$, we write $w \underset{G}{\Rightarrow} w'$. Thus, $w \underset{G}{\Rightarrow} w'$ means that there is a substring u in w that produces a substring v of w', while the remaining substrings of w and w' are identical.

Example:

(18) AN(VP) $\underset{G}{\Rightarrow}$ ANV (NP)(NP), since (VP) $\to$ V(NP)(NP) is a production, while the string AN is identical on both sides. More generally, we say that *w' may be derived from w* in the grammar G, if there is a *finite* sequence of strings

$$w_0 \underset{G}{\Rightarrow} w_1 \underset{G}{\Rightarrow} w_2 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} w_{n-1} \underset{G}{\Rightarrow} w_n,$$

such that $w = w_0$, $w' = w_n$, and all $w_i \in V^*$. In this case we write

$$w \overset{*}{\underset{G}{\Rightarrow}} w'.$$

Now we are able to tie the grammar into the definition of a language.

Definition 8:

Let $G = (V,\Sigma,P,\sigma)$ be a grammar. Then the set

$$L(G) = \{w \in \Sigma^* \,/\, \sigma \overset{*}{\underset{G}{\Rightarrow}} w\}$$

is the *language generated by the grammar G.*

Grammars as defined above are also called *Chomsky grammars* after the linguist Noam Chomsky. We present a couple of examples of languages generated by grammars.

Examples:

(19) For any set Σ, the set $\{\varepsilon\}$ (consisting of the empty word alone) is a language generated by the grammar

$$G = (V,\Sigma,P,\sigma),$$

where $V = \Sigma \cup \{\sigma\}$ and $P = \{(\sigma\to\varepsilon)\}$i.e., the only rewriting rule is $\sigma \to \varepsilon$.

(20) The whole set Σ^* is a language generated by the grammar

$$G = (V,\Sigma,P,\sigma),$$

where $V = \Sigma \cup \{\sigma\}$, and P consists of the productions $\sigma \to a\sigma$ for all $a \in \Sigma$ (finite!) and $\sigma \to \varepsilon$. For instance, if $\Sigma = \{a,b\}$, then we derive the word $aba^3 = abaaa$ as follows:

$$\sigma \to a\sigma \Rightarrow ab\sigma \Rightarrow aba\sigma \Rightarrow abaa\sigma \Rightarrow aba^3\sigma \Rightarrow aba^3,$$

hence $\sigma \overset{*}{\underset{G}{\Rightarrow}} aba^3$, and so $aba^3 \in L(G)$.
Any other word of Σ^* may be similarly derived, and hence $L(G) \supset \Sigma^*$ which, of course, implies that $L(G) = \Sigma^*$.

(21) If $w = x_1x_2\ldots x_n \in \Sigma^*$, let $w^R = x_nx_{n-1}\ldots x_2x_1$ be the word w written in reverse order. (e.g., if $w = a^2b^3c$, then $w^R = cb^3a^2$). Then the set $\{ww^R \,/\, w \in \Sigma^*\}$ is a language generated by a grammar

$$G = (V,\Sigma,P,\sigma),$$

where $V = \Sigma \cup \{\sigma\}$, and P contains the productions $\sigma \to a\sigma a$ for all $a \in \Sigma$ (finite!) and $\sigma \to \varepsilon$. For instance, if $w = aba$, then $ww^R = abaaba$ is derived as follows:

$$\sigma \to a\sigma a \Rightarrow ab\sigma ba \Rightarrow aba\sigma aba \Rightarrow abaaba,$$

i.e., $\sigma \overset{*}{\underset{G}{\Rightarrow}} aba^2ba$, so $aba^2ba \in L(G)$.

(22) The set $\{ww \mid w \in \Sigma^*\}$ is a language generated by a grammar G with the following set of nonterminals:

$$V \setminus \Sigma = \{\sigma, A_x, B_x, C_x, D_x, E_{xy}, F_{xy} \mid \text{for all } x,y \in \Sigma\}.$$

So, for each symbol $x \in \Sigma$ we define four nonterminals A_x, B_x, C_x, D_x, and for each ordered pair $(x,y) \in \Sigma \times \Sigma$ we define two nonterminals E_{xy} and F_{xy}. The productions P are:

$$\begin{aligned}
\sigma &\to \varepsilon \\
\sigma &\to A_x B_x, \text{ for each } x \in \Sigma \\
A_x &\to A_x C_y, \text{ for all } x,y \in \Sigma \\
A_x C_y C_z &\to A_x A_y E_{zy}, \text{ for all } x,y,z \in \Sigma \\
E_{xy} C_z &\to C_x E_{zy} \\
E_{xy} D_z &\to C_x F_{zy} \\
E_{xy} B_z &\to C_x D_z B_y \\
F_{xy} D_z &\to D_x F_{zy} \\
F_{xy} B_z &\to D_x D_z B_y \\
A_x C_y D_z &\to A_x A_y F_{zy} \\
A_x C_y B_z &\to A_x A_y D_z B_y \\
A_x &\to x \\
B_x &\to x \\
D_x &\to x
\end{aligned}$$

Classification of Languages (Chomsky)

In 1959 N. Chomsky classified grammars (and hence their associated languages) into four types, according to the form of the productions. He labeled the widest class of languages type 0, and then imposed increasingly severe restrictions on the productions to obtain type 1, type 2, and finally type 3 languages, which constitute the smallest class.

Type 0 (Recursive Languages)

These are all languages that we have considered so far - there are *no* restrictions on the production rules.

Examples of type 0 productions:

$$CBD \rightarrow CabbDabcB$$

$$AB \rightarrow AbbA$$

$$AC \rightarrow C$$

$$B \rightarrow \varepsilon$$

Type 1 (Context-Sensitive Languages)

In this subset of the recursive languages, productions must be of the form

$$\alpha A \beta \rightarrow \alpha w \beta,$$

where $\alpha, \beta \in V^*$, $A \in V \setminus \Sigma$, $w \in V^* \setminus \{\varepsilon\}$.

So, a *single* nonterminal A must be replaced by a nonempty string w of terminals and/or nonterminals; the strings α and β are called the *context*, and either string may be empty.

Examples of type 1 productions:

$$XAY \rightarrow XaBbcY$$

$$AB \rightarrow abYB$$

$$aaBC \rightarrow aaBbaA$$

$$X \rightarrow abX$$

Rules like $A \rightarrow \varepsilon$ or $ABC \rightarrow AB$ are not allowed - in fact the right-hand side of a type 1 production can never be shorter in length than the left-hand side.

Type 2 (Context-Free Languages)

The type 2 languages form a subset of the type 1 languages. The productions are restricted to the form

$$A \rightarrow w,$$

where $A \in V \setminus \Sigma$ and $w \in V^* \setminus \{\varepsilon\}$.

So, in a type 2 production, a single nonterminal A is replaced by a nonempty string w of terminals and/or nonterminals. Note that these are all those context-sensitive productions with the context empty.

Examples of type 2 productions:

$$A \to abAa$$

$$B \to bX$$

$$C \to aA$$

$$D \to ABC$$

Type 3 (Regular Languages)

Type 3 productions are of the form

$$A \to w,$$

where $A \in V \setminus \Sigma$, and $w = a$ (a single terminal symbol), or $w = aB$ (a single terminal, followed by a single nonterminal).

Examples of type 3 productions:

$$A \to bX$$

$$B \to a$$

$$X \to aX$$

Note that for type 1 (and hence also for types 2 and 3), $u \to v$ implies $|u| \leq |v|$, i.e., these productions cannot shorten the length of a string. In other words, if a grammar G contains a production $u \to v$ with $|u| > |v|$, then G is necessarily type 0 and not type 1.

Note that a language $L \subset \Sigma^*$. can be generated by many different grammars, in fact, even by grammars of different types. The following shows how the language

$$L = \{w \mid w \in \{0,1\}^*, \text{ and each 0 is immediately followed by a 1}\}$$

can be generated by a grammar of each of the four types.

Type 0:

$G_0 = (V_0,\Sigma,P_0,\sigma)$, where $V_0\backslash\Sigma = \{\sigma\}$, and

$P_0 = \{\sigma \to 01\sigma;\ \sigma \to 1\sigma;\ \sigma \to \varepsilon\}$.

G_0 is *type 0* (because $\sigma \to \varepsilon$) and generates L; $L(G_0) = L$.

Type 1:

$G_1 = (V_1,\Sigma,P_1,\sigma)$, where $V_1\backslash\Sigma = \{\sigma,A\}$, and

$$P_1 = \left\{\begin{array}{c} \sigma \to 1\sigma \\ \sigma \to 0A \\ 0A \to 01\sigma \\ 1\sigma \to 101 \\ \sigma \to 1 \\ A \to 1 \end{array}\right\}$$

G_1 is *type* 1 (because $0A \to 01\sigma$, $1\sigma \to 101$), and generates L: $L(G_1) = L$.

Type 2:

$G_2 = (V_2,\Sigma,P_2,\sigma)$, where $V_2\backslash\Sigma = \{\sigma,A\}$, and

$$P_2 = \left\{\begin{array}{c} \sigma \to 01A \\ A \to \sigma \\ A \to 1A \\ \sigma \to 01 \\ \sigma \to 1 \end{array}\right\}$$

G_2 is *type 2* (because $\sigma \to 01A$, $\sigma \to 01$), and $L(G_2) = L$.

Type 3:

$G_3 = (V_3,\Sigma,P_3,\sigma)$, where $V_3\backslash\Sigma = \{\sigma,A\}$, and

$$P_3 = \left\{ \begin{array}{l} \sigma \to 0A \\ A \to 1\sigma \\ A \to 1 \\ \sigma \to 1 \end{array} \right\}$$

G_3 generates L: $L(G_3) = L$, and is *type 3*.

So, the language L actually belongs to the smallest class of *regular* languages.

If n is one of the integers 0, 1, 2, or 3, then let L(n) = {L / L is language of type n}. We then have the following proper inclusions:

$$L(3) \subset L(2) \subset L(1) \subset L(0).$$

See Fig. 2.

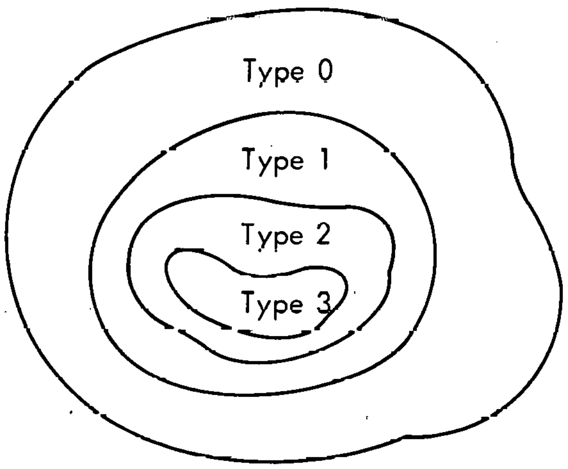


Fig. 2. Set-theoretic relationship of languages.

Note that the empty string ε cannot appear in type 1, 2, and 3 languages, since the productions do not shorten a string. It is customary, however, to include ε in all languages, and it sounds plausible that by doing so we do not change the main features of a language. We shall not give a rigorous argument for this, but shall from now on assume that $\varepsilon \in L(G)$. This can, e.g., be achieved by adding the production

$$\sigma \to \varepsilon,$$

assuming that σ *never* occurs on the *right-hand* side of any production.

The following "membership question" is of fundamental importance in our theory:

Given a grammar G and a string w, is w $\in$ L(G)?

If we could build a machine to answer the membership question, we would have a mechanical device for testing the validity of sentences in a language L(G). We are thus led to the question, "What types of machines can answer the membership question for each of the four types of languages?" Obviously, the complexity of the machines will increase as the class of languages becomes bigger, i.e., the machines that can answer the membership question for regular languages will be much simpler than those that can answer the membership question for the whole set of recursive languages. It therefore seems reasonable to concentrate on the regular languages first, and to try to find (simple) machines that handle this set of languages.

Definition 9:

If the membership question of a language L can be answered by a machine in finitely many steps, then L is called *decidable*.

As it turns out, context-sensitive, and hence context-free and regular languages are decidable. On the other hand, recursive languages are not decidable.

Algorithms, Computability, and Decidability

One of the basic concepts of computer science is that of an "algorithm." Since this word appears quite frequently in the text that follows, let us discuss its meaning and some closely-related concepts.

Quite informally we define procedure and algorithm.

Definition 10:

A *procedure* is a finite ordered sequence of instructions than can be mechanically carried out step by step.

Definition 11:

A procedure which always terminates in a finite amount of time is called an *algorithm.*

Clearly, the definition of procedure is not rigorous, since we have not defined what we mean by saying that its instructions are carried out "mechanically." Intuitively, we know what is meant, and probably identify "procedure" with "computer program." This identification will do for our future considerations. Let us present two examples:

(23) Figure 3 is the flowchart of a procedure that, given input number $N > 1$, will decide whether or not N is a prime number.

This procedure obviously terminates for every input number $N > 1$, since either a value of i will be reached that is greater than or equal to N, in which case N is a prime; or some value of i will divide N, and then N is not a prime. Thus, the procedure shown in Fig. 3 is actually an algorithm.

(24) A perfect number is an integer equal to the sum of all its divisors except itself. Thus, $N = 6$ is perfect, since $6 = 1 + 2 + 3$. $N = 28$ is perfect, since $28 = 1 + 2 + 4 + 7 + 14$. It is not known whether there are finitely many or infinitely many perfect numbers. Also, no odd perfect numbers are known. Figure 4 is a flowchart for a procedure that, given an input integer N, determines whether there is a perfect number $K > N$.

This procedure need not always terminate. If there are only finitely many perfect numbers, and N is greater than the largest one, then the procedure will continue testing larger and larger values of K, and will never halt. If, however, there is a perfect number $K > N$, then the procedure will find it in a finite number of steps. This procedure, therefore, is not an algorithm.

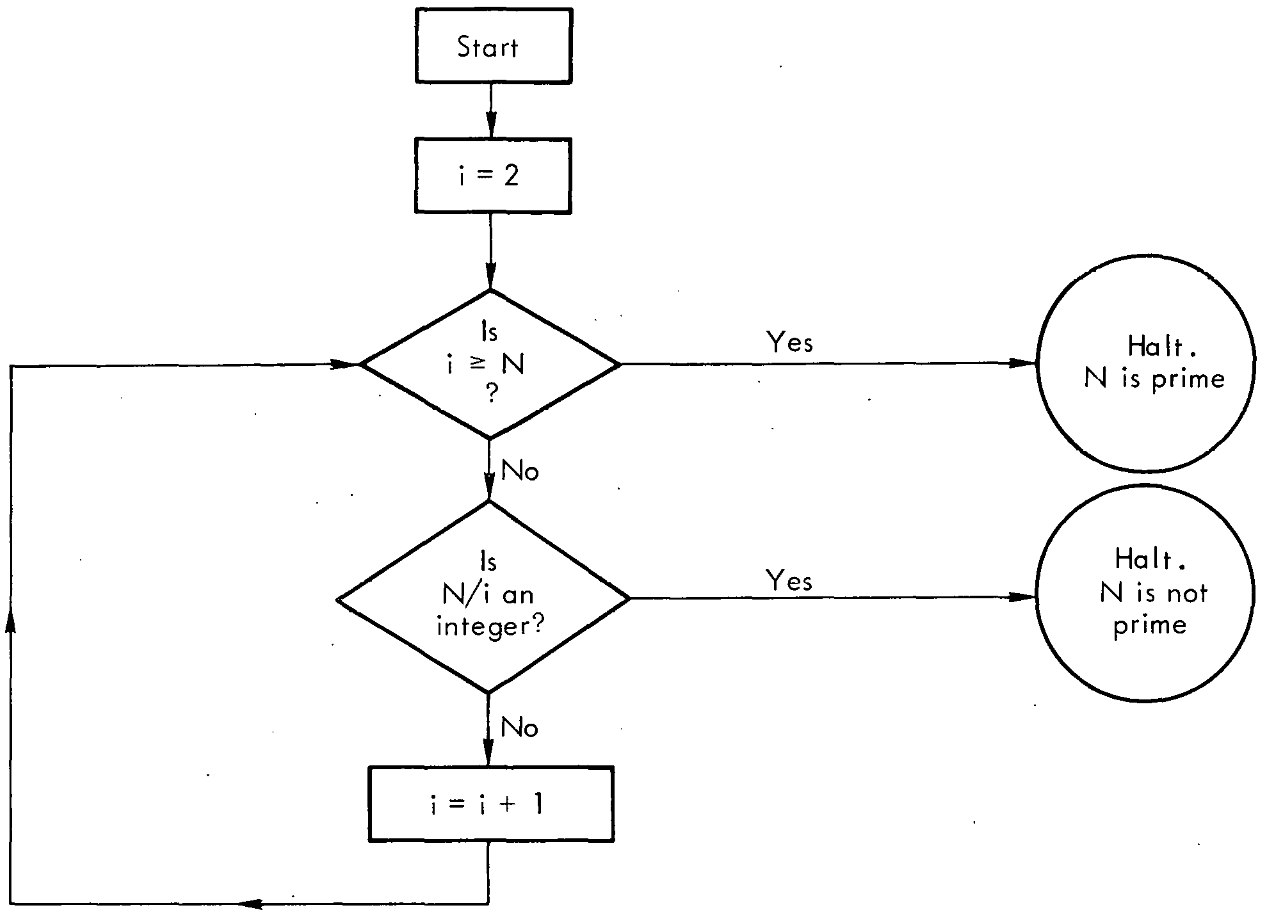


Fig. 3. Flowchart for determining whether N is a prime number.

Now let us think of an algorithm as a function F which is given an input value (N), and returns as output some integer F(N). For example, in our algorithm (1) we might output the integer 0, if N is not a prime, and 1, if N is a prime. Thus, F(5) = 1, while F(4) = 0. Thought of in these terms, an algorithm is nothing else but a function $F : \mathbf{Z}^+ \to \mathbf{Z}^+$, where $\mathbf{Z}^+$ is the set of nonnegative integers. We are now ready to define three important concepts that we will use repeatedly.

Definition 12:

A function $f : \mathbf{Z}^+ \to \mathbf{Z}^+$ is called *computable* if and only if there exists an algorithm $F : \mathbf{Z}^+ \to \mathbf{Z}^+$, such that f(n) = F(n) for all values $n \in \mathbf{Z}^+$.

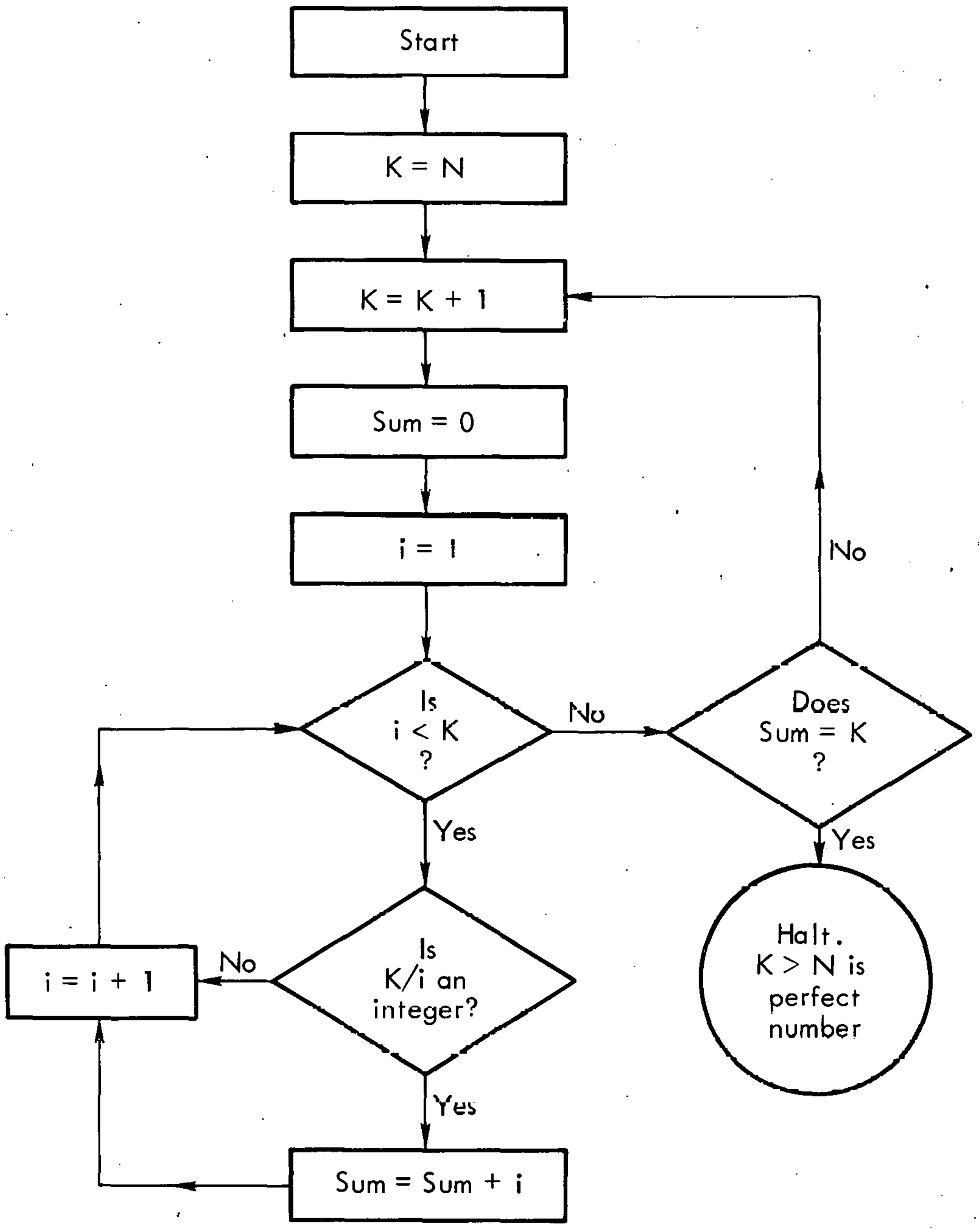


Fig. 4. Flowchart for determining whether there is a perfect number K greater than N.

Definition 13:

A subset $S \subset \mathbf{Z}^+$ is called *enumerable* if and only if there exists an algorithm $G : \mathbf{Z}^+ \to \mathbf{Z}^+$, such that $S = R_G = \{n \in \mathbf{Z}^+ / n = G(i) \text{ for some } i \in \mathbf{Z}^+\}$. Thus S is the "range" for some algorithm G.

Definition 14:

A subset $S \subset \mathbf{Z}^+$ is called *decidable* (or *solvable*) if and only if there exists an algorithm $H : \mathbf{Z}^+ \to \mathbf{Z}^+$ such that $S = \{n \in \mathbf{Z}^+ / H(n) = 1\}$. H is said to "decide S."

Clearly, a function $f: \mathbf{Z}^+ \to \mathbf{Z}^+$ is computable if it can be expressed in terms of an algorithm (i.e., if it can be programmed). We'll see shortly that noncomputable functions exist. A set $S \subset \mathbf{Z}^+$ is enumerable if there exists a computable function $f : \mathbf{Z}^+ \to S$ (onto S). Thus, a set S is enumerable if each element $n \in S$ is the result of applying an algorithm G to some integer $i \in \mathbf{Z}^+$. Nonenumerable sets exist as we'll show soon. Finally, a set $S \subset \mathbf{Z}^+$ is decidable if an algorithm exists which decides for every $n \in \mathbf{Z}^+$ whether $n \in S$ $(H(n) = 1)$ or $n \notin S$ $(H(n) = 0)$. *Finite* sets S are both enumerable and decidable.

Theorem 2:

(a) There exist functions $f : \mathbf{Z}^+ \to \mathbf{Z}^+$ that are not computable.

(b) There exist sets $S \subset \mathbf{Z}^+$ that are not enumerable.

Proof:

(a) Every procedure P (in particular every algorithm F) is a finite string over a finite alphabet. Hence, the set of all procedures is infinitely countable, and we can arrange it as a sequence $\{P_0, P_1, P_2, \ldots\}$. Some of these procedures are actually algorithms, say

$$\left\{P_{i_0}, P_{i_1}, P_{i_2}, \ldots\right\} \quad \text{is the subsequence of algorithms.}$$

Now define the function $f : \mathbf{Z}^+ \to \mathbf{Z}^+$ by

$$f(n) = P_{i_n}(n) + 1; \qquad n = 0,1,2,\ldots .$$

Suppose f were computable. Then there would be an algorithm F with $f(n) = F(n)$ for all $n \in \mathbf{Z}^+$. Since F is an algorithm, it must appear in the above sequence $P_{i_0}, \ldots$, say, $F = P_{i_k}$. Thus, $f(n) = P_{i_k}(n)$ for all n. In particular, if we choose $n = k$, we obtain:

$$f(k) = P_{i_k}(k);$$

but $f(k) = P_{i_k}(k) + 1$, which is impossible. Thus, f is not computable.

(b) Consider the sequences $P_0, P_1, \ldots$ of procedures and $P_{i_0}, P_{i_1}, \ldots$ of algorithms, and define $S = \{n \in \mathbf{Z}^+ / P_n \text{ is an algorithm}\}$. If S were enumerable, then an algorithm F would exist such that $n = F(i)$ for all $n \in S$. But $F = P_k$ for some $k \in S$. Now the function $f\colon \mathbf{Z}^+ \to \mathbf{Z}^+$, given by

$$f(j) = P_{i_j}(j) + 1$$

would be computable since $i_j = P_k(i)$ for some $i \in \mathbf{Z}^+$. By (a) we know that f is not computable, hence S is not enumerable.

Theorem 3:

S decidable implies S enumerable.

Proof:

Suppose, $S = \{n \in \mathbf{Z}^+ / F(n) = 1\}$, where F is an algorithm. Let us define an algorithm $G : \mathbf{Z}^+ \to \mathbf{Z}^+$ as given by the following FORTRAN program (with input number INPUT and answer RESULT):

```
          J = 0
          DO L3  I = 1, INPUT
      L1  J = J + 1
          IF(F(J).EQ.1)L2,L1
      L2  CONTINUE
          RESULT = J
      L3  CONTINUE
          END
```

Clearly, this algorithm G is simply given by $G(i) = i^{\text{th}}$ number of S, and hence $S = R_G$, i.e., S is enumerable.

Theorem 4:

If S and $\mathbf{Z}^+ \setminus S$ are both enumerable, then S (and hence $\mathbf{Z}^+ \setminus S$) is decidable.

Proof:

Suppose $S = R_F$ and $\mathbf{Z}^+ \setminus S = R_G$, where F, G are algorithms. Define an algorithm (FORTRAN program) H with input INPUT and answer RESULT:

```
    J = 0
L1  J = J + 1
    IF(F(J).EQ.INPUT)L3
    IF(G(J).EQ.INPUT)L2,L1
L2  CONTINUE
    RESULT = 0
    GO TO L4
L3  CONTINUE
    RESULT = 1
L4  END
```

Clearly, $S = \{n \in \mathbf{Z}^+ \, / \, H(n) = 1\}$, so S is decidable.

Theorem 5:

There exist sets $S \subset \mathbf{Z}^+$ which are enumerable but not decidable.

We shall not prove this result, but an example of an enumerable, yet undecidable set is $S = \{n \in \mathbf{Z}^+ \, / \, P_n(n) > 0\}$.

Backus-Naur Form of Grammars

A different way to write the productions of a formal grammar is known as the *Backus-Naur Form* (BNF). This form has been used extensively in describing the grammars of various programming languages, and was first applied in the formal syntactic description of ALGOL. The advantage of BNF is that it allows a more compact way of listing productions.

So far, our description of grammars made use of one "metasymbol" $\rightarrow$ which is used in productions, e.g., $\sigma \rightarrow A$. Also, two or more productions with the same left-hand side had to be listed separately, e.g.,

$$\sigma \rightarrow A$$

$$\sigma \rightarrow aB$$

$$\sigma \rightarrow C$$

In BNF three "metasymbols" are used:

$$\text{"}< >\text{"}, \text{"}::=\text{"}, \text{"}|\text{"}.$$

The symbol "< >" is used to enclose and thereby identify nonterminals, e.g., $<A>$, $<\sigma>$. The symbol "::=" replaces our old "$\rightarrow$". The symbol "|" is used to separate different right-hand sides of productions corresponding to the *same* left-hand side. For example,

$$<\sigma> ::= <A>|a<B>|<C>$$

replaces the three productions $\sigma \rightarrow A$

$$\sigma \rightarrow aB$$

$$\sigma \rightarrow C$$

The symbol "|" is read as "or".

Example:

(25) We shall now present an example of a simplified grammar for generating arithmetic expressions in ALGOL (the actual grammar used, for instance, in the ALGOL-60 report is slightly more complex).

Let $G = (V,\Sigma,P,\sigma)$ be the grammar

with $\Sigma = \{+,-,*,/,**,1,2,\ldots,9,a,b,c,\ldots,z,(,)\}$

$V \setminus \Sigma = \{\sigma,T,F,VA,L,D\}$, and productions

$$P = \left\{ \begin{array}{l} \sigma \rightarrow T \\ \sigma \rightarrow \sigma + T \\ \sigma \rightarrow \sigma - T \\ T \rightarrow F \\ T \rightarrow T * F \\ T \rightarrow T/F \\ F \rightarrow VA \\ F \rightarrow F ** VA \\ VA \rightarrow L\ VA \\ VA \rightarrow L \\ VA \rightarrow VA\ D \\ VA \rightarrow (\sigma) \end{array} \right\}$$

We'll call

σ an arithmetic expression,
T a term,
F a factor,
VA a variable,
L a letter,
D a digit.

When writing the same productions in BNF we'll use different names for our nonterminals to convey their meaning better:

<AE> = σ
<TERM> = T
<FACTOR> = F
<VAR> = VA
<LETTER> = L
<DIGIT> = D

The productions in BNF are:

```
       <AE> ::= <TERM>|<AE> + <TERM>|<AE> = <TERM>
     <TERM> ::= <FACTOR>|<TERM>*<FACTOR>|<TERM>/<FACTOR>
   <FACTOR> ::= <VAR>|<FACTOR>**<VAR>
      <VAR> ::= <LETTER><VAR>|<LETTER>
                <VAR><DIGIT>|(<AE>)
```

Note that in this grammar ** has precedence over * and /; * and / have precedence over + and -.

<u>Exercises</u>

(6) Let $\Sigma = \{(,)\}$ and $V = \Sigma \cup \{\sigma\}$, and let there be a grammar $G = (V,\Sigma,P,\sigma)$ with productions

$$P = \begin{Bmatrix} \sigma \to () \\ \sigma \to \sigma\sigma \\ \sigma \to (\sigma) \end{Bmatrix}$$

Describe the resulting language L(G), and classify the grammar according to Chomsky's classification.

(7) Let $w = x_1x_2x_3\ldots x_k \in \Sigma^*$ a string of length k. The grammar (V,Σ,P,σ) with $V = \Sigma \cup \{\sigma,A\}$ and productions

$$P = \begin{vmatrix} \sigma \to wA \\ \sigma \to w \end{vmatrix}$$

is context-free. Show that L(G) can be generated by a *regular* grammar G'.

(8) Define a grammar $G = (V,\Sigma,P,\sigma)$, where

$\Sigma = \{\Delta$, who, saw, the, cat, which, chased, rat, went, ?$\}$.

$V = \{\sigma, A, B\} \cup \Sigma$, and P consists of the productions:

$$P = \begin{cases} & \sigma \to AB? \\ & A \to A \text{ which } \Delta \text{ chased } \Delta \text{ the } \Delta \text{ rat } \Delta \\ \text{rat } \Delta & B \to \text{rat } \Delta \text{ which } \Delta \text{ went } B \\ & A \to \text{who } \Delta \text{ saw } \Delta \text{ the } \Delta \text{ cat } \Delta \end{cases}$$

Describe the resulting language L(G).

(9) Construct a context-free grammar, such that the resulting language contains all FORTRAN arithmetic statements (i.e., all FORTRAN statements that use the operators +, -, *, /, **).

(10) Using the grammar of the previous problem, given an explicit derivation of the statement

a - (a*((b + c)/d)**e).

FINITE-STATE AUTOMATA

Intuitive Description of a Finite-State Automaton

The simplest model of a machine is a finite-state automaton. Intuitively, we can think of a finite-state machine as a "black box," which, at discrete time points (e.g., every nanosecond or microsecond), is in exactly one of a finite number of configurations called "states."

Furthermore, at every time point t we provide the machine with an input symbol I(t) (e.g., a binary digit or a letter of the alphabet). Let the machine be in state S(t) at time point t. Then the state S(t + 1) at the following time point t + 1 will depend on the previous state S(t) and the previous input I(t). In functional notation:

$$S(t + 1) = F(S(t), I(t)).$$

If we also assume that the machine gives us output O(t) at each time point t, then the output at time t + 1 depends on the previous state and input signal:

$$O(t + 1) = G(S(t), I(t)).$$

The two functions F and G, which describe the state transition and the output, are intrinsically associated with each machine.

Note that our description contains three major restrictions. First, we assume that the set of internal states of the machine is finite. This happens to be true for all digital computers, but is not the case for analog computers. By restricting the number of states to be finite we have committed ourselves to models of digital computers rather than analog devices.

Second, the state and output at time t do not depend on the input signal, which arrives at the same time (I(t)). This restriction simply means that the input signal at time t needs a finite time to traverse the machine. Since all signals that occur in nature are limited by the speed of light, the second restriction assures us of a realistic model in which all signals cause a delay because of their finite velocity.

Finally, note that the two functions F and G do not depend on the particular time t. In other words, let t_1, t_2 be two distinct time points such that

$$S(t_1) = S(t_2) \text{ and } I(t_1) = I(t_2)$$

(i.e., the machine happens to be in the same state at both times and the input signals at both times are also the same).

Then $$S(t_1 + 1) = S(t_2 + 1)$$

$$O(t_1 + 1) = O(t_2 + 1) .$$

Again, this fact reflects the behaviour of digital computers: running the identical program at different times should result in identical answers.

Thus, every single restriction reflects a condition that actually occurs in the "real" world, so we have no reason to feel guilty about these restrictions.

Definition and Examples of Finite-State Machines

The intuitive model we have just considered has been studied by Mealy and is therefore called the "Mealy model" of a finite-state machine. Its formal definition follows.

Definition 15:

The *Mealy model* of a *finite-state automaton* is a sextuple $(K, \Sigma, O, s_0, f, g)$, where

(1) K is finite set (of *states*)

(2) Σ is a finite set (of *input symbols*)

(3) O is a finite set (of *output symbols*)

(4) $s_0 \varepsilon K$ (s_0 is a *starting state*)

(5) $f : K \times \Sigma \rightarrow K$ (*state-transition function*)

(6) $g : K \times \Sigma \rightarrow O$ (*output function*)

There is another, slightly simpler model by *Moore* in which condition (6) in the previous definition is replaced by

$(6)^1$ $g : K \rightarrow O$ (output function depends only on the state).

It can be shown that every Mealy model can be converted to a Moore model, and vice versa; see exercise (13).

Remarks: The function $f : K \times \Sigma \rightarrow K$, which describes the transition from one state to the next, is also called the *next-stage function.*

Examples:

(26) The first example will be that of a *parity-recognition* machine:
Wc put

$K = \{s_0, s_1\}$ (s_0 is the starting state)

$\Sigma = 0 = \{0,1\}$ (set of binary digits)

$$\left.\begin{array}{l} f(s_0,0) = s_0,\ f(s_1,0) = s_1 \\ f(s_0,1) = s_1,\ f(s_1,1) = s_0 \end{array}\right\} \text{(next-state function)}$$

$g(s_0) = 0,\ g(s_1) = 1$ (output function)

It is usually easier to write the two functions in a two-dimensional table as shown in Table 1.

Table 1. State-transition table for example 26.

States	Input 0	Input 1	Output
	0	1	
s_0	s_0	s_1	0
s_1	s_1	s_0	1

Remember that we are using the Moore model, so the output depends only on the previous state, as shown in Table 1. By tracing the parity-recognition machine for a string consisting of 0's and 1's, one sees that the output is 1 if and only if there is an odd number of 1's in the input string, otherwise the output is 0. Note also that both states change only if the input symbol is 1, i.e., if the parity changes.

Tables such as Table 1 are called *state-transition* tables. The information contained in a state-transition table may also be represented by a *state diagram*, which for our example is shown in Fig. 5. Here

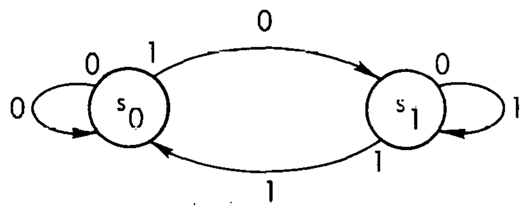


Fig. 5. State diagram for example 26.

states are represented by circles, while arrows represent the transition from one state to the next, depending on the input symbol located at the tail of that particular arrow. The symbol over the center of the arrow designates the output symbol. In the Moore model the output symbols for all arrows leaving the same state are identical!

(27) Let us next construct another machine with $\Sigma = O = \{0,1\}$ such that the output symbol at time t is 1 if and only if the input at times t - 1 and t - 2 is 1. This machine will recognize all input sequences that end in two 1's (this is a case of a machine with a rudimentary ability to "remember" signals that occured two time steps ago). This machine has three states, s_0, s_1, s_2, and the state-transition table is shown in Table 2. Again, s_0 is the starting state.

Table 2. State-transition table for example 27.

States	Input		Output
	0	1	
s_0	s_0	s_1	0
s_1	s_0	s_2	0
s_2	s_0	s_2	1

The corresponding state-diagram is given in Fig. 6.

(28) The following example is that of a *binary adding machine*. In this case we first construct a *Mealy* model, and then find a *Moore* model that does the same thing.

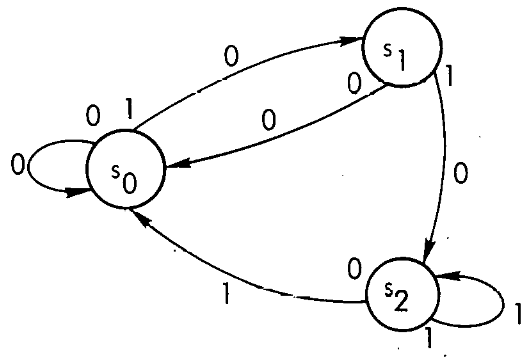


Fig. 6. State diagram for example 27.

We assume that the machine simultaneously receives as input, two strings of binary digits, where each string represents a number in binary form with *least significant digits first.*

The string of output symbols represents the binary sum of the two input numbers, again with least significant digit output first. Thus at some starting moment the machine receives two binary digits (one for each number). That is, the input symbols to our machine are $\Sigma = \{00, 01, 10, 11\}$. In the Mealy model only two states are needed: a "no-carry" state s_0 and a "carry" state s_1. The state-transition table is shown in Table 3.

Table 3. State-transition table for example 28.

States	Input			
	00	01	10	11
s_0	s_0,0	s_0,1	s_0,1	s_1,0
s_1	s_0,1	s_1,0	s_1,0	s_1,1

Since this is a Mealy model, the output depends not only on the state, but also on the input symbol, hence we have written the output symbol after each state in the table. The corresponding state-diagram is shown in Fig. 7. For example, the sum 45 + 57 = 102 has the binary form

$$\begin{array}{r} 101101 \\ \underline{+111001} \\ 1100110 \end{array}$$

Performing this addition would cause the sequence of events depicted in Table 4.

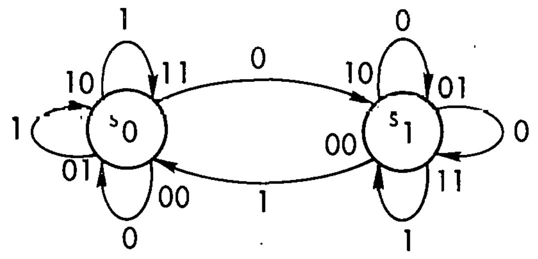


Fig. 7. State diagram for example 28 (Mealy model).

Table 4. Configuration of the Mealy model corresponding to the addition 45 + 57 = 102.

Time t	0	1	2	3	4	5	6	7 . . .
45	1	0	1	1	0	1	0	0 . . .
57	1	0	0	1	1	1	0	0 . . .
Input Signal	11	00	10	11	01	11	00	00 . . .
State	s_0	s_1	s_0	s_0	s_1	s_1	s_1	s_0 . . .
Output Signal	-	0	1	1	0	0	1	1 0 . . .

Note the absence of an output signal at t = 0. Also note that all numbers are presented backwards, least significant digit first (as mentioned before).

Now let us find an equivalent Moore model. First, we represent the major difference of the two models as follows:

Mealy model: $K \times \Sigma \xrightarrow{g} O$

Moore model: $K \times \Sigma \xrightarrow{f} K \xrightarrow{g} O$

For the Moore model to obtain an output signal that depends on the input (as the Mealy model), it must go through two steps: first get the next state using the next-state function f, and then get the output signal using the function g. Since each step requires one time step, we need two time steps to obtain an input-dependent output signal in the Moore model, i.e., the output will be delayed by one time step as compared to the Mealy model. With this consideration in mind, we can write the state-transition table for the Moore model, which requires four states s_0, s_1, s_2, s_3 (see Table 5).

Table 5. State-transition table for the Moore model.

States	Input				Output
	00	01	10	11	
s_0	s_0	s_1	s_1	s_2	0
s_1	s_0	s_1	s_1	s_2	1
s_2	s_1	s_2	s_2	s_3	0
s_3	s_1	s_2	s_2	s_3	1

The corresponding state diagram is given in Fig. 8.
Table 6 shows the table of events for the example 45 + 57 = 102 for the Moore model.

Table 6. Configuration of the Moore model corresponding to the addition 45 + 57 = 102.

Time t	0	1	2	3	4	5	6	7	8 . . . .
45	1	0	1	1	0	1	0	0	0 . . . .
57	1	0	0	1	1	1	0	0	0 . . . .
Input signal	11	00	10	11	01	11	00	00	00 . . . .
State	s_0	s_2	s_1	s_1	s_2	s_2	s_3	s_1	s_0 . . . .
Output signal		0	0	1	1	0	0	1	1 0 . . .

Observe the delay of one time step just discussed. Since the machine starts in state s_0, the first output signal at t = 1 is always 0, independent of the input at time t = 0. We therefore have to regard this first output signal simply as a signal that the machine has started. It is not part of the output sequence we are interested in.

The outcome from example (28) is Theorem 6.

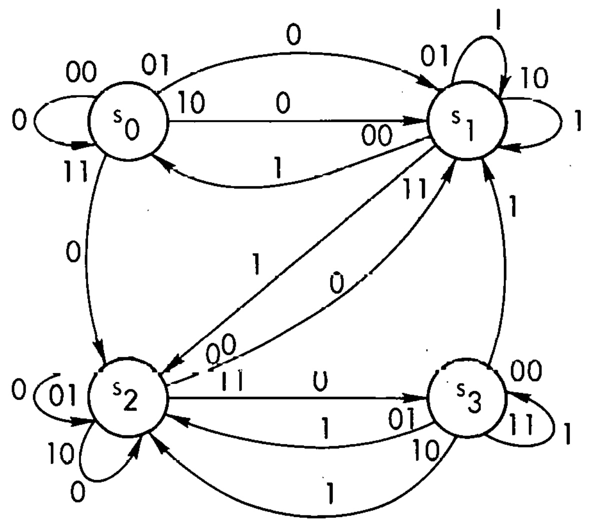


Fig. 8. State diagram for example 28 (Moore model).

Theorem 6:

Finite-state machines are able to add arbitrarily large binary numbers. In contrast to this result we have the following theorem.

Theorem 7:

No fixed finite-state machine can multiply arbitrarily large binary numbers.

Proof:

Let n be the finite number of states of the machine M, and consider the problem of multiplying the number $2^{n + 1}$ by $2^{n + 1}$. Both numbers are represented in binary form by 1, followed by n + 1 0's. Their product, which is $2^{2(n + 1)}$, is represented by 1, followed by 2n + 2 zeros. Since the product contains 2n + 3 digits, and each of the factors contain n + 2 digits, the machine must output n zeros followed by a 1 after the input has stopped. But once the machine has received its input, M operates as a machine with constant input (00). Since it contains only n states, and since it has output n zeros (after the input stopped), it must be caught in a loop. Thus, it has to go on generating zeros forever, contrary to the requirement to output a final 1.

We have here a problem that is beyond solution by any finite-state machine (because the machine is limited to a finite number of states). In fact, to multiply two numbers a and b with $0 \leq a \leq 2^n - 1$ and $0 \leq b < 2^n - 1$, we need a machine with n states (Mealy model!). Note that *two* states sufficed for arbitrary additions.

Languages Accepted by Finite-State Machines

In our three examples of finite-state machines the set O of output symbols consisted each time of the digits 0 and 1. In particular, in examples (26) and (27), the symbol 1 represented the answer "yes," while the symbol 0 stood for the answer "no." The question in example (26) was, "Is there an odd number of ones in the input sequence?" and the question in example (27) was, "Does the input sequence end in two ones?" In each case we could say that the sequence of input symbols is "accepted" by the machine if the output is 1 (yes). Since (for the Moore model) each output symbol is associated with a state, we can make the following definition.

Definition 16:

A state $s \epsilon K$ is called an *accepting* (or *final*) *state*, if

$$g(s) = 1, \text{ where } O = \{0,1\}.$$

Using this definition, the set O of output symbols becomes redundant, since we can declare some of the states as final. Omitting the set O of outputs, and introducing the set F of final states in our definition of a finite-state machine, we arrive at the formal definition of a finite-state acceptor. This serves the purpose of accepting certain strings and rejecting others.

Definition 17:

A *finite-state acceptor* is a quintuple (K,Σ,F,s_0,f), where

(1) K is finite set (of states)

(2) Σ is a finite set (of input symbols)

(3) $F \subset K$ is a subset (of final states)

(4) $s_0 \epsilon K$ (the start state)

(5) $f : K \times \Sigma \rightarrow K$ (the next-state function)

In example (26) we had $F = \{s_1\}$, in example (27) we had $F = \{s_2\}$, and in example (28) (using the Moore model) we had $F = \{s_1,s_3\}$.

Now let us imagine that the set Σ of input symbols is the set of *terminals* for some language. By inputting one symbol at a time, we obtain a string of symbols, i.e., an element of Σ^*. We can now inductively extend our next-state function f to a function $f: K \times \Sigma^* \rightarrow K$ as follows:

(i) $f(s,\varepsilon) = s$ (ε = empty word)

(ii) $f(s,x_1,x_2...x_n) = f(f(s,x_1,x_2...x_{n-1}),x_n)$

where $n \geq 1$, and each $x_i \epsilon \Sigma$.

In fact, this is exactly the way that states change if the machine is presented with a sequence of input symbols $x_1x_2...x_n$. Using this extension of the function f, we can formulate the fundamental definition.

Definition 18:

Let $A = (K,\Sigma,F,s_0f)$ be a finite-state acceptor. Then the set

$$T(A) = \{w \epsilon \Sigma^* / f(s_0,w) \epsilon F\}$$

is the *language accepted by A*. (Here T comes from "tapes," since we may think of a tape that provides the input string to the machine.)

This definition sets up the basic relationship between languages and finite-state machines. Observe that the finite-state acceptor A answers the membership question for the language T(A) (i.e., any string $w \in \Sigma^*$ that is accepted by A belongs to the language T(A), and vice versa).

Kleene's Theorem

We are now in a position to pose the question, "Which languages are accepted by finite-state automata?" The following important theorem answers this question completely.

Theorem 8 (KLEENE's Theorem):

A language is accepted by some finite-state automaton if and only if it is a *regular* language.

Before embarking on the proof of this theorem let us collect a few facts that will be needed during the proof.

First, suppose that M and N are two subsets of Σ^*. Then we define $MN = \{v\,w \,/\, v \in M,\ w \in N\}$ (Concatenation). In particular, if M = N we write M^2 for MM, and $M^3 = MMM$, etc. Also, for $M \subset \Sigma^*$ put

$$M^* = \{\varepsilon\} \cup M \cup M^2 \cup M^3 \cup \ldots\, .$$

Next we supply a different description of the class of regular (type 3) languages.

Proposition 2:

The class of regular languages in Σ^* is the smallest collection of subsets of Σ^* containing all finite subsets and being closed with respect to the three operations union, *, and concatenation. More precisely, let

$$F_0 = \text{collection of finite subsets of } \Sigma^*,$$

$$F_1 = \text{collection of subsets of the form}$$

$$\{S_1 \cup S_2 \,/\, S_1, S_2 \in F_0\} \cup \{S_1 S_2 \,/\, S_1, S_2 \in F_0\} \cup \{S^* \,/\, S \in F_0\}.$$

Inductively, let F_{n+1} be the collection of subsets of Σ^* of the form

$$\{S_1 \cup S_2 \ / \ S_1, S_2 \in F_n\} \cup \{S_1 S_2 \ / \ S_1, S_2 \in F_n\} \cup \{S^* \ / \ S \in F_n\}$$

$(n \geq 1)$. Then Proposition 2 says that $\bigcup_{n=1}^{\infty} F_n$ is the class of regular languages in Σ^*.

Proof:

There are two parts to prove:

(a) $\bigcup_{n=1}^{\infty} F_n$ is contained in the set of regular languages;

(b) the regular languages are contained in $\bigcup_{n=1}^{\infty} F_n$.

Proof of (a):

(i) Every finite set F is a regular language. If $F = \{x_1, x_2, \ldots, x_k\}$, then the productions are simply $S \to x_i$ $(i = 1,2,\ldots,k)$. Since these are type 3 productions, F is regular.

(ii) If L_1 and L_2 are regular, so is $L_1 \cup L_2$. Suppose P_i is the set of productions for L_i, and s_i is the sentence symbol for L_i $(i = 1,2)$. Then the set P of productions for $L_1 \cup L_2$ is simply $P_i \cup P_2$, together with the two productions $s_2 \to s_1$ and $s \to s_2$ (s is the sentence symbol).

(iii) Again, let L_1, L_2 be regular languages with productions P_1, P_2, respectively. To show that L_1L_2 is regular, we modify the productions in P_1 as follows: for every production of the form $X \to x$ we write now $X \to x\ s_2$, where s_2 is the sentence symbol for L_2. Productions of the form $X \to x\ Y$ are not altered. Then, let $\tilde{P}_1$ be this modified set of productions, and put $P = \tilde{P}_1 \cup P_2$; $s = s_1$. Clearly, P is a set of regular productions for the concatenation L_1L_2.

(iv) To show that L^* is regular, assume L is regular, let P be the set of productions for L, and let s be the sentence symbol. Now introduce a new symbol X as an additional nonterminal symbol. The set P is now modified as follows: first we add the production $X \to \alpha$ for every production $s \to \alpha$ (here α may be either a single terminal element a, or a terminal element a, followed by a nonterminal symbol A, i.e., $\alpha = a$ or $\alpha = aA$); second, to every production of the form $A \to x$, where x is a terminal symbol, we add the production $A \to x\,X$. Using the new set $\tilde{P}$ of modified productions, it is easy to see that L^* is regular. This proves part (a).

The proof of part (b) is a little involved and is therefore omitted.

Below are two examples of languages that are accepted by a finite-state automaton.

Examples:

(29) Let $\Sigma = \{a\}$, and let $L = \{a^{2n} \,/\, n \geq 0\}$. Define a finite-state acceptor A as follows:

$$A = (K, \Sigma, F, s_0, f),$$

where $K = \{s_0, s_1\}$, $F = \{s_0\}$, $f(s_0, a) = s_1$, and $f(s_1, a) = s_0$. The reader may verify that $T(A) = L$.

(30) Let $S \subset \Sigma^*$ be a finite subset (i.e., $S \in F_0$, using our previous notation). Define a finite-state acceptor A by:

$$A = (K, \Sigma, F, s_0, f),$$

where:

$K = \{\bar{s}\} \cup \{w \in \Sigma^* \,/\, \exists\, w' \in \Sigma^* \text{ with } ww' \in S\}$

$F = S$ (observe that $S \subset K$, so $F \subset K$)

$s_0 = \varepsilon$ (start state is empty word)

$$f(w,a) = \begin{cases} wa, & \text{if } wa \in K \\ \bar{s}, & \text{if } wa \notin K \end{cases} \qquad (w \in K,\ a \in \Sigma)$$

$f(\bar{s}, a) = \bar{s}$

Note that the set K of states is finite since S if finite. By induction one can show that

$$f(w,w') = \begin{cases} ww', & \text{if } ww' \in K \\ \bar{s}, & \text{otherwise} \end{cases}$$

Hence,

$$w' \in T(A) \Leftrightarrow f(\varepsilon, w') \in F \Leftrightarrow f(\varepsilon, w') \in S \Leftrightarrow w' \in S,$$

so we conclude $T(A) = S$, i.e., every finite subset of Σ^* is a language accepted by a finite-state acceptor.

In our next example we go through various steps to exhibit a language that is not accepted by any finite-state machine.

(31) Let $S \subset \Sigma^*$ be an arbitrary subset. Define an equivalence relation $\sim_S$ among elements of Σ^* by $w \sim_S w'$ if and only if for *all* words $u \in \Sigma^*$ we have $wu \in S \Leftrightarrow w'u \in S$. Clearly, $\sim_S$ is an equivalence relation in Σ^*. Furthermore, if $v \in \Sigma^*$, then $w \sim_S w'$ implies $wv \sim_S w'v$ (by definition).

Denote the equivalence class of w by [w], thus $[w] = \{w' \in \Sigma^* / w \sim_S w'\}$. Let $K_S = \{[w] / w \in \Sigma^*\}$ be the set of equivalence classes in Σ^*. Now define an automaton $A_S = (K_S, \Sigma, F_S, s_0, f)$ by

$F_S = \{[w] / w \in S\}$
$s_0 = [\varepsilon]$ (class of empty word)
$f([w],a) = [wa]$ for $[w] \in K_S$, $a \in \Sigma$

More generally, the next-state function is $f([w],w') = [ww']$. Note that A_S is in general an infinite-state automaton, since K_S might be an infinite set.

Proposition 3:

A_S is the minimal state automaton that accepts the language S.

Proof:

There are two parts to the proof:

(a) A_S accepts S, i.e., $T(A_S) = S$,
(b) If A accepts S, then A has "more" states than A_S.

(a) Let $w \in \Sigma^*$, then we have:

$$w \in T(A_S) \Leftrightarrow f([\varepsilon],w) \in F_S \Leftrightarrow [w] \in F_S \Leftrightarrow w \sim_S w' \ (w' \in S) \Leftrightarrow w \in S.$$

Therefore $T(A_S) = S$.

(b) Let $A = (K, \Sigma, F, p_0, f')$ be an automaton that also accepts S. Define a mapping $\phi: \Sigma^* \to K$ by $\phi(w) = f'(p_0, w)$. Suppose $\phi(w) = \phi(w')$ for $w, w' \in \Sigma^*$. Then $\phi(wu) = \phi(w'u)$ for all $u \in \Sigma^*$, since $\phi(wu) = f'(p_0,wu) = f'(f'(p_0,w),u) = f'(p_0,w'u) = \phi(w'u)$.

Now $wu \in S \Leftrightarrow wu \in T(A) \Leftrightarrow f'(P_0,wu) \in F \Leftrightarrow \phi(wu) \in F \Leftrightarrow \phi(w'u) \in F \Leftrightarrow w'u \in S$,

hence, $\phi(w) = \phi(w')$ implies $w \sim_S w'$, i.e., $\phi(w) = \phi(w')$ implies $[w] = [w'] \in K_S$.

We can therefore define a mapping $\Psi: \phi(\Sigma^*) \to K_S$ by $\Psi(\phi(w)) = [w]$. Since Ψ is onto, we have

$$|K_S| \leq |\mathrm{Im}\,\phi| \leq |K| \quad \text{(cardinalities)}.$$

Corollary 1:

If $S = T(A)$ for some finite-state machine A, then K_S is finite.

Corollary 2:

$S = \{a^n b^n \,/\, n > 0\}$ is a language that is not accepted by any finite-state automaton.

Proof:

For $i \neq j$ we have $a^i \not\sim_S a^j$, for if $i < j$, then $a^i b^i \in S$, but $a^j b^i \notin S$, contradicting the definition of $\sim_S$. Consequently, there are infinitely many equivalence classes $[a^i]$, $i = 1,2,\ldots,$ and so K_S is infinite.

Remark: The language $\{a^n b^n \,/\, n > 0\}$ is context-free, given by the productions

$S \to ab$

$A \to aSb$

In this chapter we have obtained two results which exhibit the limitations of finite-state machines: they cannot multiply arbitrary numbers, and they can only accept regular languages. The first result is a consequence of the finiteness of the number of states, while the second result is in part due to the simple nature of the next-state function, which simply does not allow for the recognition of more complicated languages.

<u>Proof of Kleene's Theorem</u>

The proof of Kleene's theorem naturally falls into two parts: we have to show (a) that for any finite-state acceptor A there is a regular language L such that A accepts L, i.e., $L = T(A)$, and (b) that for any regular language we can construct a finite-state acceptor that accepts the language. Part (a) is relatively straightforward. Starting with a finite-state acceptor we simply construct a set of productions that define a regular language, which in turn is accepted by the given machine. To prove part

(b) we have to do a little more work. In particular we have to define the notion of a nondeterministic finite-state machine. Let us therefore prove (a) first.

Let $A = (K,\Sigma,F,s_0,f)$ be a finite-state acceptor, and let us denote its states by $s_i (i = o,\ldots,n)$ so $K = \{s_0,s_1,s_2,\ldots,s_n\}$. To define the language L accepted by A, recall that the set $V \setminus \Sigma$ was the set of nonterminal elements, containing as the sentence symbol the element σ_0.

We shall construct $V \setminus \Sigma$ so that it is simply a copy of the set K, i.e., for every state $s_i \in K$, we want a nonterminal symbol $\sigma_i \in V \setminus \Sigma$. In particular, the sentence symbol σ_0 corresponds to the start state s_0. Then $V \setminus \Sigma = \{\sigma_0,\sigma_1,\sigma_2,\ldots,\sigma_n\}$. Let us further denote the input symbols as follows: let $x_{ij} \in \Sigma$ be the input symbol for which $f(s_i,x_{ij}) = s_j$, so x_{ij} is the input symbol that causes the machine to change from state s_i to state s_j. Now we can construct the set P of production rules for our language L.

Let the productions in P be

$$\sigma_i \to x_{ij}\sigma_j, \text{ if and only if } f(s_i,x_{ij}) = s_j$$
$$\sigma_i \to x_{ij} \quad, \text{ if and only if } f(s_i,x_{ij}) \in F.$$

Let L be the language defined by the grammar $G = (V,\Sigma,P,\sigma_0)$.

To show that $T(A) = L$, we have to verify two things: (i) $T(A) \subset L$, and (ii) $L \subset T(A)$. Both verifications are straightforward in view of the close resemblance of the productions and the next-state function, and are left as an exercise.

We have therefore shown that for every finite-state acceptor A, there exists a regular language that is accepted by A.

To prove part (b) we have to start with a regular language L and find a finite-state acceptor A that accepts L. To do this, we introduce the concept of nondeterministic finite-state acceptor.

Definition 19:

A nondeterministic finite-state acceptor is a quintuple $A = (K,\Sigma,F,S_0,f)$, where

(1) K is finite set (of states)

(2) Σ is a finite set (of input symbols)

(3) F is a subset of K (of final states)

(4) S_0 is a subset of K (of start states)

(5) f: $K \times \Sigma \to 2^K$ (the next-state function) - $2^K \underset{\text{def}}{=} P(K)$=power set of K

Observe two major differences to the notion of a (deterministic) finite-state acceptor: first, there can be more than one starting state; second, the next-state function is not unique, but has as values subsets of the set of states (hence the name "nondeterministic"). Thus, given a state s and an input signal $x \in \Sigma$, f(s,x) is a set of possible next states. We extend f to a function

$f = K \times \Sigma^* \to 2^K$ be defining

$f(s,\varepsilon) = s$ (ε = empty word)

$f(s,x_1x_2...x_n) = \bigcup_{s'} \{f(s',x_n)/s' \in f(s,x_1x_2...x_{n-1})\}$

Furthermore we can define $f : 2^K \times \Sigma^* \to 2^K$ by $f(K',w) = \bigcup_{s' \in K'} f(s',w)$ for $K' \subset K$. Then one verifies that $f(K',uv) = f(f(K',u),v)$.

Definition 20:

Let A be a nondeterministic finite-state automaton. Then the set $T(A) = \{w \in \Sigma^* / f(S_0,w) \cap F \neq \emptyset)\}$ is the *language accepted by A.*

It turns out that a string $w = x_1x_2...x_n$ is in T(A) if and only if there is a sequence $s_0,s_1,...,s_n$ of states with $s_0 \in S_0$ such that $s_i \in f(s_{i-1},X_i)$ and $s_n \in F$. In other words, a string $w = x_1x_2...x_n$ is accepted by A if there is at least one sequence of states, consistent with the input string, that leads the machine to an accepting state.

Example:

(32) Let $A = (K,\Sigma,F,S_0,f)$ be a nondeterministic finite-state acceptor, where $\Sigma = \{0,1\}$, $K = \{s_0,s_1,s_2.s_3\}$, $F = \{s_3\}$, and f is given by the next-state table in Table 7.

Table 7. The next-state table for example 32.

States	Input 0	Input 1
s_0	s_1, s_2	s_1, s_3
s_1	s_0	s_1, s_3
s_2	$\emptyset$	s_0,s_2
s_3	s_0, s_3	s_0, s_1, s_2

For example, the input sequence w = 0110 gives rise to the "state tree" shown in Fig. 9.

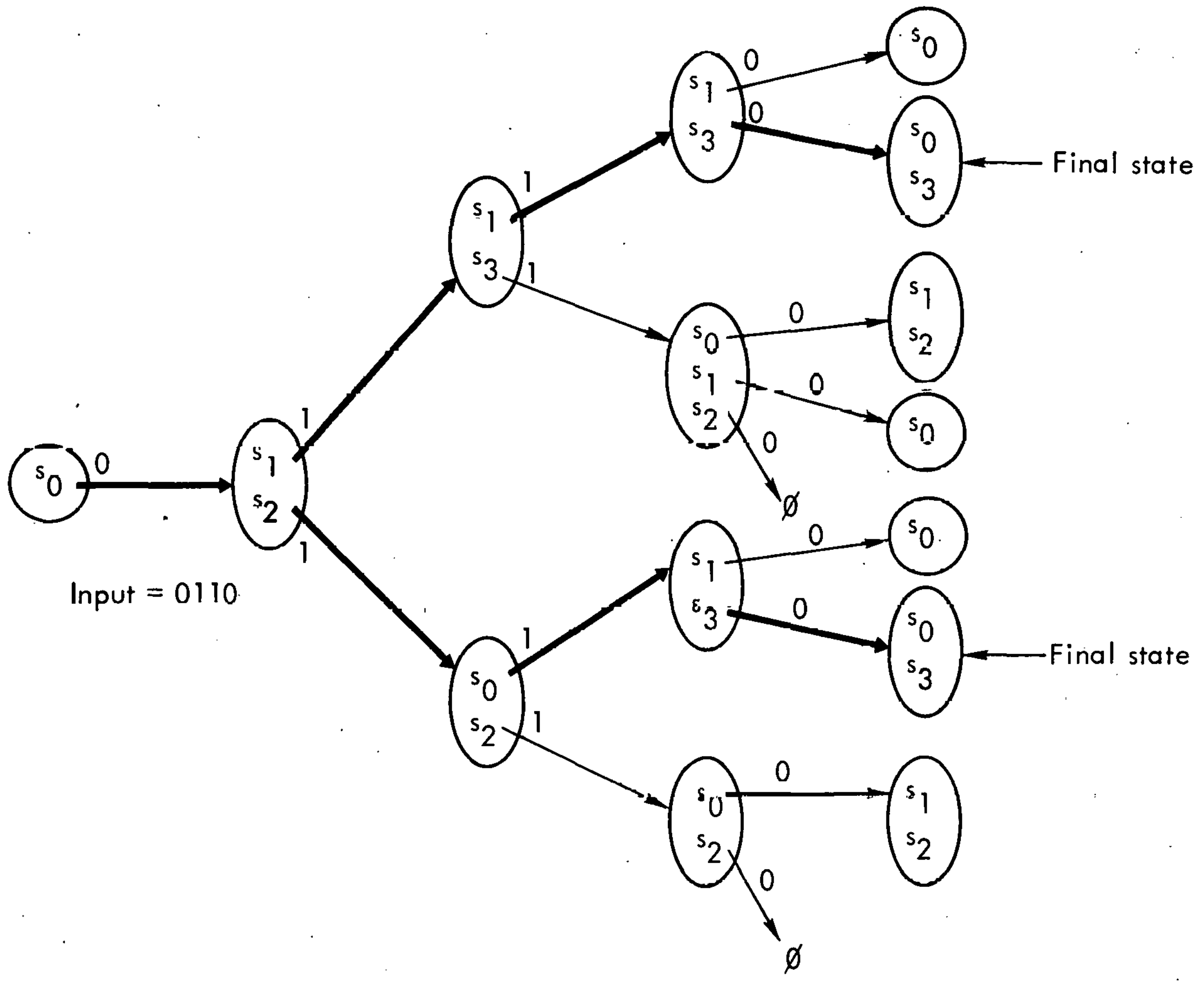


Fig. 9. State tree for example 32 when the input sequence is w = 0110.

Note that 0110 is accepted by A, since there are two possible branches that lead to a final state S_3.

Since nondeterministic finite-state automata are more general than deterministic ones, one might expect that they are more powerful in the sense that they accept more strings than deterministic automata do. The following result shows that this is not the case.

Proposition 4:

Let $A = (K,\Sigma,F,S_0,f)$ be a nondeterministic finite-state machine. Then there exists a deterministic finite-state automaton $D(A)$ such that $T(A) = T(D(A))$.

This proposition shows that any language accepted by a nondeterministic finite-state (F-s) machine A can also be accepted by a deterministic F-S machine D(A). That is, nondeterministic F-S machines have no more power than deterministic ones!

Proof of proposition 4:

Define $D(A) = (K',\Sigma,F',s_0',f')$ by
$K' = 2^K$ (= set of subsets of K),
$F' = \{\tilde{K} \subset K \ / \ \tilde{K} \cap F \neq \emptyset\}$
$s_0' \in K'$ is the subset $S_0 \in 2^K$,
$f'(\tilde{K},x) = f(\tilde{K},x)$ for $\tilde{K} \subset K$, $x \in \Sigma$.

Then, if $w \in \Sigma^*$ we have

$$w \in T(A) \Leftrightarrow f(S_0,w) \cap F \neq \emptyset \Leftrightarrow f'(S_0,w) \cap F \neq \emptyset$$
$$\Leftrightarrow f'(S_0,w) \in F' \Leftrightarrow w \in T(D(A)).$$

Continuing now with part (b) of the proof of Kleene's theorem, we shall make use of proposition 2. Beginning with that proposition, we can prove part (b) if we can show four facts:

(i) finite languages are accepted by F-S automata;
(ii) if L_1 and L_2 are accepted by a F-S automaton, then so is $L_1 \cup L_2$;
(iii) if L_1 and L_2 are accepted by F-S automata, then so is their "product" L_1L_2;
(iv) if L is accepted by a F-S automata, then so is L^*.

(i) By example (30) on page 45, we know that every finite language is accepted by a finite-state acceptor.

(ii) Suppose $L_1 = T(A_1)$ and $L_2 = T(A_2)$, where $A_1 = (K_1,\Sigma,F_1,s_0,f_1)$ and $A_2 = (K_2,\Sigma,F_2,s_0',f_2)$. Define a new (deterministic) F-S acceptor $A = (K_1 \times K_2,\Sigma,K_1 \times F_2 \cup F_1 \times K_2,(s_0,s_0'),f)$,

where $f((s_1,s_2),x) = (f_1(s_1,x),f_2(s_2,x))$, for $s_1 \in K_1$, $s_2 \in K_2$, $x \in \Sigma$.

Then, for any word $w \in \Sigma^*$: $f((s_1,s_2),w) = (f_1(s_1,w), f_2(s_2,w))$.

Furthermore, $w \in T(A) \Leftrightarrow f((s_0,s_0'),w) \in F$, where $F = K_1 \times F_2 \cup F_1 \times K_2$.

But, $f((s_0,s_0'),w) = (f_1(s_0,w), f_2(s_0',w)) \in K_1 \times F_2 \cup F_1 \times K_2$, if and only if $f_1(s_0,w) \in F_1$ *or* $f_2(s_0',w) \in F_2$. That is, $w \in T(A)$ if $w \in T(A_1) \cup T(A_2)$. Thus $L_1 \cup L_2 = T(A)$.

(iii) Again, let $L_1 = T(A_1)$, $L_2 = T(A_2)$, where A_1, A_2 are defined as in (ii) with the additional condition that $K_1 \cap K_2 = 0$. Define a nondeterministic F-S automaton

$$A = (K_1 \cup K_2,\Sigma, F_2, S_0, f)$$

where,

$$S_0 = \{s_0,s_0'\}, \text{ if } s_0 \in F_1$$
$$S_0 = \{s_0\}, \text{ if } s_0 \notin F_1.$$

For $s \in K_1$, let

$$f(s,x) = \begin{cases} \{f_1(s,x)\}, \text{ if } f_1(s,x) \notin F_1 \\ \{f_1(s,x), s_0'\}, \text{ if } f_1(s,x) \in F_1 \end{cases}$$

For $s \in K_2$ let $f(s,x) = f_2(s,x)$ always.

Observe that the start-state for A is the same as that for A_1 (at least if $s_0 \notin F_1$). Also, the next-state function f is actually unique except in one situation: if the machine A is in a state of K_1 and receives an input symbol x that would lead to a final state in A_1. In that situation A can choose to go either to that final state in K_1 *or* to the start state s_0' of A_2. If now $w \in \Sigma^*$ is accepted by A, then it has to end up in F_2, i.e., in a final state of A_2. As we just saw, this is only possible if an initial substring of w leads to a final state in A_1 - that is, if and only if $w_1 \in T(A_1)$.

If instead of going to that final state, A chooses s_0' as its next state, then the rest of the string (the substring following w_1) must lead to a final state in F_2, i.e., the substring following w_1 must be in $T(A_2)$. Thus, $w = w_1w_2$ with $w_1 \in T(A_1)$, $w_2 \in T(A_2)$ if and only if $w \in T(A)$.

Suppose $w_1 = x_1x_2 \ldots x_n$, while $w_2 = y_1y_2 \ldots y_m$. Then we can graphically represent the sequence of states through which A passes as in Fig. 10. Thus, $L_1L_2 = T(A_1)T(A_2) = T(A)$.

(iv) Suppose $L = T(A)$ with $A = (K,\Sigma,F,s_0,f)$. Construct a nondeterministic F-S automaton

$$A' = (K \cup \{s_0'\}, \Sigma, F \cup \{s_0'\}, \{s_0'\}, f')$$

where $f'(s_0', x) = \{f(s_0, x)\}$,

$$f'(s,x) = \begin{cases} \{f(s,x)\}, & \text{if } f(s,x) \notin F \\ \{f(s,x), f(s_0,x)\}, & \text{if } f(s,x) \in F \end{cases}$$

The idea in constructing this automaton is quite similar to that used in (iii). The extra state $\{s_0'\}$ has been added to make sure that ε (the empty word) is accepted by A' (any F-S automaton whose start state is also a final state will accept the empty word ε). Again, f' is unique except in the situation in which a word has been accepted by A - then f' can either choose the final state in F as the next state *or* it can go to $f(s_0,x)$ and start all over again. Thus A' accepts any word $w = w_1w_2 \ldots w_n$ with $w_i \in T(A)$, n = 0, 1, 2,... Consequently, $L^* = T(A')$, which proves (iv). We have completed the proof of Kleene's theorem.

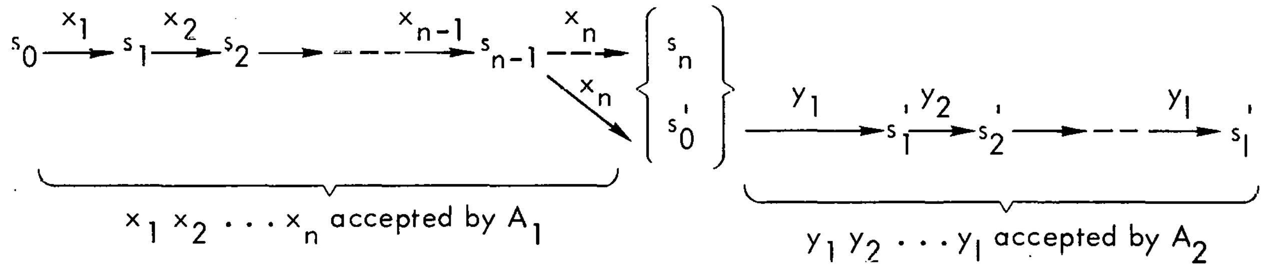

$s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \to --- \xrightarrow{x_{n-1}} s_{n-1} \xrightarrow{x_n}$
s_n
s_0'
$\xrightarrow{y_1} s_1' \xrightarrow{y_2} s_2' \to --- \xrightarrow{y_l} s_l'$
$x_1 x_2 \ldots x_n$ accepted by A_1
$y_1 y_2 \ldots y_l$ accepted by A_2

Fig. 10. Possible sequence of states for a nondeterministic automaton A, which accepts L_1L_2.

We might mention that part (b) of our proof can also be proved by starting with a regular language L and using the productions of L to construct a nondeterministic automaton A(L) that accepts L: L = T(A(L)). Given the regular grammar $G = (V,\Sigma,P,\sigma)$ we construct the nondeterministic acceptor $A = (K,\Sigma,F,s_0,f)$ as follows:

Let $K = (V \setminus \Sigma) \cup \{X\}$, so the states of A are in one-to-one correspondence with the nonterminals $V \setminus \Sigma$, plus one additional state X. Let $s_0 = \sigma$ be the start state. If P contains the production $\sigma \to \varepsilon$, define $F = \{s_0, X\}$, otherwise let $F = \{X\}$. The next-state function f is given by:

(i) $X \in f(s,a)$, if and only if $s \to a$

(ii) $s' \in f(s,a)$, if and only if $s \to as'$

(iii) $f(X,a) = \emptyset$ for all $a \in \Sigma$

One can show that for this automaton

$$T(A) = L(G).$$

This concludes our discussion of finite-state automata.

Syntax Trees and Ambiguity

In the following, we restrict our considerations to *context-free* grammars. Consider the grammar $G = (V,\Sigma,P,\sigma)$ given by $V = \Sigma \cup \{\sigma,A,B,C\}$, $\Sigma = \{s,t,x,y,z\}$, and the productions P:

$$P = \begin{Bmatrix} \sigma \to x\sigma A \\ A \to BC \\ \sigma \to y \\ B \to z \\ C \to st \end{Bmatrix}$$

The sentence $xyzst \in L(G)$ can be derived by $\sigma \to x\sigma A \to x\sigma BC \to x\sigma Bst \to x\sigma zst \to xyzst$. This sequence can be represented geometrically in the form of a *syntax tree* (or *derivation tree*) (see Fig. 11).

In a context-free grammar the order in which the terminal symbols in a sentential form are derived is not important. The following different derivations of xyzst are all represented by the same tree

$$\sigma \to x\sigma A \to xyA \to xyBC \to xyzC \to xyzst$$

or

$$\sigma \to x\sigma A \to x\sigma BC \to xyBC \to xyBst \to xyzst$$

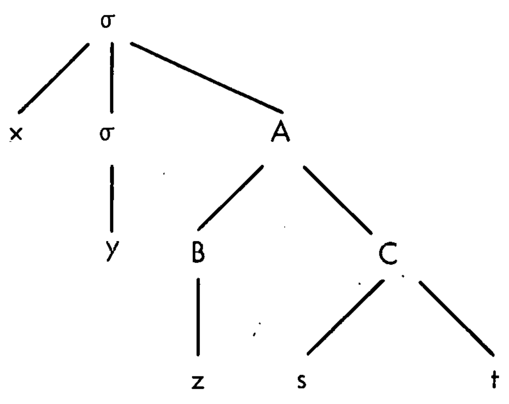


Fig. 11. Syntax tree for a context-free grammar.

Two derivations of a sentence $w \in L(G)$ are considered *equivalent* if they lead to identical syntax trees.

Figure 12 is the syntax tree corresponding to all equivalent derivations of the string $xxyzstzst \in L(G)$.

Consider next the grammar $G' = (V',\Sigma',P',\sigma')$, where $\Sigma' = \{+\}$, $V' = \Sigma' \cup \{\sigma',A,B\}$, and P' is

$$P' = \left\{ \begin{array}{l} \sigma' \to A \\ A \to B \\ A \to + \\ B \to A \end{array} \right\}$$

Obviously, there are many different derivations for the word +, e.g.,

$$\sigma' \to A \to +$$

or $$\sigma' \to A \to B \to A \to +$$

or $$\sigma' \to A \to B \to A \to B \to A \to +$$

Their derivation trees are given in Fig. 13. Since all three trees are different from each other, the three derivations are nonequivalent.

Definition 21:

If $G = (V,\Sigma,P,\sigma)$ is a context-free grammar, and $w \in L(G)$, then we call the string w *ambiguous in G* if there exist at least two nonequivalent derivations of w (i.e., if there exist at least two derivations of w with different derivation trees).

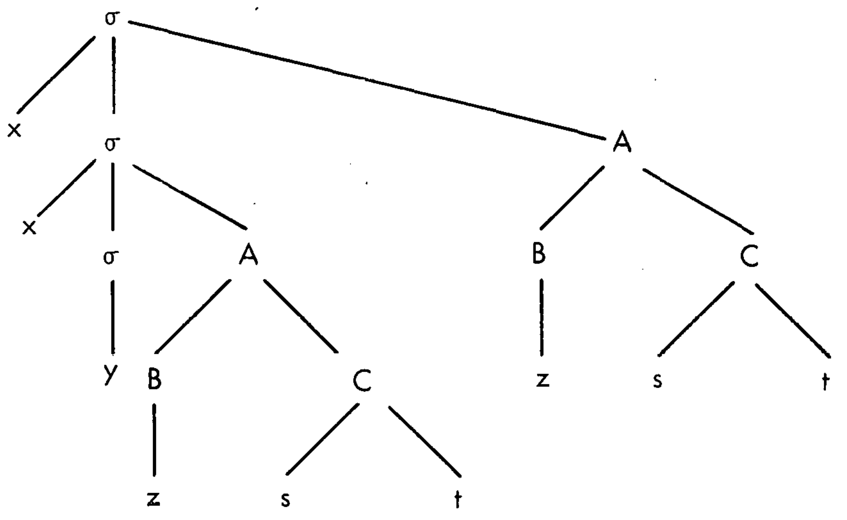


Fig. 12. Syntax tree for equivalent derivations of the sentence xxyzstzst.

Thus, in the grammar G' above, the string "+" is ambiguous in G' (see Fig. 13).

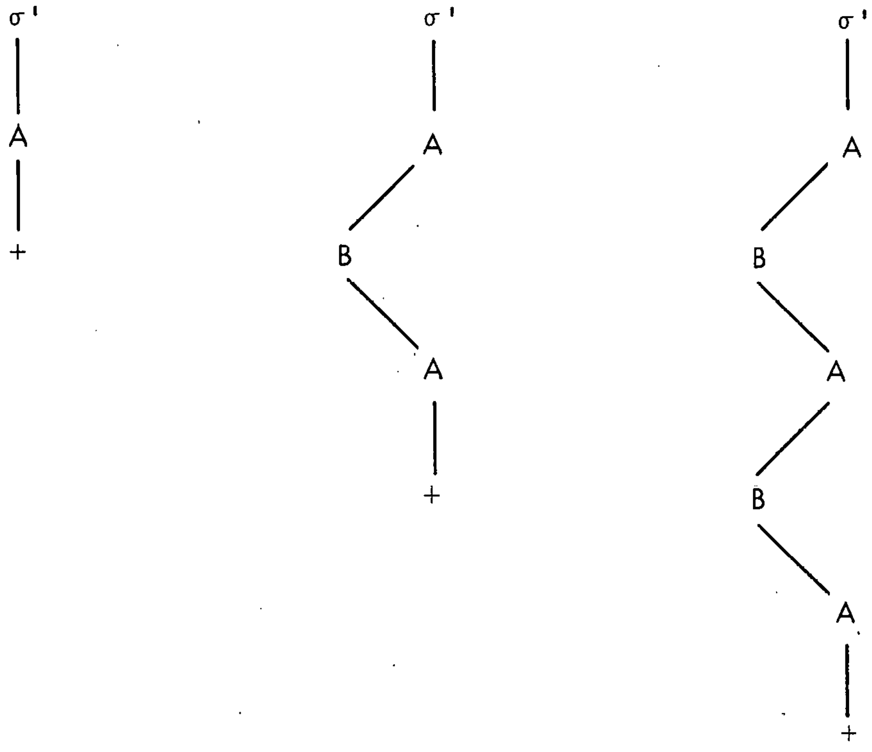


Fig. 13. Syntax trees for three nonequivalent derivations for the word +.

Definition 22:

A grammar G is *ambiguous* if there exists at least one string $w \in L(G)$ that is ambiguous in G.

Note that a language L could be ambiguous when described by one grammar G_1, but unambiguous when described by another grammar G_2.

Definition 23:

A language is *inherently ambiguous* if it cannot be described by an unambiguous grammar.

Example:

(33) The language $L(G') = \{+\}$ in the example above is not inherently ambiguous, since we can also describe it using the following unambiguous grammar G":

$G'' = (V',\Sigma',P'',\sigma')$, where V',Σ',σ' are as above, but P" consists of $P'' = \{\sigma \rightarrow A, A \rightarrow B, B \rightarrow +\}$.

An ambiguous English sentence is, "In all books examined sentences were not ambiguous." The ambiguity results from the fact that "examined" may either go with "books" or with "sentences." Since either choice leads to a different meaning, there can be no grammar which resolves this ambiguity, i.e., the *English language is inherently ambiguous.*

Here is another example of an ambiguous grammar:

$G = (V,\Sigma,P,\sigma)$,

where $\Sigma = \{+,*,a,b\}$, $V = \Sigma \cup \{\sigma\}$, and P is

$$P = \begin{Bmatrix} \sigma \rightarrow \sigma + \sigma \\ \sigma \rightarrow \sigma * \sigma \\ \sigma \rightarrow a \\ \sigma \rightarrow b \end{Bmatrix}$$

Tow nonequivalent derivations of $a + b * a + b$ are

(i) $\sigma \rightarrow \sigma + \sigma \rightarrow \sigma + \sigma * \sigma \rightarrow \sigma + \sigma * \sigma + \sigma \rightarrow \ldots \rightarrow a + b * a + b$

(ii) $\sigma \rightarrow \sigma * \sigma \rightarrow \sigma + \sigma * \sigma \rightarrow \sigma + \sigma * \sigma + \sigma \rightarrow \ldots \rightarrow a + b * a + b$

The syntax trees are given in Fig. 14. This language is unambiguous, since the following unambiguous grammar $\tilde{G}$ describes it also:

$\tilde{G} = (\tilde{V},\Sigma,\tilde{P},\sigma)$, where $\tilde{V} = \Sigma \cup \{\sigma,A\}$, and
$\tilde{P} = \{\sigma \to A + \sigma,\ \sigma \to A,\ A \to A * A,\ A \to a,\ A \to b\}$.

In $\tilde{G}$, the string a + b * a + b is unambiguous, since any derivation is equivalent to

$$\sigma \to A + \sigma \to A + A + \sigma \to A + A + A \to A + A * A + A \to \dots \to a + b * a + b$$

The syntax tree is shown in Fig. 15. The question of whether or not a grammar is ambiguous is quite important when applied to programming languages, most of which are context-free (or nearly context-free). If the grammar describing a programming language is ambiguous, then there are at least two ways to interpret some sentence in the language. In particular the compiler might interpret the sentence differently than the programmer, and the resulting code would do something different than the programmer had intended. Unfortunately, the problem of whether a grammar is ambiguous is unsolvable for context-free grammars.

Theorem 9:

(1) The ambiguity problem is unsolvable for context-free grammars.
(2) For regular grammars the ambiguity problem is solvable.

A related question is whether or not a given language is inherently ambiguous. Here we state the following results.

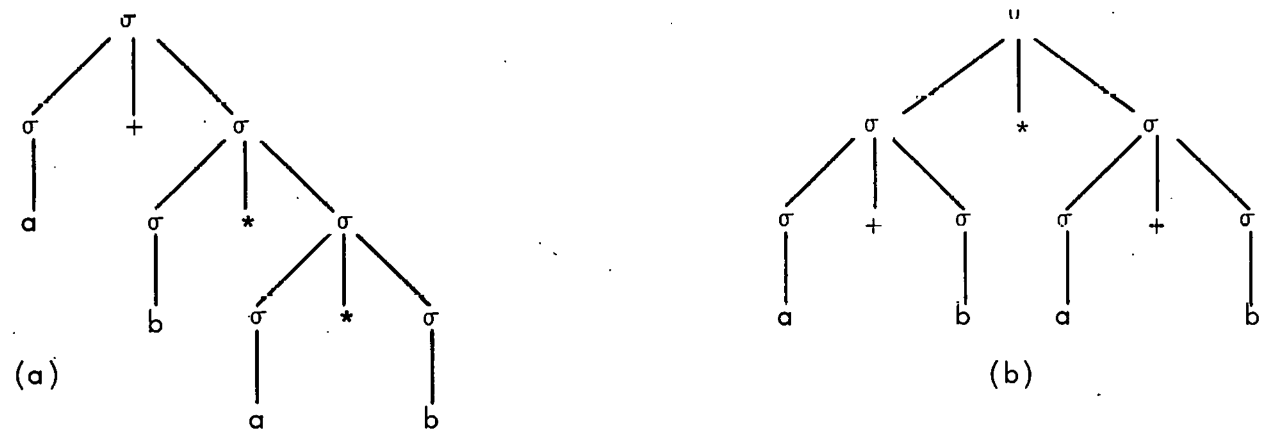


Fig. 14. Syntax trees fcr two nonequivalent derivations for a + b * a + b.

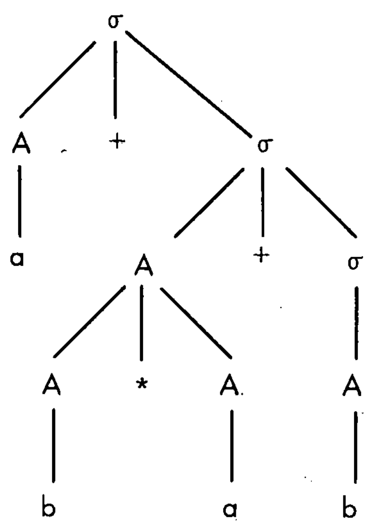


Fig. 15. Syntax tree for deriving a + b * a + b in the unambiguous grammar $\tilde{G}$.

Theorem 10:

The inherent ambiguity problem is unsolvable for context-free languages.

Theorem 11:

If a language L is accepted by a deterministic machine, then L is unambiguous.

Corollary 3:

There are no regular languages that are inherently ambiguous.

Corollary 4:

The inherent ambiguity problem is (trivially) solvable for regular languages.

There exist inherently ambiguous context-free languages. An example is the language $L = \{a^i b^j c^k \ / \ i = j \text{ or } j = k\}$. In general, it is quite hard to decide whether a language is inherently ambiguous, since this decision must be based on some analysis of *all* grammars that can describe the given language!

Exercises

(11) Draw the state-diagram for a Mealy-automaton that can subtract binary numbers (least significant digits input first).

(12) Describe a finite-state acceptor able to accept all strings in $\Sigma^* = \{0,1\}^*$, such that every 0 is immediately followed by a 1. (Thus, 11, 01, 10111, 101011 are all acceptable, but 0, 11001, 10 are not.)

(13) Using the definition below of equivalence for a Moore and Mealy automaton, construct for every Moore machine an equivalent Mealy machine, and vice versa. (Going from Mealy to Moore involves an increase in the number of states.)

Let $A = (K,\Sigma,0,s_0,f,g)$ be a Mealy machine, and $B = (K',\Sigma,0,s_0',f',g')$ a Moore machine with the same input and output alphabets as A.

Define recursively for A:

$$f(s,\varepsilon) = s$$

$$f(s,x_1,x_2,\ldots x_n) = f(f(s,x_1,x_2,\ldots,x_{n-1}),x_n)$$

$$g(x_1) = g(s_0,x_1)$$

$$g(x_1,x_2,\ldots,x_n) = g(f(s_0,x_1,x_2,\ldots,x_{n-1}),x_n)$$

Similarly, for the Moore machine B we define:

$$f'(s',\varepsilon) = s'$$

$$f'(s',x_1,x_2,\ldots,x_n) = f'(f'(s',x_1,x_2,\ldots,x_{n-1}),x_n)$$

$$g'(x_1) = g'(f'(s_0',x_1))$$

$$g'(x_1,x_2,\ldots,x_n) = g'(f'(s_0',x_1,x_2,\ldots,x_n))$$

Now we call A and B equivalent if and only if $g(x_1\ldots x_n) = g'(x_1\ldots x_n)$ for all strings $w = x_1\ldots x_n \in \Sigma^*$, i.e., if the same input string yields the same output string in both A and B.

(14) Let $A = (K,\Sigma,F,s_0,f)$ be a finite-state acceptor that accepts a certain language L, i.e., $T(A) = L$. We define the "reversed" language L^R by $L^R = \{w^R / w \in L\}$, so L^R contains the reversals of all strings in L. (For example, if $L = \{001, 1011, 100111\}$, then $L^R = \{100, 1101, 111001\}$.)

Construct a nondeterministic finite-state acceptor $A^R = (K',\Sigma,F',s_0',f')$, such that it accepts L^R: $T(A^R) = L^R$.

(15) Apply the above construction of A^R to the finite-state acceptor in example (27) (put $F = \{s_2\}$ there), so A^R accepts all strings starting with two 1's.

TURING MACHINES

Discussion, Definition, and Examples

In this chapter we study a class of machines that were invented by Alan M. Turing in 1936 and now bear his name.

While finite-state automata are quite simple machines - they are not even able to multiply arbitrary numbers - Turing machines are probably the most complex and powerful machines. In fact, Turing in his original paper asserted that his machine is able to perform *any* calculation that can possibly be performed by *any* machine! This assertion cannot be proved mathematically since there is no precise mathematical definition of "calculations that can be performed by a machine."

We enter here into the general problem of describing precisely what problems can be solved by mechanical processes, by machines. Are there processes that can be precisely described but cannot be realized by machines? Mathematicians use the term "effective procedure" or "algorithm" for processes that should be solvable by machines (e.g., that can be programmed). One could say, "An effective procedure is a set of rules that tell us (or the machine), from moment to moment, precisely how to behave (e.g., there are at each step no choices left open; there is at each step a unique instruction to follow)." In particular, any computer program is an effective procedure, and most people agree intuitively that only effective procedures can be handled by machines. *Turing's thesis* - also known as *Church's thesis* - can now be stated as follows:

Any effective procedure can be handled by a Turing machine.

As we have mentioned, this assertion cannot be proved rigorously because the notion "effective procedure" is only defined intuitively. The best we can do is believe Turing or disbelieve him. One should know, however, that so far no effective procedure has been found that could not be handled by a Turing machine. In fact, various different mathematicians (e.g., Church, Post, and Kleene) have arrived at the same conclusion from quite different considerations - and that fact is quite compelling.

What is particularly surprising about the thesis that Turing machines can handle *all* effective procedures, is the relatively simple definition of Turing machines.

Definition 24:

A *Turing machine* consists of a finite-state control unit which controls the motion of a read/write head that scans an infinite tape. The infinite (linear tape) is divided into "squares" or "cells," each containing a *single* symbol chosen from a finite set $\Gamma = \{b\} \cup \Sigma$, where "b" denotes a "blank" symbol, and Σ is a finite set of tape symbols (nonblank, of course). The read/write head is located over a square on the tape at each time point. Depending on the input symbol in that square and the present state of the control unit, the machine acts as follows:

(i) a new symbol is written on the tape in the square under the head,

(ii) the head then moves either one square to the left or one square to the right,

(iii) the control unit enters a new state,

(iv) the machine might halt.

A few remarks are in order. First of all, we may assume without any loss in generality that the symbol written by the Turing machine on a tape square is nonblank. Also, when a symbol is written on a square, it erases the symbol that previously occupied that square. Because the machine can move either way along the tape, it is possible for it to return to a previously printed location to recover the information inscribed there. We shall see later that this makes it possible to use the tape for the storage of an arbitrarily large (finite) amount of information.

We will also make the restriction that when the machine is started, the tape must be blank except for a *finite* number of squares. This means that we have an *infinite* storage medium available, but at each time there is only a *finite* amount of information stored on it.

To describe the behavior of a Turing machine, we observe that each pair (state, tape symbol) gives rise to:

(a) a new *state*,
(b) a new *tape symbol*,
(c) a *head motion* L (left) or R (right)

Thus, a Turing machine may be described by quintuples (s_i, x_j, s_{ij}, x_{ij}, m_{ij}), where

s_i = current state,

x_j = tape symbol currently under the read/write head,

s_{ij} = new state,

x_{ij} = new tape symbol (replaces x_j),

m_{ij} = head motion, i.e., $m_{ij} \epsilon \{L,R\}$

If s_{ij} = H, we mean that the machine halts. In this case there will be no head motion, so we write m_{ij} = –. Some authors use the term *instantaneous description* for quintuples.

Let us look at some examples now.

Examples:

(34) This example presents a *parity recognition* machine like the finite-state machine in example (26) on page 00. The tape contains blank squares except for a finite sequence of squares containing zeros and ones. In its starting state s_0, the machine has its head over the leftmost nonblank square on the tape. Just as in the case of a finite-state machine, we have two states s_0 and s_1, and the machine changes state only if a one is encountered on the tape. The "state-transition" table in terms of quintuples is shown in Table 8.

Table 8. State-transition table for example 34.

s_i	x_j	s_{ij}	x_{ij}	m_{ij}
s_0	0	s_0	0	R
s_0	1	s_1	0	R
s_0	b	H	0	–
s_1	0	s_1	0	R
s_1	1	s_0	0	R
s_1	b	H	1	–

In this example, the original input sequence has been erased by the machine. Also, if it contained an odd number of ones, a one is written into the first blank square after the input sequence; otherwise a zero is written. In both cases the machine comes to a halt.

Note that the head always moves in one direction (R), so there is no possibility of recording information on the tape and returning to it at a later time. Hence one could not expect it to do anything that could not be done by a simple finite-state machine.

Suppose now that a machine satisfies the following condition: $Q_1 = (s_i, x_j, s_{ij}, x_{ij}, m_{ij})$ and $Q_2 = (s_k, x_\ell, s_{k\ell}, x_{k\ell}, m_{k\ell})$ are two quintuples such that $s_{ij} = s_{k\ell}$ (i.e., in both situations Q_1 and Q_2, the machine enters the same state $s_{ij} = s_{k\ell}$). Then $m_{ij} = m_{k\ell}$ (i.e., then the head moves in the same direction in both situations). If we have such a situation, we can draw the state diagram shown in Fig. 16 to represent our parity recognizing machine.

We have used the same conventions as in the case of finite-state automata, except that the symbol in each circle represents the head motion that results whenever this state (circle) is entered. An "H" in the diagram denotes a halting state of the machine. Quintuples of the form $(s_i, x_j, s_i, x_j, m_{ij})$ have been omitted from the diagram.

(35) In our second example we construct a Turing machine that computes the function $f(n) = 2n$ (for all $n \geq 1$). Two nonblank tape symbols "1" and "2" are used, and the machine has five internal states s_0, s_1, s_2, s_3, s_4. The table of quintuples is given in Table 9. The machine acts as follows. The tape consists of a finite sequence of 1's, and the tape head initially

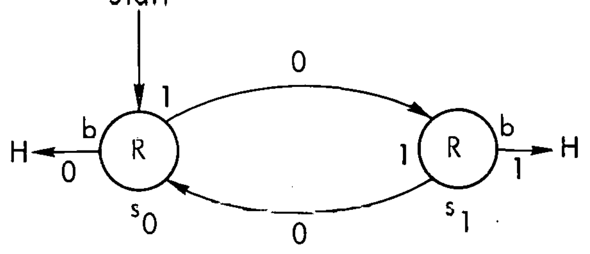


Fig. 16. State diagram for the parity-recognizing Turing machine in example 34.

scans the left-most 1 of this sequence. The starting state is s_0. The head then moves to the right until the first blank is encountered. The machine (which is still in state s_0) then writes a 2 after the end of the input sequence and moves into state s_1. It then moves to the far left of the sequence, erases the leftmost 1 (state is s_2), and changes to state s_3. In states s_3 and s_4 the tape head moves to the right end of the nonblank portion of the tape and writes two 1's. This procedure is repeated n times (where n is the number of 1's originally on the tape). For each of the original 1's that is erased, two 1's are added to the right end of the sequence. The machine halts in state s_4 with the head over the leftmost "1" of the answer, the answer consisting of a sequence of 2n 1's.

Table 9. State-transition table for example 35.

s_i	x_j	s_{ij}	x_{ij}	m_{ij}
s_0	b	s_1	2	L
s_0	1	s_0	1	R
s_0	2		not possible	
s_1	b	s_2	b	R
s_1	1	s_1	1	L
s_1	2	s_1	2	L
s_2	b		not possible	
s_2	1	s_3	b	R
s_2	2	s_4	b	R
s_3	b	s_4	1	R
s_3	1	s_3	1	R
s_3	2	s_3	2	R
s_4	b	s_1	1	L
s_4	1	H	-	-
s_4	2	H	-	-

Observe that the condition for drawing a state diagram is fulfilled. The diagram is shown in Fig. 17. Note that the original input sequence of n 1's has been completely erased by this machine. The reader may try to modify the machine so that the input is preserved.

(36) In this example we construct a machine that *multiplies* arbitrary numbers. This is our first demonstration that Turing machines are more powerful than finite-state machines. We assume that the set of nonblank tape symbols is $\Sigma = \{1,A,X,Y\}$, and that the tape is initially as shown in Fig. 18a (in this example the first number m is 2 and the second number n is 3).

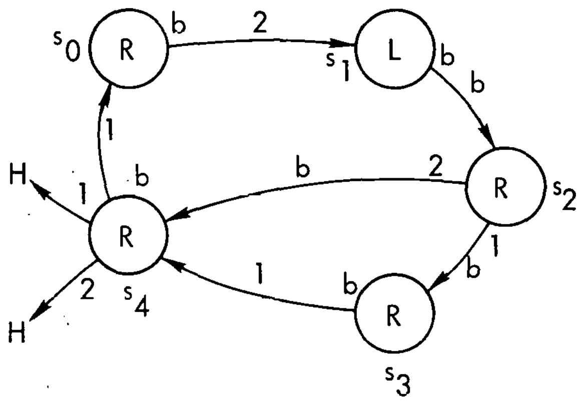
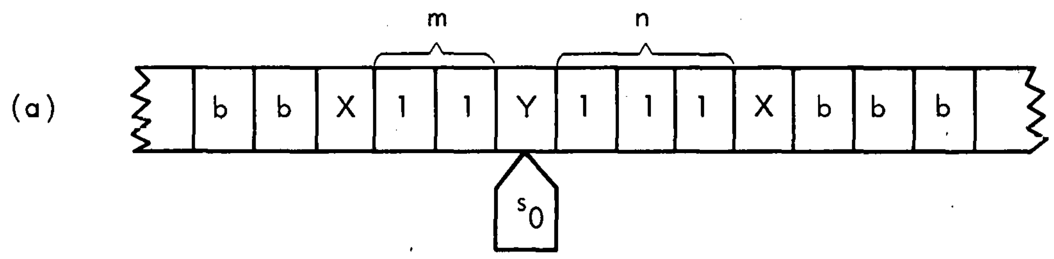


Fig. 17. State diagram for example 35.



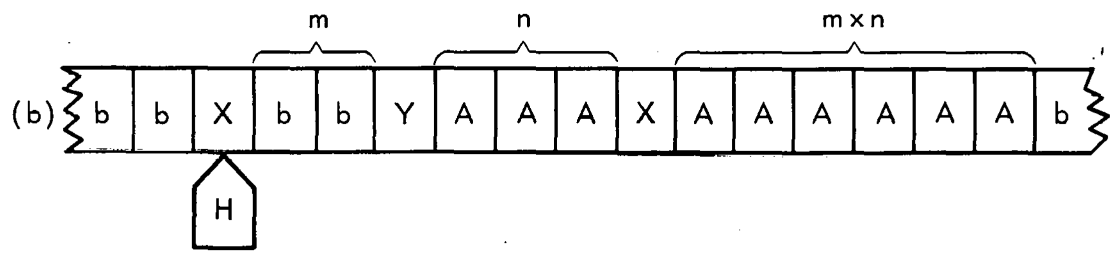


Fig. 18. (a) Initial status of the tape for the Turing machine of example 36, and (b) final status of the tape.

The symbols X and Y are end markers, Y separating the two numbers. The machine is initially in state s_0 with its head pointing to the square containing Y. There are four states s_0, s_1, s_2, s_3, with the following state transition table (Table 10):

Table 10. State-transition table for example 36.

s_i	x_j	s_{ij}	x_{ij}	m_{ij}
s_0	b	s_0	b	L
s_0	1	s_1	b	R
s_0	X	H	–	–
s_0	Y	s_0	Y	L
s_0	A	s_0	A	L
s_1	b	s_1	b	R
s_1	1	s_1	1	R
s_1	X	s_2	X	L
s_1	Y	s_1	Y	R
s_1	A	s_1	1	R
s_2	b	s_2	b	L
s_2	1	s_3	A	R
s_2	X	s_2	X	L
s_2	Y	s_0	Y	L
s_2	A	s_2	A	L
s_3	b	s_2	A	L
s_3	1	s_3	1	R
s_3	X	s_3	X	R
s_3	Y	s_3	Y	R
s_3	A	s_3	A	R

The corresponding state diagram is shown in Fig. 19.

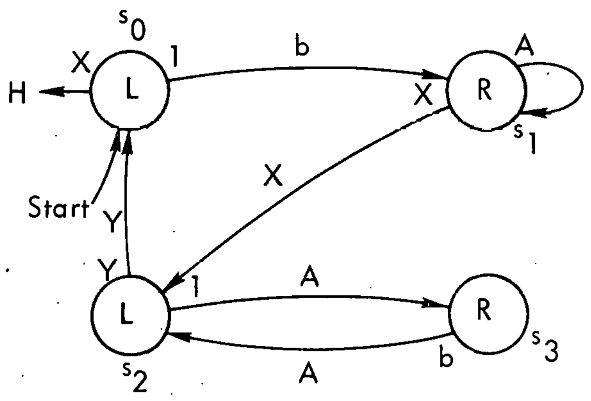


Fig. 19. State diagram for example 36.

Using the example n = 2, n = 3, the reader may trace the actions of the machine and determine the significance of each state. The end result is shown in Fig. 18b. The answer m x n is written as a sequence of m x n A's to the right of the right end marker X.

(37) The state diagram shown in Fig. 20 represents a Turing machine that converts a sequence of n 1's on the tape to the binary representation of the number n.

The tape looks initially (n = 10) as shown in Fig. 21a. At the end it has the form given in Fig. 21b. So the answer is BABA ≡ 1010 = 10 (base 2). In principle, the machine repeatedly divides by 2, writes the remainder (A for 0, B for 1) and repeats this process on the quotient until the quotient becomes zero. Again this is a computation that cannot be done by a finite-state machine (for arbitrary n) - a fact we are not going to prove.

Through these examples the reader should have gained sufficient insight into the structure of Turing machines to appreciate the following remarkable facts that have been proved mathematically:

(A) Any Turing machine can be replaced by one that uses only *two* tape symbols (say 0 and 1, 0 being also used for "blank"). The basic idea is to encode every nonblank symbol as a binary number. However, by reducing the number of symbols we increase the number of states.

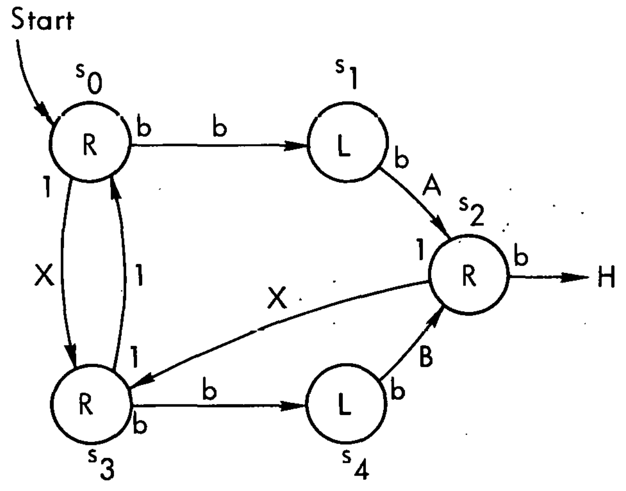
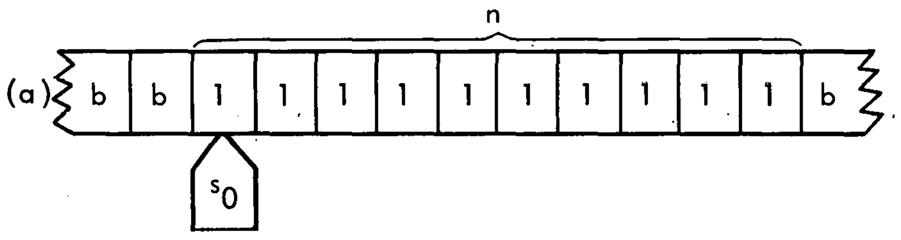


Fig. 20. State diagram for example 37.



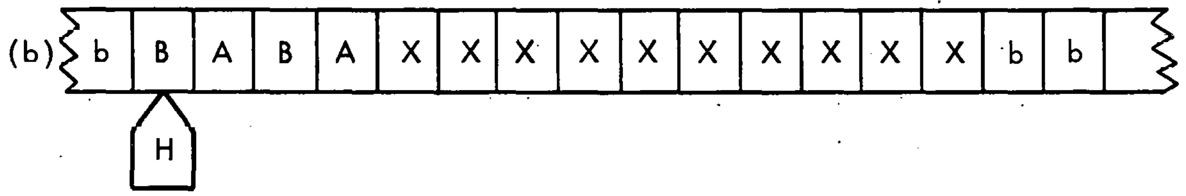


Fig. 21. (a) Initial status of the tape for example 37, and (b) final status of the tape.

(B) More remarkable is the fact that every Turing machine is equivalent to one that has only *two* internal states. The equivalence is achieved by greatly increasing the alphabet of the multistate machine. This fact was proved by Claude Shannon, the inventor of information theory.

(C) Turing machines that have tapes infinite in both directions have no advantage over machines with tapes that are infinite only in one direction.

(D) Turing machines that control several read/write heads that independently scan several tapes are no more powerful than single-tape machines.

(E) Recall the condition under which we could draw a state diagram: the head motion (L or R) depends only on the state which the machine enters. One can prove that every Turing machine is equivalent to such a "directed-state" machine, by adding suitable states (exercise!).

Universal Turing Machines

Let us suppose that T is a Turing machine that can compute a certain function f(x). That is, for each value of x (represented on the tape of T as a certain nonblank string S_x), T will eventually come to a halt with an appropriate "answer string" $S_{f(x)}$ remaining on the tape. Given a description of T (e.g., in terms of its quintuples) and of the initial string S_x on the tape, the reader could trace out the behavior of T with input S_x and find for himself what the corresponding value of f(x) is. In fact, this is presumably what the reader has done for the examples given above. He required an external storage medium (paper and pencil) and a precise understanding of how to interpret the description of the machine.

We intend to replace the reader by a "universal" Turing machine and give it the same necessary materials: an external storage medium (its tape and read/write head), a description on its tape of a machine T and of T's tape S_x, and the built-in capacity to interpret correctly the description of T's behavior.

In other words, there exists a universal Turing machine U, which is able to interpret and simulate the behavior of any Turing machine T. All U needs is a string of symbols d_T containing the encoded description of T, and it can then compute (by simulating T) all functions that T can compute. This means that there exists a Turing machine U, which is so sensitive that, by properly adjusting the input representation, it can compute *any* function that is possibly computable by a Turing machine (hence the name "universal").

By analogy we can think of a Turing machine T as an algorithm or computer program that solves a particular task (e.g., multiplies two numbers). A universal Turing machine then corresponds to a digital "stored program" computer that can "simulate" *any* program, as long as it is properly described and encoded (in machine language).

Asserting the existence of this universal machine is one of the highlights of Turing's original paper, and is also Turing's chief support that Turing machines can handle all effective procedures. To construct a universal Turing machine U, we shall first show how Turing machines can be used to store and retrieve information, by using the tape as a general-purpose file.

Suppose R_1, R_2, R_3, ..., R_k are a number of items (records), each R_i being given a name N_i, and suppose that (after encoding all items and names in, say, binary numbers) they have been arranged on a tape in pairs (N_i, R_i), separated by markers (say X's) (see Fig. 22).

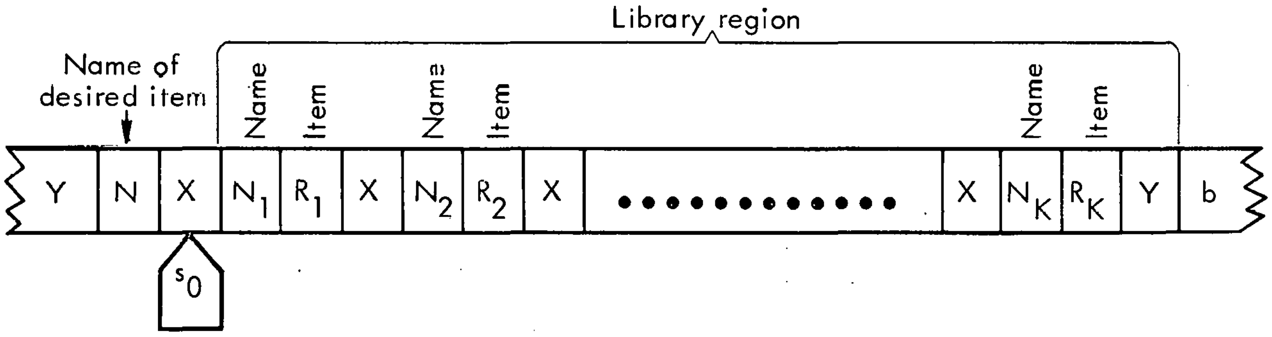


Fig. 22. Items and names arranged on tape for the universal Turing machine.

Now suppose we want to locate the item whose name is N (we assume that all names have the same length when encoded). Here the Y's are special end markers. The state diagram in Fig. 23 represents a machine that compares the given name N with each of the names N_1, N_2, ... in the "library." When it finds a match, it stops and returns to its starting position after having changed all 0's and 1's between the desired record R_i and the start position to A's and B's, respectively. For instance, this machine would cause the tape conversion shown in Fig. 24.

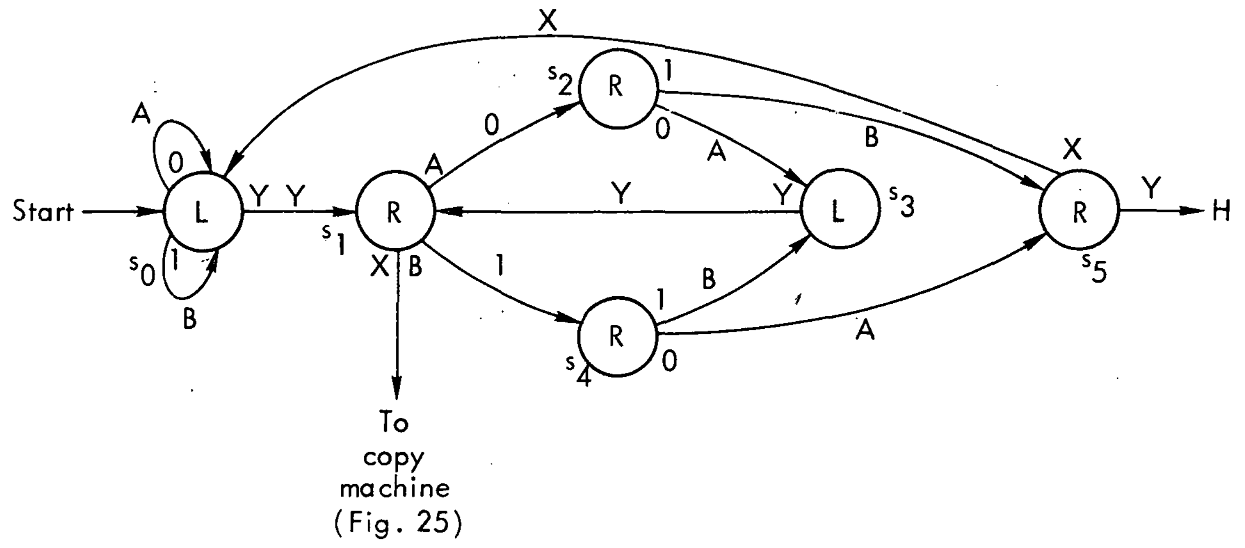


Fig. 23. State diagram for an item-locating machine.

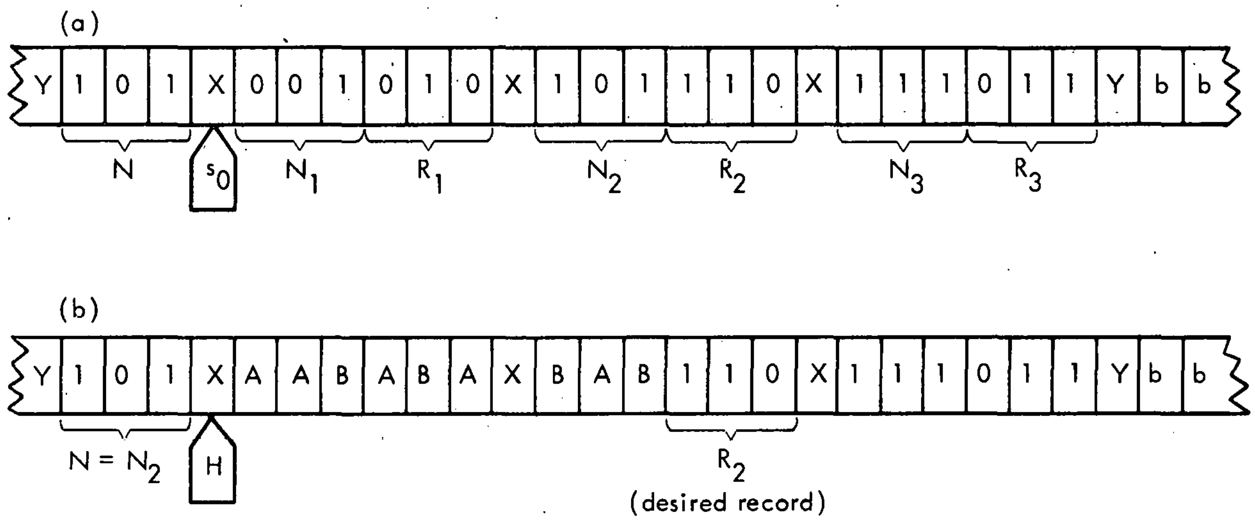

(a)
Y 1 0 1 X 0 0 1 0 1 0 X 1 0 1 1 1 0 X 1 1 1 0 1 1 Y b b
N
s_0
N_1
R_1
N_2
R_2
N_3
R_3
(b)
Y 1 0 1 X A A B A B A X B A B 1 1 0 X 1 1 1 0 1 1 Y b b
N = N_2
H
R_2
(desired record)

Fig. 24. Tape status (a) before and (b) after processing by the item-locating machine of Fig. 23.

Having located the particular record R_i, we might want to copy (move) it to another place on the tape. The state diagram shown in Fig. 25 represents a Turing machine that will copy the record R_i, just located, into the block that contains its name (we assume here that names and records have equal length). Thus, by applying the copy machine to the tape obtained from the item-locating machine above, we finally arrive at the tape depicted in Fig. 26.

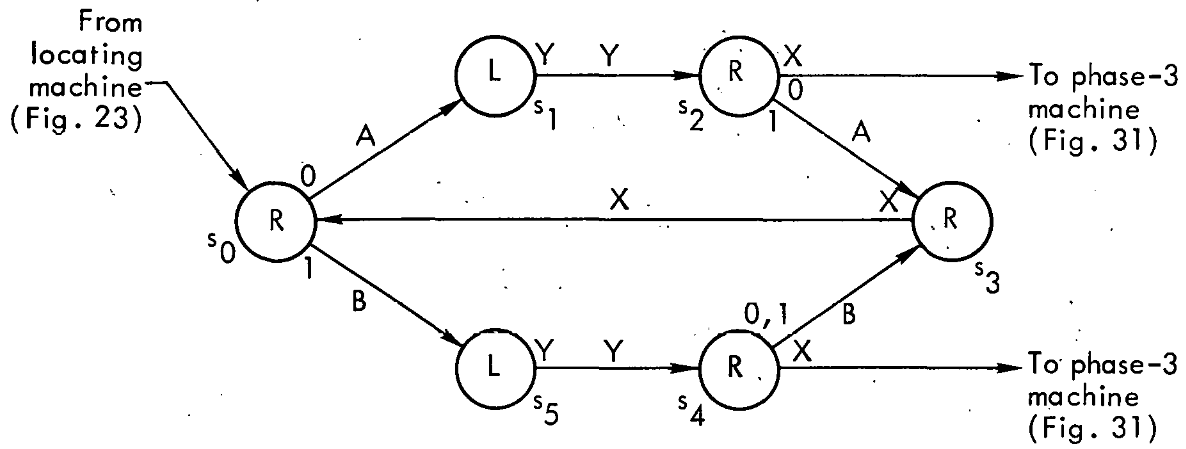

From locating machine (Fig. 23)
R
s_0
0
A
1
B
L
s_1
Y
Y
R
s_2
X
0
1
A
To phase-3 machine (Fig. 31)
X
X
R
s_3
L
s_5
Y
Y
R
s_4
0,1
B
X
To phase-3 machine (Fig. 31)

Fig. 25. State diagram of a Turing machine for copying a record into the block that contains its name.

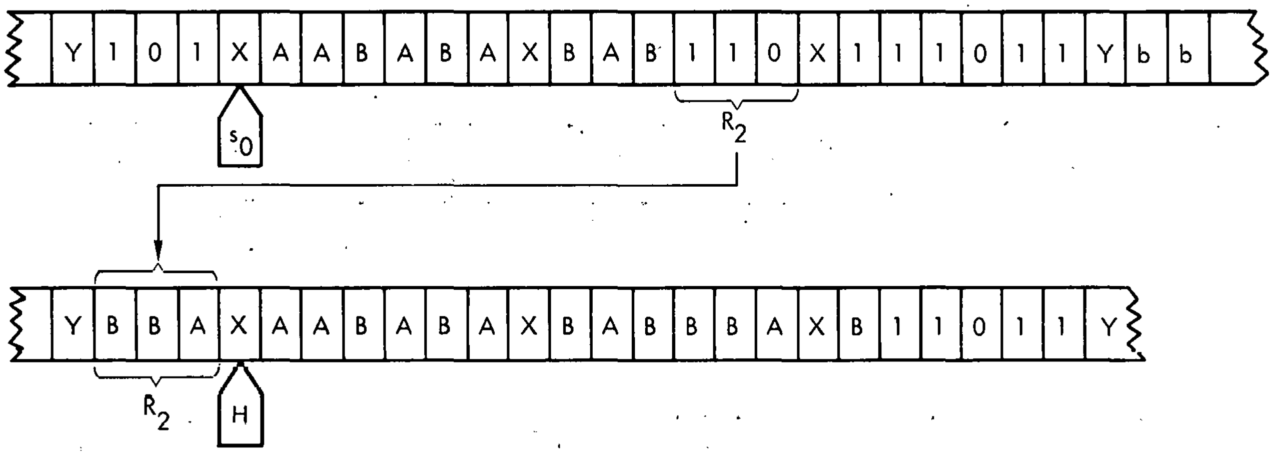

Y 1 0 1 X A A B A B A X B A B 1 1 0 X 1 1 1 0 1 1 Y b b
s_0
R_2
Y B B A X A A B A B A X B A B B B A X B 1 1 0 1 1 Y
R_2
H

Fig. 26. Tape status (a) before and (b) after processing by the copy machine shown in Fig. 25.

Having introduced file-locating and copying machines we can now begin to construct a universal Turing machine U. First, we describe the structure of U's input tape, which must contain (a) an encoded description of a Turing machine T (e.g., T's quintuples), and (b) a description of T's input tape. The description of the machine T in terms of quintuples (s_i, x_j, s_{ij}, x_{ij}, m_{ij}) appears on U's input tape as shown in Fig. 27.

Here the following conventions are used: if T has n states, and n as a binary number has b digits, then blocks of b binary digits are used to represent T's states on U's tape. In the above example, b = 2, so that this

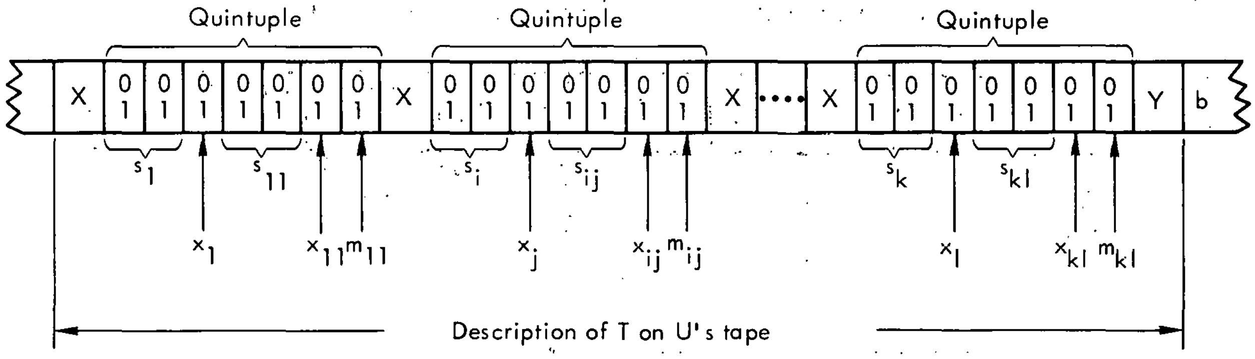


Fig. 27. Description of Turing machine T as it appears on U's tape.

is a description of a machine having at most 3 states. Furthermore, we assume that T's input-symbols are 0 and 1, so that x_j is represented as a single binary digit. Also, $m_{ij} = 0$, if T's tape head moves left, otherwise, $m_{ij} = 1$, so that m_{ij} is also represented by a single binary digit. $\boxed{\begin{matrix}0\\1\end{matrix}}$ means that either 0 or 1 may appear in this tape square. "Y" is an end marker which tells U where the description of T ends.

We assume next that the description of T's tape, its head position, and T's current state and symbol appear to the left of T's description on U's tape (Fig. 28). The part on U's tape immediately to the left of the left-most X is the machine condition of T, i.e., its current state and the symbol on T's tape currently being read by T. Further left is the description of T's tape, which is simply a "copy" of T's tape, except that in place of the symbol currently being read a special symbol M appears on U's tape, indicating the current position of T's tape head.

This concludes the description of U's input tape. Now we show how U operates. U's operation proceeds in a four-part cycle.

(1) Starting with the tape head over the left-most X as shown in Fig. 28, U sets the "item-locating" machine in operation. It searches to the right (i.e., in the machine description area) until it finds the first state-symbol pair in a quintuple that matches the one contained in the "machine condition" area (which initially will be (s_0, x_i), where s_0 is T's starting state, and x_i is the symbol initially under T's head). As we know, all 0's and 1's between the left-most X and the desired pair are changed to A's and B's, respectively.

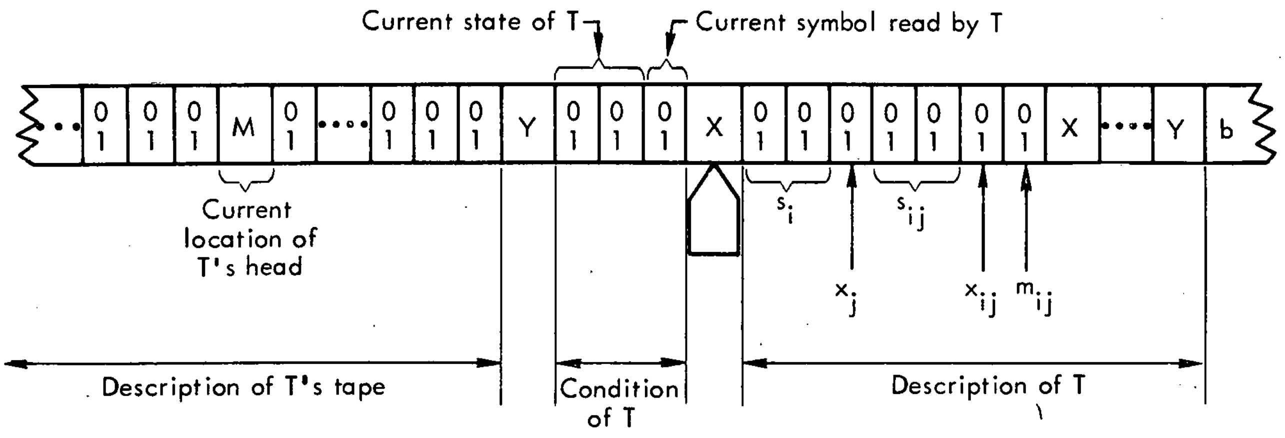


Fig. 28. U's tape containing the description, location of tape head, and current state and symbol of T.

After the matching pair has also been converted to A's and B's, U's tape head moves back to its starting position over the leftmost X. At that point U's tape appears as shown in Fig. 29.

(2) Now the "copy" machine is invoked. It moves U's head to the right until it encounters the first 0 or 1. It then copies the part that contains the new state (s_{0i}, or in general s_{ji}) and the new symbol (x_{0i}, or in general x_{ji}) into the "machine condition" portion of U's tape. The symbol representing the new head motion (m_{0i}, or in general m_{ji}) is "remembered" by U. After the copying process is finished, U's tape looks as shown in Fig. 30.

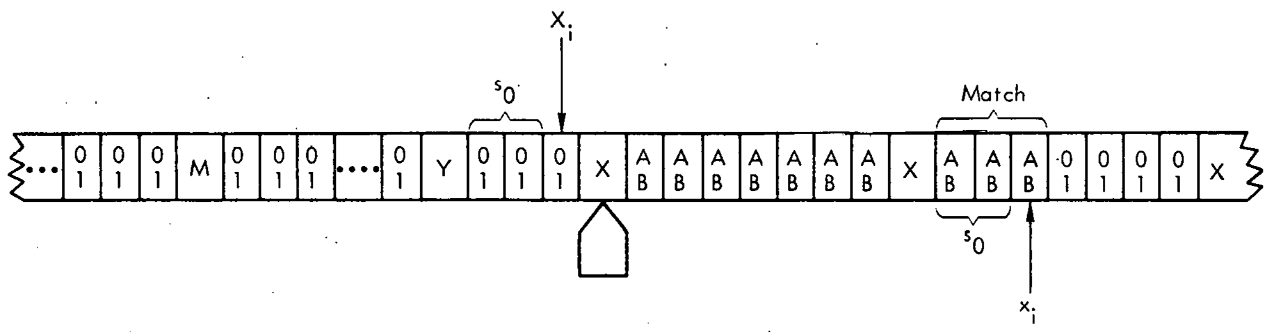


Fig. 29. U's tape after processing by the item-locating machine.

(3) Next, U's tape head moves to the left (i.e., along the portion that contains a description of T's tape) until it encounters the M. It erases the M and replaces it by the direction m_{ij} (A or B) (which had been "remembered" by U). Refer to the diagram of the copy machine (Fig. 25) and note the two exit arrows at the right. If m_{ij} was an A, the copy machine exits from state s_2, if m_{ij} was a B, it exits from s_4. This is the way U "remembered" m_{ij}! Then, U's tape head moves right again, changing all A's and B's to 0's and 1's (except the A or B which represents m_{ij} in M's old location). After that, it moves to the left of the left-most X, erasing the x_{ij} (and "remembering" it), and printing a special symbol S in its place. All this is done by the machine in Fig. 31. We'll call it the "phase-3" machine. The appearance of U's tape at the end of phase-3 is given in Fig. 32.

(4) In its fourth and final phase, U's head moves to the left until it encounters an A or B (this is the one representing the direction m_{ij} of T's head). U replaces the A or B with the value of x_{ij} (0 or 1) and shifts one square to the left or right, depending on whether the symbol encountered was A or B. The symbol in the new square is replaced by an M and "remembered." Then U shifts its head to the right until it reaches the symbol S. This it replaces by A or B, depending on whether the "remembered" symbol on T's tape was 0 or 1. The machine that does all this is represented by the diagram in Fig. 33. After completing phase 4, the tape appears as shown in Fig. 34.

At this point, the machine U has completed one cycle and is ready to start the next cycle, going through the same four procedures. During each cycle the machine U first locates the proper quintuple (s_i, x_j, s_{ij}, x_{ij}, m_{ij}) which contains the same pair (s_i, x_j) as is contained in the machine condition area. It then copies the new state and symbol (s_{ij}, x_{ij}) into the machine

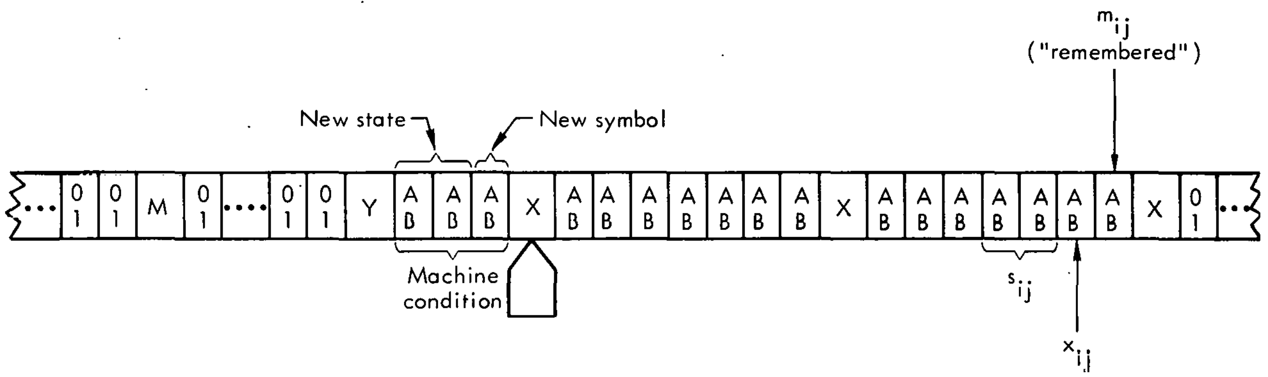

m_{ij}
("remembered")
New state
New symbol
... 0 1 0 1 M 0 1 .... 0 1 0 1 Y A B A B A B X A B A B A B A B A B A B A B X A B A B A B A B A B A B A B X 0 1 ...
Machine condition
s_{ij}
x_{ij}

Fig. 30. U's tape after processing by the copy machine.

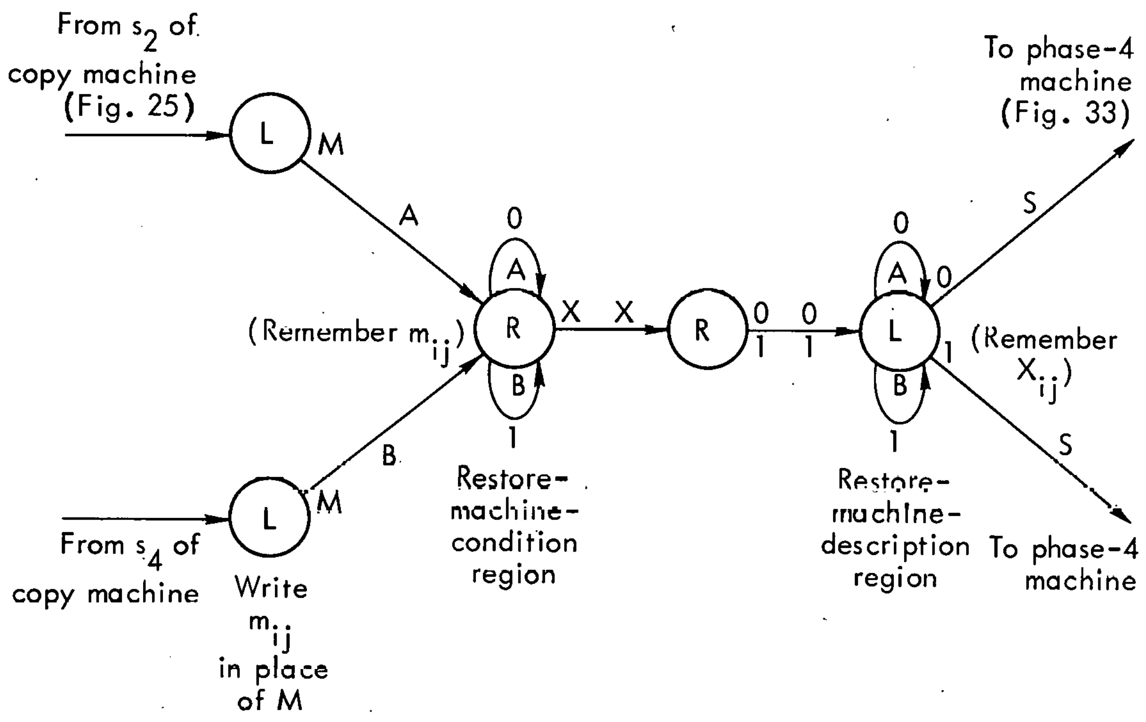

From s_2 of copy machine (Fig. 25)
L
M
A
0
A
(Remember m_{ij})
R
B
1
X
X
R
0
1
0
1
0
A
L
B
1
0
1
(Remember X_{ij})
S
To phase-4 machine (Fig. 33)
S
To phase-4 machine
B
L
M
From s_4 of copy machine
Write m_{ij} in place of M
Restore-machine-condition region
Restore-machine-description region

Fig. 31. Phase-3 machine.

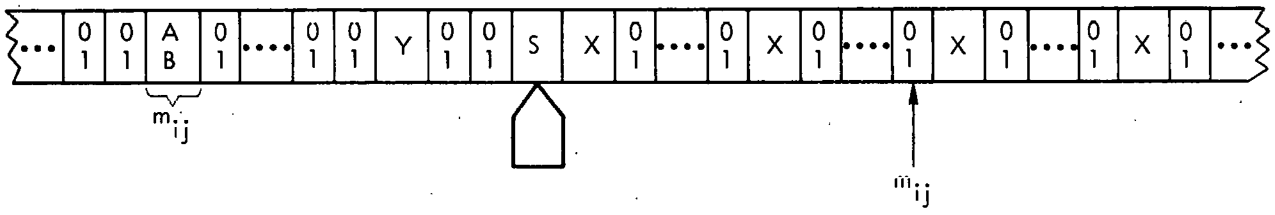


Fig. 32. U's tape after processing by the phase-3 machine.

condition area, moves the marker M, depending on m_{ij}, and reads the new symbol, replacing it by M. Thus, U carries out the instructions of exactly one quintuple of T during each cycle. This completes the construction of a universal Turing machine.

Some Unsolvability Results

Recall Turing's thesis that Turing machines can solve (compute) any problem (function) that is an effective procedure. In this chapter we are

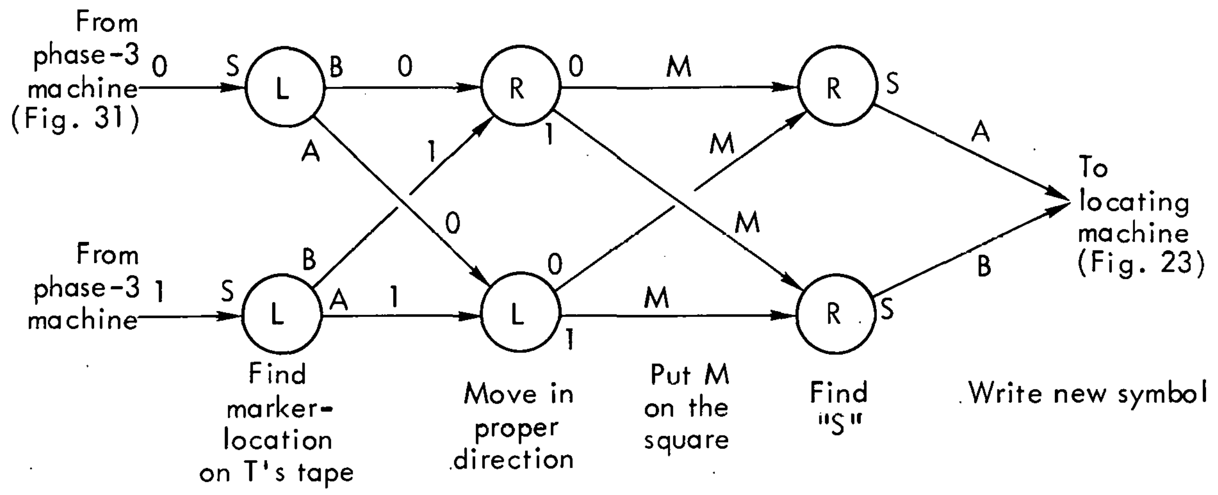


Fig. 33. Phase-4 machine.

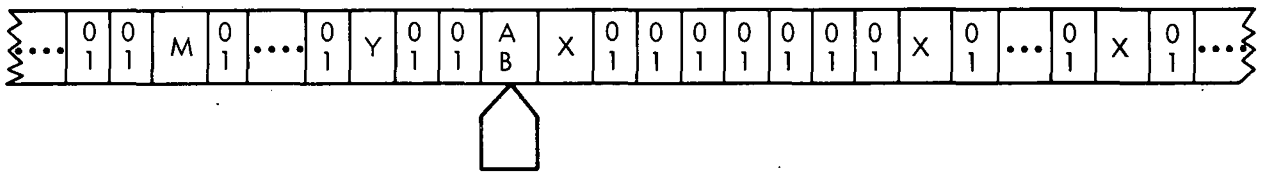


Fig. 34. U's tape after processing by the phase-4 machine.

going to look at some procedures that cannot be solved by any Turing machine, i.e., there are no algorithms to solve these procedures.

For our first question let us consider a Turing machine T with input tape τ. If T starts its operation it may be a very long time until it finally comes to a halt. In fact, T might get caught in an infinite loop and never halt. We are therefore entitled to ask the question below.

Halting Problem

Is there a Turing machine H that, given any Turing machine T and its input tape τ, is able to decide whether or not T comes to a halt?

It is obvious that the existence of such a machine H would be quite useful, but as it turns out no such machine can exist. To prove the unsolvability of the halting problem, we shall restrict the problem as follows: instead of considering *all* pairs (T,τ), we only select those pairs of the form (T, d_T), i.e., where the tape τ contains the description of T. Then we can ask the next question.

Restricted Halting Problem

Is there a Turing machine H' that, for any Turing machine T, can decide whether or not T halts if T is given its own description d_T on its input tape.

Clearly, if we can show that the restricted halting problem is unsolvable, we can imply the unsolvability of the more general halting problem.

Let us therefore assume that such a "decision" machine H' exists. Then H' must contain two internal states s_1, s_2, such that H' arrives in state s_1 when it decided that (T, d_T) halts, otherwise H' arrives in state s_2 (see Fig. 35).

Now let us denote by H" the machine H' modified by adding two states s', s" as shown in Fig. 36.

Obviously, H" halts if (T,d_T) does not halt. However, if (T, d_T) halts, then H" enters the infinite loop (s's"s's"....) and never halts.

Now, suppose that we give H" its *own* description $(d_{H''}, d_{H''})$. What will happen? If H", given its own description, should halt, then it must enter state S_1, and then the infinite loop (s's"s's"...), i.e., H" would never

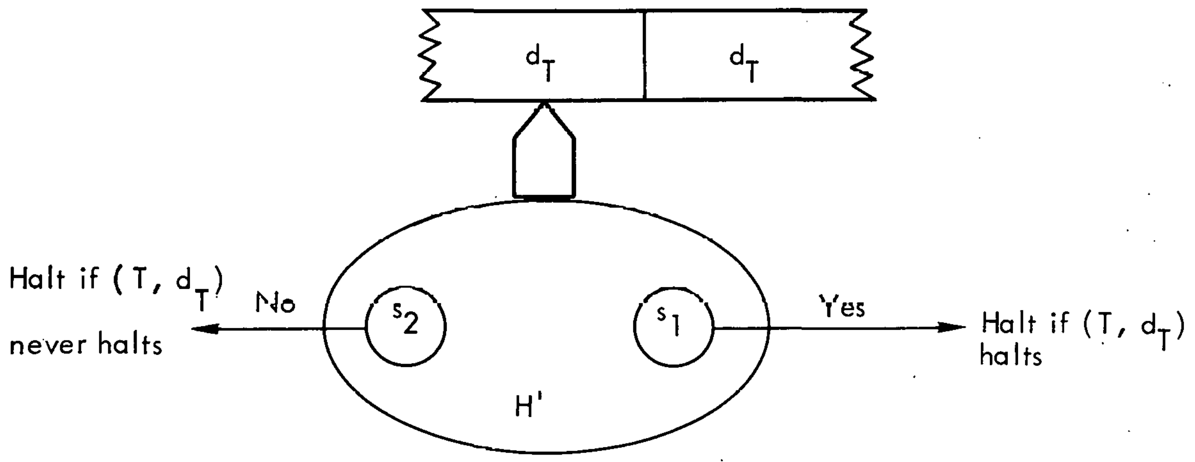


Fig. 35. Decision machine.

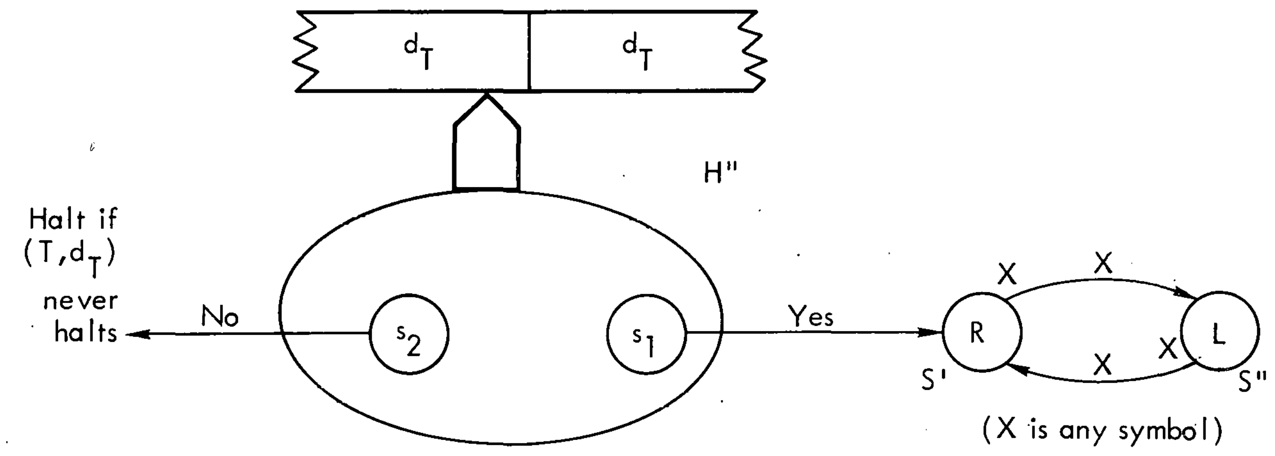


Fig. 36. Decision machine with two states added.

halt, which is a contradiction. Let us then suppose that H" never halts when given its own description. This implies that H" enters state s_2 and then halts - again we reached a contradiction! The only way out of this mess is to conclude that a machine like H" cannot exist, i.e., the restricted halting problem is unsolvable, and consequently the general halting problem is unsolvable.

Consequences of the Halting Problem

A consequence of the halting problem is the unsolvability of the *printing problem*, namely, "Does a Turing machine T ever print the symbol α when started with tape τ?"

Next we provide an example of a function that is not Turing-computable in the sense that there is no Turing machine T which, given a representation of a number x on its input tape, will come to a halt with the description of f(x) on the tape. The function f is defined as follows:

Let $C_m = \{T \ / \ T$ is a Turing machine with m states and tape symbols 0, 1$\}$.

Let τ_m be a tape containing m consecutive 1's.

Let us discard all $T \epsilon C_m$ which never halt when given tape τ_m, and let $\tilde{C}_m = \{T \epsilon C_m \ / \ (T,\tau_m)$ does halt$\}$ be the class of those $T \epsilon C_m$ that do halt when presented with input tape τ_m.

Define for each $T \epsilon \tilde{C}_m$: $\rho_m(T)$ = number of 1's on the tape after T halts. Since C_m and $\tilde{C}_m$ are finite sets, we can define the function f by

$$f(m) = \text{maximum } \{\rho_m (T) \;/\; T \epsilon \tilde{C}_m\}.$$

Assume now that there is a machine F which computes f (by Shannon's theorem we may assume that F uses also the two tape symbols {0,1}). Define a new machine F' as follows: F' proceeds just like F, but instead of halting when it has computed f(m), F' writes one more 1 on the tape and then halts. That is, F' computes the function f' given by

$$f'(m) = f(m) + 1$$

Let K be the number of states of F, and ask F to compute f(k). By its very definition, F' will compute

$$f'(k) = f(k) + 1$$

However, $F' \epsilon C_k$, and therefore

$$\rho_k(F') \leq f(k) \underset{\text{def}}{=} \max \{\rho_k(T')/T' \epsilon \tilde{C}_k\}.$$

But we just saw that $\rho_k(F') = f(k) + 1$ which is a contradiction! The only way to resolve this contradiction is by giving up the assumption that a machine F exists which computes f. This problem is known as the "busy-beaver problem," and we have just shown its unsolvability.

An interesting consequence of the unsolvability of the halting problem is the fact that *no* computer program can ever be devised that is capable of deciding whether an arbitrary program ever halts or gets caught in an infinite loop. This is an example of how an abstract theory like automata theory can yield results that are of interest to computer scientists and programmers.

Recursive Functions

Before discussing the concepts of recursive-function theory, I want to mention a result that concerns the class of languages accepted by Turing machines. First of all, let us assume that certain states of the Turing machine T are declared *final* (or *accepting*) states (just as for finite-state automata). Next, let Σ be the input alphabet to the machine (i.e., the input tape symbols), together with two special end markers α and β. We assume that a string $w \in \Sigma^*$ is presented on the input tape in the form $\alpha w \beta$, and that at the start the tape head is over the left end marker α. We now advance a definition.

Definition 25:

The Turing machine T *accepts* the string w Σ^* if and only if it halts in a final state, when w is presented on the input tape as described above. The string w is not accepted if either T does not halt or T halts in a nonfinal state. The set $L(T) = \{w \epsilon \Sigma^* / w \text{ accepted by } T\}$ is *the language accepted by T*.

We can now state a theorem analogous to Kleene's theorem (it was proved by Chomsky in 1959).

Theorem 12:

A language is accepted by some Turing machine if and only if it is a recursive (type 0) language.

The proof of this theorem is quite lengthy and in principle similar to the proof of Kleene's theorem, so we are not going to present it. It was first proved by Chomsky who used recursive function theory for the proof.

Next let us study in a little more detail the class of functions that can be computed by Turing machines. First, we want to give a precise definition of what we mean by saying "a function is Turing-computable." As we have seen, among the functions that can be computed by Turing machines are the function $s(x,y) = x + y$ and $P(x,y) = x \cdot y$. These are obviously functions of two variables x and y, so we are naturally lead to the consideration of functions of one or more arguments: $f(x_1, x_2, \ldots, x_n)$. Also, each argument x_i is assumed to be a nonnegative integer, and the function value $f(x_1, \ldots, x_n)$ is again a nonnegative integer. Furthermore, the function $f(x_1, \ldots, x_n)$ may not be defined for *all* arguments $(x_1, \ldots, x_n)$, e.g., the function $g(x,y) = \frac{x}{y}$ is *not* defined for $(x,0)$ and for those pairs (x,y) for which $\frac{x}{y}$ is not an integer, i.e., $x \not\equiv 0 (\text{mod } y)$.

Definition 26:

A function $f(x_1,x_2,\ldots,x_n)$ is called a *partial function* if it is *not* defined for *all* arguments $(x_1,\ldots,x_n)$. If $f(x_1,\ldots,x_n)$ is defined for *all* arguments $(x_1,\ldots,x_n)$, then f is called a *total function.*

Definition 27:

A partial function $f(x_1,\ldots,x_n)$ is called *Turing-computable in general* (Tcg), if there exists a Turing machine T that, when given a tape containing the arguments $(x_1,\ldots,x_n)$, will either halt with the value $f(x_1,\ldots,x_n)$ if $f(x_1,\ldots,x_n)$ is defined, or will not halt if $f(x_1,\ldots,x_n)$ is not defined. If f is a total function, and the Turing machine described above exists (i.e., it will halt for all $(x_1,\ldots,x_n)$ with answer $f(x_1,\ldots,x_n)$ on the tape), then f is called *Turing computable* (Tc). Thus, f being a total function and Tcg implies f is Tc. In either case (f Tcg or Tc) we write $f \sim T$ if T is a Turing machine that computes f.

Remark: In the following we shall only consider Turing machines that contain *two* tape symbols {0,1}. This does not restrict the generality, since we know that any Turing machine is equivalent to one using only two symbols.

Now that we have a precise definition of what it means for a function f to be Turing-computable, one can ask whether there is some *algebraic classification of the Turing-computable functions.* It turns out that there is indeed such an algebraic description of Turing-computable functions, in terms of so-called recursive functions. To clarify the term "recursive" let's look at the familiar Fibonacci sequence, and let us define the function $\phi(n)$ to be the nth term in the Fibonacci sequence 1,1,2,3,5,8,... (generally, each term is the sum of the two preceding terms, the first two terms being defined to be 1). Clearly, the function ϕ is defined as follows:

$$\phi(1) = \phi(2) = 1;$$

$$\phi(n) = \phi(n - 2) + \phi(n - 1) \text{ for } n \geq 3.$$

As we see, before we can find $\phi(n)$ for some value n, we have to know $\phi(n - 2)$ and $\phi(n - 1)$, i.e., the value $\phi(n)$ depends on the preceding values $\phi(n - 1)$ and $\phi(n - 2)$. In such a case one says that the function (here ϕ) is defined "recursively." The simplest class of recursive functions are the so-called *primitive recursive functions.*

Definition 28:(Primitive recursion):

(1) The *zero function* $Z(x) = 0$ is a primitive recursive function.

(2) The *successor function* $S(x) = x + 1$ is a primitive recursive function.

(3) For every i with $1 \leq i \leq n$, the i^{th} *projection function* $P_i^{(n)}(x_1,x_2,\ldots,x_n) = x_i$ is a primitive recursive function.

(4) If g and h are primitive recursive functions, then the function f defined by the *recursion scheme*

$$f(0, x_1,x_2,\ldots x_n) = g(x_1,x_2,\ldots,x_n),$$

$$f(y + 1,x_1,x_2,\ldots,x_n) = h(f(y,x_1,\ldots,x_n),y,x_1,\ldots,x_n)$$

is again a primitive recursive function.

(5) If h and $g_1,g_2,\ldots,g_k$ are primitive recursive functions, then their *composition* $f(x_1,\ldots,x_n) = h(g_1(x_1,\ldots,x_n), \ldots,g_k(x_1,\ldots,x_n))$ is a primitive recursive function.

Thus, any function that is constructed by a *finite* number of applications of rules (1) through (5) is *primitive recursive.*

Let us give a few examples of primitive recursive functions.

Examples:

(38) *Addition:* $f(x,y) = x + y$ is primitive recursive, in fact,
by (3): $f(0,y) = P_1^{(1)}(y)$,
by (4): $f(x + 1,y) = S(P_1^{(3)}(f(x,y),x,y))$.
(Since $f(x + 1,y) = (x + 1) + y = (x + y) + 1 = f(x,y) + 1$, one would be tempted to define $f(x + 1,y) = S(f(x,y))$, but this would not fit into our five rules above, so we have to use the more cumbersome definition (definition (4), above).

(39) *Multiplication:* $m(x,y) = xy$ is primitive recursive:
$m(0,y) = Z(y) = 0$,
$m(x + 1,y) = h(m(x,y),x,y)$, where
$h(x_1,x_2,x_3) = f(P_1^{(3)}(x_1,x_2,x_3),P_3^{(3)}(x_1,x_2,x_3))$,

f being the "sum" function in example (38). (Since $m(x + 1,y) = (x + 1)y = xy + y$, one would like to simply put $m(x + 1,y) = f(m(x,y),y)$, f being the addition function; but again this formula is not covered by any of the five rules for primitive recursion.)

(40) *Exponentiation:* $\exp(x,y) = y^x$ is a primitive recursive:

$$\exp(0,y) = S(Z(y)) \ (=1),$$
$$\exp(x + 1,y) = h(\exp(x,y),x,y), \text{ where}$$
$$h(x_1,x_2,x_3) = m(P_1^{(3)}(x_1,x_2,x_3),P_3^{(3)}(x_1,x_2,x_3)),$$ m being "multiplication" from example (39).

(41) *Factorial:* Recall the definition of the "factorial operator" !:

$$n! = \begin{cases} 1, \text{ if } n = 0 \\ 1\cdot 2\cdot 3\ldots n, \text{ if } n > 0, \text{ so } n! = n \cdot (n - 1)! \text{ for } n > 0. \end{cases}$$

The function $fc(x) = x!$ is primitive recursive:

$$fc(0) = S(Z(0)) = 1,$$
$$fc(x + 1) = h(fc(x),x), \text{ where}$$
$$h(x_1,x_2) = m(P_1^{(2)}(x_1,x_2), S(P_2^{(2)}(x_1,x_2)))$$

(42) *Zero-test:* The function

$$\overline{sg}(x) = \begin{cases} 1 \ , \text{ if } x = 0 \\ 0 \ , \text{ if } x > 0 \end{cases}$$

is primitive recursive:

$$\overline{sg}(0) = S(Z(0)) = 1,$$
$$\overline{sg}(x + 1) = Z(P_2^{(2)}(\overline{sg}(x),x)).$$

(43) *Parity:* The function

$$P(x) = \begin{cases} 0 \ , \text{ if } x \text{ is even} \\ 1 \ , \text{ if } x \text{ is odd} \end{cases}$$

is primitive recursive:

$$P(0) = Z(0) = 0,$$
$$P(x + 1) = \overline{sg}(P_1^{(2)}(P(x),x)).$$

(44) *Half:* The function

$H(x) = [x/2]$ is primitive recursive:

$$H(0) = Z(0),$$
$$H(x + 1) = h(H(x),x), \text{ where}$$
$$h(x_1,x_2) = f(P_1^{(2)}(x_1,x_2),P(P_2^{(2)}(x_1,x_2))),$$

f being "sum," P being "parity." (Here we have implicitly made use of the relationship $H(x + 1) = H(x) + P(x)$. Also note $x = 2\,H(x) + P(x)$ for *all* x).

(45) *Predecessor:* The function

$$pd(x) = \begin{cases} 0, \text{ if } x = 0 \\ x - 1, \text{ if } x > 0 \end{cases}$$

is primitive recursive:

$$pd(0) = Z(0) = 0,$$
$$pd(x + 1) = P_2^{(2)}(pd(x),x).$$

(46) *Subtraction:* The function

$$sub(x,y) = \begin{cases} y - x, \text{ if } y \geq x \\ 0, \text{ if } y < x \end{cases}$$

is primitive recursive:

$$sub(0,y) = P_1^{(1)}(y) = y$$
$$sub(x + 1,y) = h(sub(x,y),x,y),$$

where $h(x_1,x_2,x_3) = pd(P_1^{(3)}(x_1,x_2,x_3))$.

For our purposes we have to consider a wider class of recursive functions. This extension is accomplished by introducing a new type of function, and in some sense "adding" it to the primitive recursive functions.

Definition 29:

Let $g(y,x_1,x_2,\ldots,x_n)$ be a function of n + 1 arguments. Then we define the *minimization operator* μ_y by:

$\mu_y[g(y,x_1,\ldots,x_n) = 0]$ = the *smallest* value y for which $g(y,x_1,\ldots,x_n) = 0$ (here $(x_1,\ldots,x_n)$ are *fixed* parameters). Of course, there might not be such a smallest y, in which case $\mu_y[g(y,x_1,\ldots,x_n) = 0]$ is not defined. Also note that as the arguments $(x_1,\ldots,x_n)$ vary, the value of μ_y varies (if it exists at all), so that we can define the following *partial* function of n variables:

$$f(x_1,\ldots,x_n) = \begin{cases} \mu_y[g(y,x_1,\ldots,x_n) = 0] \text{ if such a y exists;} \\ \text{undefined otherwise.} \end{cases}$$

We call f also the *minimization operation on the function g*. It is assumed that the function g is a total function.

Example:

(47) Let $g(Y,X_1,X_2) = \begin{cases} 1, \text{ if } X_1^2 + X_2^2 < Y^2 \\ X_1^2 + X_2^2 \quad Y^2 \text{ otherwise.} \end{cases}$

For example, $g(2,3,4) = 3^2 + 4^2 - 2^2 = 21$

and $g(6,3,4) = 1$, since $3^2 + 4^2 < 6^2$.

Then

$$f(X_1,X_2) = \mu_y[g(Y,X_1,X_2) = 0] = \begin{cases} \sqrt{X_1^2 + X_2^2}, \text{ if } Y^2 = X_1^2 + X_2^2 \\ \text{undefined otherwise} \end{cases}$$

For example, $\mu_y[g(Y,3,4) = 0] = 5$

$\mu_y[g(Y,3,5) = 0]$ is undefined.

Now we can extend the class of primitive recursive functions to the class of partial recursive functions.

Definition 30:

The class of *partial recursive functions* is defined by six rules, the five rules from definition 27 with the word "primitive" replaced by "partial," and the following rule:

(6) If $g(y,x_1,\ldots,x_n)$ is a partial recursive function that is total (i.e., defined for *all* arguments), then the *minimization operation* μ_y applied to g is a partial recursive function.

It can be shown that the minimization operation μ_y is *not* primitive recursive, and consequently, the class of partial recursive functions is actually a wider class than the class of primitive recursive functions. In contrast to the primitive recursive functions, which can be shown to be total functions, the partial recursive functions may not be defined for all arguments (since μ_y is sometimes undefined). Hence they are partial functions in our initial sense of the word. The subset of all those partial recursive functions that are actually total functions is quite important and is called the set of *recursive* or *general-recursive functions*. The class of recursive functions lies between that of the primitive recursive and the partial recursive functions and coincides with neither class:

$$\{\text{Primitive recursive } f_{ns}\} \underset{\substack{\uparrow \\ \text{proper inclusion}}}{\subset} \{\text{recursive } f_{ns}\} \underset{\substack{\uparrow \\ \text{proper inclusion}}}{\subset} \{\text{partial recursive } f_{ns}\}.$$

We can now state the result that answers our question concerning an algebraic classification of Turing-computable functions.

Theorem 13:

(A) A function $f(x_1,\ldots,x_n)$ is Turing-computable (Tc) if and only if it is recursive.

(B) A function $f(x_1,\ldots,x_n)$ is Turing-computable in general (Tcg) if and only if it is a partial recursive function.

Clearly, it is sufficient to prove part (B) of the theorem, since part (A) follows more or less by definition of Tc and recursiveness as compared to Tcg and partial recursiveness.

Proof of theorem:

The proof of part (B) breaks into two parts: (i) If f is a partial recursive function, then f is Tcg, and (ii) If f is Tcg, then f is a partial recursive function.

Proof of (i):

A function f is partial recursive if and only if it is either the zero function, successor function, or projection function, or if it is formed by finitely many applications of (a) the recursive scheme, (b) composition, and (c) the minimization operation. Thus, one must show that there are Turing machines that compute the first three functions, the composition, the "recursion scheme," and the minimization operation. The proof is tedious and is therefore omitted. The proof of part (ii) is more interesting.

Proof of (ii):

Recall that we start with a Tcg function f, and that we want to show that f is partial recursive. So let $f \sim T$, where T is a Turing machine with only two input symbols $\{0,1\}$.

Suppose we take a look at T's tape at a certain time t, when T is in state s_t (Fig. 37). Here x_t is the tape symbol currently under the tape

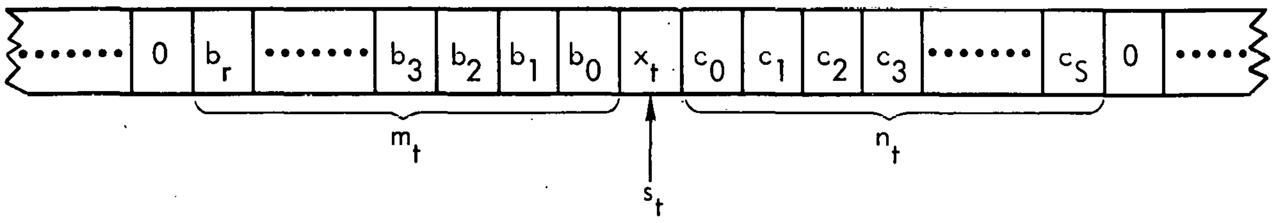


Fig. 37. T's tape at time t.

head, while m_t is the nonblank portion of the tape to the left of x_t and n_t is the nonblank portion to the right of x_t. Since both portions have finite length, we can also define

$$m_t = \sum_{i=0}^{r} b_i 2^i \quad \text{and} \quad n_t = \sum_{i=0}^{s} c_i 2^i .$$

Suppose that the quintuple beginning with state s_t and symbol x_t is $(s_t, x_t, s_{t+1}, X, 0)$, where we use 0 for motion to the right and 1 for motion to the left. Then, at time t + 1 the tape would appear as in Fig. 38. Thus,

$$m_{t+1} = X + \sum_{i=0}^{r} b_i 2^{i+1} = X + 2m_t \text{ and}$$

$$n_{t+1} = \sum_{i=1}^{s} c_i 2^{i-1} = H(n_t) = [n_t/2] ,$$

$$x_{t+1} = P(n_t) = c_0 .$$

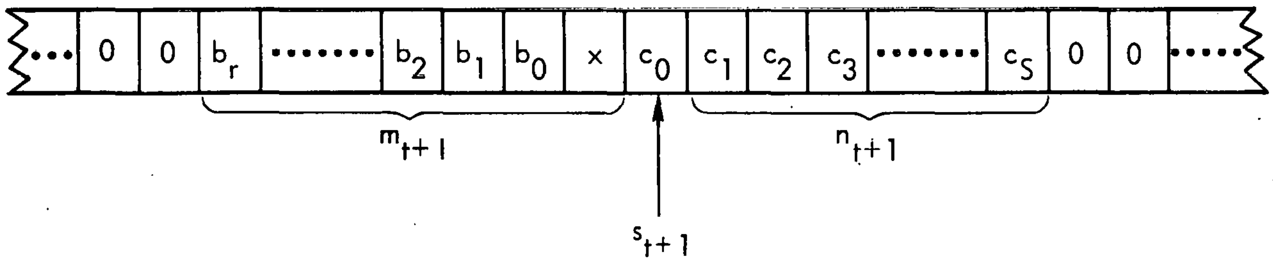


Fig. 38. T's tape at time t + 1, after tape head moved to the right.

If the quintuple had been $(s_t, x_t, s_{t+1}, X, 1)$, we would have

$$m_{t+1} = H(m_t) = [m_t/2].$$

$$n_{t+1} = X + 2n_t,$$

$$x_{t+1} = P(m_t) = b_0.$$

Now let us assign integers to the states of T by assigning the number i to state s_i (using some fixed ordering of T's states). Then, instead of using quintuples, we can associate three functions with T:

$Q(s_i, x_j) = s_{ij}$ (next-state function)

$S(s_i, x_j) = x_{ij}$ (next-symbol function)

$M(s_i, x_j) = m_{ij}$ (next-move function)

All three functions are number-theoretic functions using numbers for the states, symbols, and moves (0 for R, 1 for L). Now it is a simple exercise to show that any number-theoretic function f that must assume preassigned values at a finite number of arguments, and can be defined arbitrarily elsewhere, is a primitive recursive function. In particular, the three functions Q, S, and M are primitive recursive functions.

Let us now consider the four functions:

$q_T(t,n)$ (state at time t, if input at time 0 is n),

$m_T(t,n)$ (portion of tape to the left of the tape head at time t),

$n_T(t,n)$ (portion of tape to the right of the tape head at time t),

$s_T(t,n)$ (symbol being read at time t).

Initially, at t = 0 we have:

$q_T(0,n) = 0$ (T starts at state $s_0 \simeq 0$);

$m_T(0,n) = 0$ (tape head starts at left);

$n_T(0,n) = H(n) = [n/2]$

$s_T(0,n) = P(n)$ (least significant digit of n).

Then, at time t + 1 we have:

$$\begin{aligned}
q_T(t+1,n) &= Q(q_T(t,n),\ s_T(t,n)),\\
m_T(t+1,n) &= (2\ m_T(t,n) + S(q_T(t,n),\ s_T(t,n)))\\
&\quad \cdot (1 - M(q_T(t,n),\ s_T(t,n)))\\
&\quad + H(m_T(t,n)) \cdot M(q_T(t,n),\ s_T(t,n))\\
n_T(t+1,n) &= (2\ n_T(t,n) + S(q_T(t,n),\ s_T(t,n)))\\
&\quad \cdot M(q_T(t,n),\ s_T(t,n))\\
&\quad + H(n_T(t,n)) \cdot (1 - M(q_T(t,n),\ s_T(t,n)))\\
s_T(t+1,n) &= P(n_T(t,n)) \cdot (1 - M(q_T(t,n),\ s_T(t,n)))\\
&\quad + P(m_T(t,n)) \cdot M(q_T(t,n),\ s_T(t,n))
\end{aligned}$$

Since all the functions on the right-hand side are primitive recursive, the functions q_T, m_T, n_T, and s_T are primitive recursive. Now suppose T computes the given function f, i.e., T halts with f(n) on the tape, where n is the initial input string at t = 0. Thus, assuming the tape head is to the *right* of the "answer" f(n) when T halts, we have:

$$f(n) = m_T(t^*,n) \text{ when } q_T(t^*,n) = H,$$

where t* is the smallest t such that

$$q_T(t,n) = H.$$

Hence: $f(n) = m_T(\mu_t[q_T(t,n) = H],\ n)$,

using the minimization operator μ_t. Note that q_T is a total function since it is primitive recursive. Hence μ_t is defined. This proves that f is a partial recursive function, which is what we had to show.

Before completing our discussion of Turing machines, we might mention that one can also define nondeterministic Turing machines by simply allowing a finite number of possible "moves" for each given pair (state, input symbol). That is, given (s_i,x_j) there are finitely many quintuples $(s_i,x_j,s_{ij},x_{ij},m_{ij})$ that start with (s_i,x_j). A string on the input tape is "accepted" by a nondeterministic Turing machine if there is at least one possible sequence of moves that leads to a halt in an accepting state. The set of these words is then the language accepted by the machine.

2

Theorem 14:

Nondeterministic Turing machines are no more powerful than deterministic Turing machines. In other words, both types of machines accept the same class of languages (recursive). This is similar to proposition 4 for finite-state automata.

Exercises

(16) Construct a Turing machine with no more than four states, that will add two integers.

(17) A Turing machine initially starts out on a tape of the form shown in Fig. 39a. Its state diagram is shown in Fig. 39b. Analyze this machine.

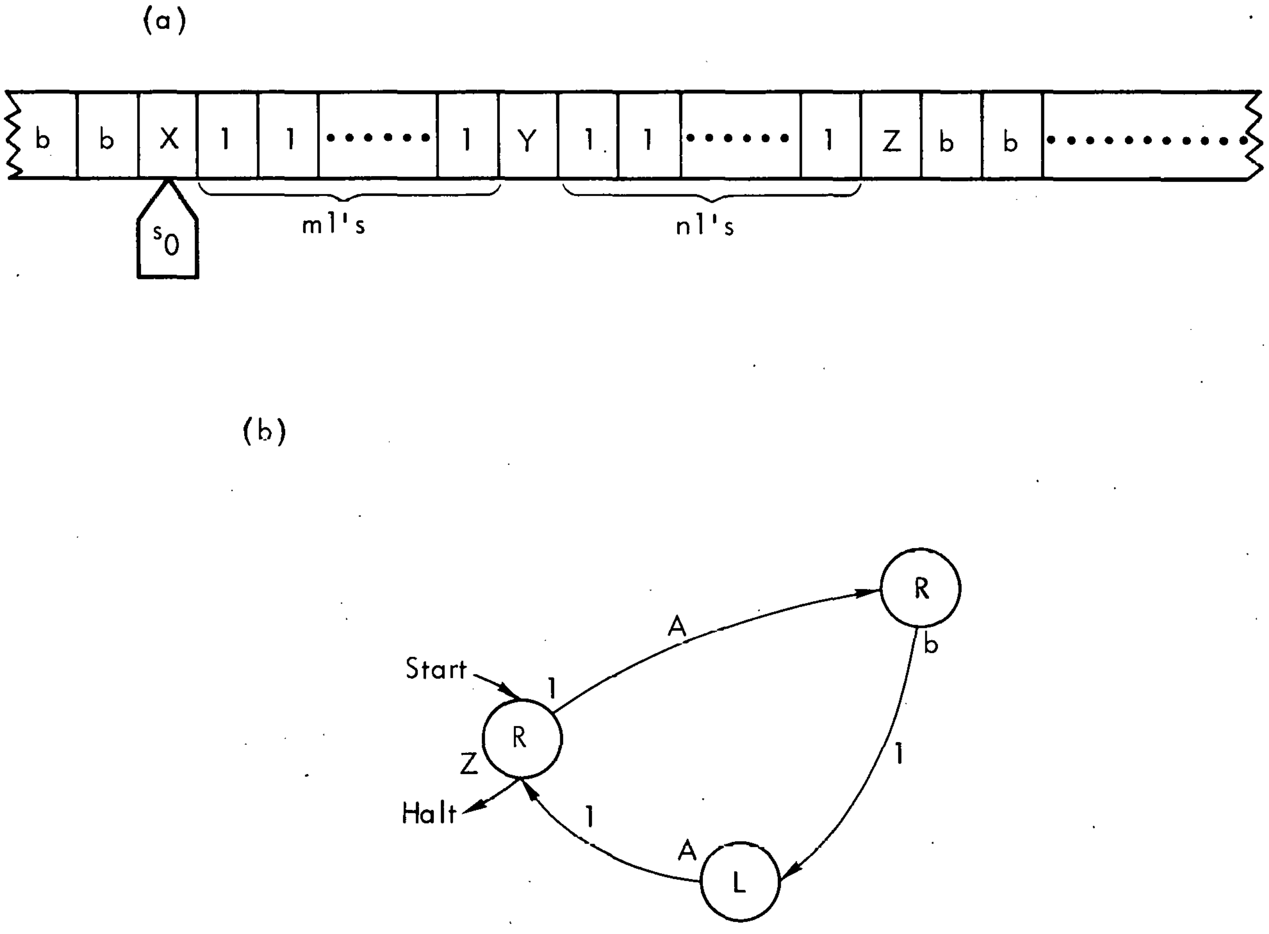


Fig. 39. (a) Tape and (b) state diagram for exercise 2.

(18) A Turing machine is called a *directed state machine* if, for any 2 quintuples $(s_i, x_j, s_{ij}, x_{ij}, m_{ij})$ and $(s_k, x_l, s_{kl}, x_{kl}, m_{kl})$ with $s_{ij} = s_{kl}$, we have $m_{ij} = m_{kl}$ (entering the same state results in the same head motion). Given any Turing machine, construct an equivalent directed-state machine.

(19) Prove that a Turing machine whose only head motion is to the right can be replaced by a finite-state automaton.

PUSHDOWN AND LINEAR-BOUNDED AUTOMATA

Pushdown Automata

A *pushdown automaton* (PDA) is essentially a finite-state control unit that controls an input tape of finite length and a "pushdown store" or "pushdown stack." The pushdown store is a first-in-last-out type stack, i.e., a symbol (out of some finite alphabet Γ) is entered at the *top* of the stack, thereby pushing all symbols that are already in the stack down by one unit. So, the symbol that had previously been at the top is now at the second place from the top, the symbol previously second from the top becomes third, etc. The symbol at the top of the stack may also be removed in which case all other symbols move up by one unit.

A familiar example of a pushdown store is the stack of plates resting on a spring; this device is frequently seen in cafeterias. Putting a new plate on top of the stack moves all plates in the stack down by one unit; removing the top plate raises the stack by one unit.

We can think of the stack as a tape that is infinitely long in one direction (say to the right), so the *bottom* of the pushdown stack is the left-most square of that tape and the *top* is the square currently scanned by the tape head.

Initially, the tape head for the input tape is over the left-most symbol on that tape. Also, in contrast to Turing machines, the tape head only moves to the right until it reaches the last symbol on the tape. The input symbols are elements of some finite alphabet Σ.

Let a particular pushdown symbol Z_0 ($Z_0 \in \Gamma$ = pushdown alphabet) initially be in the left-most cell of the pushdown tape, and the tape head initially scan that left-most tape square. The initial configuration of a pushdown automaton is shown in Fig. 40. There exist two types of moves:

(i) Depending on the current input symbol, the top symbol in the pushdown stack, and the state of the control unit, the machine moves into a new state, the read head for the input tape moves one square to the right, and the R/W head for the pushdown tape either moves one square to the right and writes a new pushdown symbol into that square (thereby "lowering" the stack, with the new symbol now the "top" symbol) or it erases the "top" symbol and moves one square to the left (thereby "raising" the stack). If the last symbol in the stack is removed the machine halts. Also, the pushdown tape might not be modified at all (denote this by -).

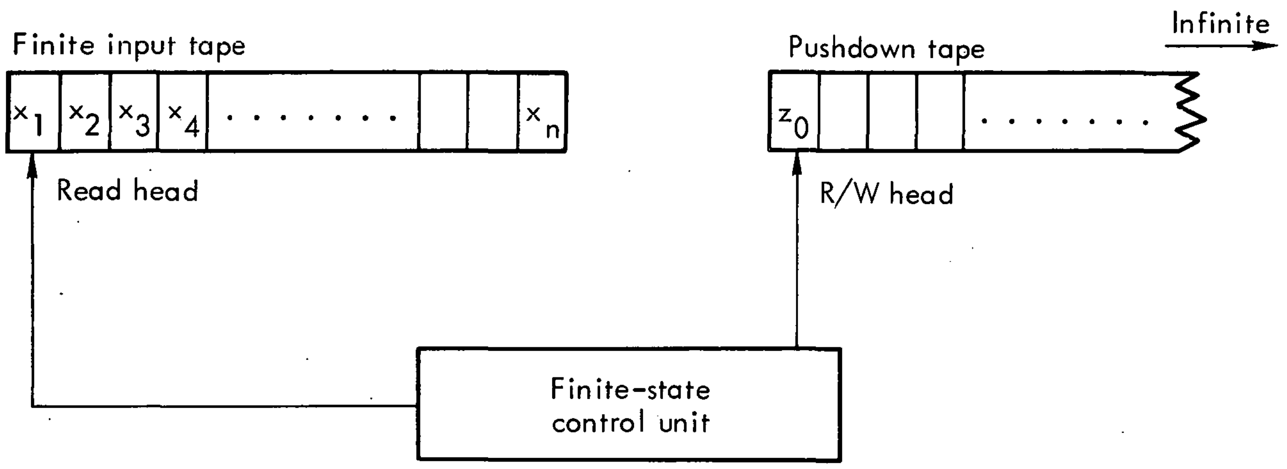


Fig. 40. Initial configuration of a pushdown automaton.

(ii) In the second type of move (called an "ε move"), the input symbol is ignored and the input tape head is *not* advanced. This allows the machine to manipulate the pushdown stack without reading input symbols.

Let us fix some notation.

As usual, let K be a finite set of states;

let Σ be a finite input alphabet;

let Γ be a finite pushdown alphabet;

let $F \subset K$ be a subset of final states;

let $s_o \in K$ be the starting state;

let $Z_o \in \Gamma$ be the start pushdown symbol;

let $f : K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to K \times (\Gamma \cup \{\varepsilon,-\})$ be the *"next-state" function.*

Thus, if $s_i \in K$, $x_j \in \Sigma \cup \{\varepsilon\}$, $p_k \in \Gamma$,

then

$$f(s_i, x_j, p_k) = (s_{ijk}, p_{ijk}),$$

where $s_{ijk} \in K$ is the next state, and

$p_{ijk} \in \Gamma \cup \{\varepsilon,-\}$ is the new pushdown-symbol at the "top" of the stack.

Note: If $x_j = \varepsilon$, then this signals an ε-move (no input is read!); if $p_{ijk} \in \Gamma$, then the pushdown tape head moves *right* and adds p_{ijk} to the top of the stack; if $p_{ijk} = \varepsilon$, then the top symbol is erased and the pushdown tape head moves to the *left*.

A "move" of the pushdown machine can therefore be described by a quintuple $(s_i, x_j, p_k, s_{ijk}, p_{ijk})$. Note that this is the *deterministic* prototype of a PDA. Now, an input string $w \in \Sigma^*$ can be accepted in two ways:

(i) If the machine, after reading w, moves into a final state $s \in F$, then the string is said to be *accepted by final state*.

(ii) If the machine, after reading w, ends up with an empty pushdown store, then the string is said to be *accepted by empty store*.

As the following proposition shows, both types of acceptance are equivalent.

Proposition 5:

Let P_1 be a pushdown automaton, and let $L_f(P_1)$ be the language accepted by P_1 by final state. Then there exists another PDA P_2 such that $L_f(P_1) = L_\varepsilon(P_2)$ is the language accepted by P_2 by empty store. Similarly, given P_2, one can find a PDA P_1, such that $L_\varepsilon(P_2) = L_f(P_1)$.

In view of this result, we can define *L(DPDA) to be the class of languages accepted by deterministic pushdown automata*, without worrying about the type of acceptance (final state or empty store).

Instead of deterministic PDA's we can also consider *non-deterministic PDA's (NDPDA)*, where the next-state function is multi-valued, i.e., $f : K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow P(K \times (\Sigma \cup \{\varepsilon, -\}))$. So, given a triple (s_i, x_j, p_k), there are several possible choices for the next state and the new pushdown symbol. Again, an input string w is accepted by final state if at least one choice eventually leads to a final state, and a string is accepted by empty store if at least one choice eventually leads to an empty pushdown stack. As before, both types of acceptance are equivalent, and we can uniquely define *L(NDPDA)* as the *class of languages that are accepted by NDPDA's.*

For finite-state automata and for Turing machines, the deterministic and the nondeterministic prototypes turned out to be equivalent with respect to the class of accepted languages. This is not true for PDA's. In fact the language $L = \{w\,w^R \,/\, w \in \Sigma^*\}$ is accepted by an NDPDA but not by a DPDA (see example 49), so PDA's are our first example of machines where the nondeterministic model is more powerful than the deterministic one.

Obviously, by ignoring the pushdown store, a DPDA is nothing more than a finite-state machine. Consequently, every regular language can be accepted by some DPDA. The following fundamental theorem was proved by N. Chomsky in 1962.

Theorem 15:

The class of languages accepted by NDPDA's coincides with the class of context-free languages:

$L(\text{NDPDA}) = \{\text{type } 2\}.$

We therefore have the following chain of (proper) set-inclusions:

$\{\text{type } 3\} \subset L(\text{DPDA}) \subset L(\text{NDPDA}) = \{\text{type } 2\}.$

In fact, the language $\{1^n c 1^n \,/\, n > 0\}$ over the alphabet $\Sigma = \{1, c\}$ is in $L(\text{DPDA})$ but is not regular, proving that $\{\text{type } 3\} \neq L(\text{DPDA})$. Languages accepted by DPDA's are of importance and are called *deterministic*.

Example:

(48) Here we describe a DPDA that will accept the language $L = \{wcw^R \,/\, w \in \{0, 1\}^*\}$. We use "acceptance by empty store." Thus, let $\Sigma = \{0, 1, c\}$, $\Gamma = \{X, Y, Z\}$, $K = \{s_0, s_1\}$, s_0 = start state, and X = initial pushdown symbol.

Table 11 is the state transition table.

"Illegal" configurations (like s_1, 0, X, ...)) have been omitted from the table - they certainly never lead to a final state or an empty pushdown store.

The operation of this machine can be described as follows. As long as we have not reached a "c", every "0" input causes a "Y" to be added to the top of the pushdown stack, while every "1" causes a "Z" to be added. Also, the state does not change. The moment a "c" is encountered, the state changes from s_0 to s_1, but the pushdown stack is not modified. From then on, for every "0" input the machine expects a "Y" on top of the stack and removes it; similarly for a "1" input the machine expects a "Z" on top of the stack, and promptly removes it. The moment an "X" is encountered on top of the stack,

Table 11. State-transition table for example 48.

Current state	Current input	Current top pushdown symbol	Next state	Next top pushdown symbol
s_0	0	X	s_0	Y
s_0	0	Y	s_0	Y
s_0	0	Z	s_0	Y
s_0	1	X	s_0	Z
s_0	1	Y	s_0	Z
s_0	1	Z	s_0	Z
s_0	c	X	s_1	-
s_0	c	Y	s_1	-
s_0	c	Z	s_1	-
s_1	0	Y	s_1	ε
s_1	1	Z	s_1	ε
s_1	ε	X	s_1	ε

it is removed (we have reached the last input symbol). The stack is empty, so the string wcw^R has been accepted. Figure 41 shows the varying contents of the pushdown store as the input sequence 011c110 is accepted. A sequence (such as wcw^R) which reads the same forwards or backwards is called a *palindrome*. We have shown that the language $L = \{wcw^R \ / \ w \in \{0,1\}^*\}$ is in L(DPDA), so, by our earlier definition, L is a deterministic language.

Example:

(49) We now exhibit a language that, in contrast to the previous example, cannot be recognized by a deterministic PDA, but only by an NDPDA. The NDPDA that we shall construct will recognize the language $L = \{ww^R \ /$ $w \in \{0,1\}^*\}$. Again we'll use "acceptance by empty store."

Let $\Sigma = \{0,1\}$, $\Gamma = \{X,Y,Z\}$, and $K = \{s_0,s_1\}$, where s_0 = start state and X = initial pushdown symbol. Table 12 is the state-transition table.

Input:		0		1		1		c		1		1		0		
Top of pushdown store:	X		Y		Z		Z		Z		Z		Y		X	ϵ
			X		Y		Z		Z		Y		X			↑
					X		Y		Y		X					
							X		X							

Pushdown store emptied

Fig. 41. Contents of the pushdown store as the sequence 011c110 is accepted.

Note that this machine acts nondeterministically only in two situations:

(a) in state s_0, after reading 2 consecutive 0's,

(b) in state s_0, after reading 2 consecutive 1's.

In both situations the machine must choose between two possibilities: either it is still reading a symbol in the *first* part w of the string ww^R, or it has just reached the first symbol of the *second* part w^R (i.e., it has just reached thc "midpoint" of the sequence).

Since the machine has no way of knowing which of the two possibilities it has encountered (at least not until it has read the *last* symbol of the input string), it must allow for *both* possibilities. This is where the nondeterminism becomes unavoidable (a fact which we will not prove, though). Note that if the machine "thinks" it has reached the midpoint of the string, it changes from state s_0 to s_1, and then it keeps removing symbols from the top of the pushdown stack.

In contrast to the language L = $\{ww^R\}$, the language $\{wcw^R\}$ of the previous example could be recognized by a deterministic machine since the special "center symbol" c marked the midpoint of an input string.

Table 12. State-transition table for example 49.

Current state	Current input	Current top pushdown symbol	Next state	Next pushdown symbol
s_0	0	X	s_0	Y
s_0	1	X	s_0	Z
s_0	0	Y	s_0	Y
			s_1	ε
s_0	0	Z	s_0	Y
s_0	1	Y	s_0	Z
s_0	1	Z	s_0	Z
			s_1	ε
s_1	0	Y	s_1	ε
s_1	1	Z	s_1	ε
s_0	ε	X	s_1	ε
s_1	ε	X	s_1	ε

Linear-Bounded Automata

Having so far found three types of machines (finite-state automata, Turing machines and nondeterministic pushdown automata) that recognize the classes of regular, recursive, and context-free languages, respectively, we now complete this subject by presenting machines that recognize context-sensitive languages.

Definition 30:

A *linear bounded automaton of multiplicity K* is a Turing machine which, given a tape with an input sequence of length n (i.e., occupying n adjacent squares) is only allowed to use K n squares on the tape. In other words, an LBA is a Turing machine for which the length of the usable tape is a *linear* function of the length of the input sequence. The multiplicity (K) is an integer that is intrinsically associated with the particular LBA. The following theorem shows that the multiplicity K can be assumed to be 1 without loss of generality.

Theorem 16:

For any LBA L_1 of multiplicity K (K>1) there exists another LBA L_2 of multiplicity 1 which can perform the same calculations as L_1.

We won't go into the details of the proof but rather we mention that the main idea of the proof consists in converting a tape of length Kn (input tape for L_1) into a tape of length n with K "tracks," which then serves as input tape for L_2. We also assume that the input sequence is bounded on the left by a left end marker ¢ and on the right by a right end marker \$. As L_2 scans the K-track input tape, it moves to the next track any time it encounters the right end marker \$.

Because of this theorem, most authors only consider LBA's of *multiplicity* 1, i.e., they define an LBA as a Turing machine that is never allowed to leave that portion of the tape initially containing the input sequence. Since an LBA is a special Turing machine, we can talk about *deterministic and non-deterministic LBA's* (denoted DLBA's and NDLBA's).

Let *L(DLBA)* be the *class of languages accepted by DLBA's*, and let *L(NDLBA)* be the *class of languages accepted by NDLBA's*.

Unsolved Problem: Is L(DLBA) = L(NDLBA)?

Here is a first example of a machine for which it is not known whether the nondeterministic prototype is more powerful than the deterministic one. Of course, as always $L(DLBA) \subset L(NDLBA)$, but does the inverse inclusion also hold?

Concerning the languages accepted by NDLBA's, Landweber and Kuroda (1963 and 1964) proved the following result:

Theorem 17:

The class of languages accepted by NDLBA's coincides with the class of context-sensitive languages, i.e., L(NDLBA) = {type 1}.

Even though we do not know whether L(DLBA) = L(NDLBA), we have a lower bound on L(DLBA), namely, every context-free language is accepted by a DLBA: {type 2} $\subset$ L(DLBA). In fact the two sets are not equal, since the language $L = \{0^N 1^N 2^N / N \geq 1\}$ is not context-free, but L ϵ L(DLBA).

Summarizing the results of this chapter we have the following chain of set-theoretic inclusions:

$$\{\text{type } 3\} \subset L(DPDA) \subset L(NDPDA) = \{\text{type } 2\} \subset L(DLBA) \overset{?}{\subseteq} L(NDLBA) = \{\text{type } 1\}.$$

In Table 13 we present the four classes of formal languages and the associated recognizing machines.

Table 13. The classes of formal languages and associated recognizing machines.

Language	Recognizing machine
Regular	DFSA = NDFSA
Context-free	NDPDA $\supset$ DPDA
Context-sensitive	NDLBA $\supseteq$ DLBA (?)
Recursive	DTM = NDTM

ACKNOWLEDGMENTS

I would like to thank Kelly Booth and Mary Zosel, both from LLL's Computation Department, for presenting to my class some current applications of the abstract theory contained in this report. Also, I want to express my gratitude towards the members of my class for their fruitful and stimulating criticisms and discussions that have led to a number of improvements in the notes.

APPENDIX 1 - NOTES ON NP-COMPLETE PROBLEMS

by Kelly Booth

Background

Previous lectures have introduced the idea of an algorithm, either as a computer program written in a language like FORTRAN or ALGOL, a Turing-machine description, or as a general phrase-structure grammar (Post production system).

It can be shown that *all* of these definitions define the same set of problems. We can solve a problem using a program if and only if there is a Turing machine that solves the problem and if and only if there is a phrase-structure grammar that solves the problem.

Church's thesis is the intrinsically unverifiable assertion that *any* definition of effective computability will lead to exactly the same set of computable problems.

We also know that there are noncomputable (undecidable, nonrecursive) problems that have no solution.

In what follows, we examine the *relative complexity* of problems. Ignoring all unsolvable problems (who needs to waste time solving the halting problem!), there are still problems with solutions so complicated that for all *practical* purposes we cannot solve them, even though *theoretically* solutions exist.

Polynomial Transformability

A problem P has *time complexity* T(n) on a Turing machine if there is some Turing machine that solves the problem in T(n) steps for inputs of size n.

As an example, a Turing machine can recognize a string of 0's and 1's in which no two 0's are consecutive in linear time, simply by scanning the string. Thus $T(n) = c \cdot n$ for some constant $c>0$.

Edmonds has suggested that a useful criterion for deciding whether a problem has a practical solution is to demand that its complexity be a *polynomial* function of n. Thus, the previous example qualifies, but $T(n) = 2^n$ does not. This seems to be a useful rule-of-thumb, and we will simply use it, without further debate.

One comment, however, should be made. All of the models of computation cited earlier have the following property: model I (e.g., Turing machines) can be simulated by model II (e.g., computer programs) in a polynomial number of steps. Thus if a problem has Turing-machine complexity T(n), its complexity as a computer program is at worst P(T(n)) where P(x) is a polynomial. In particular, if T(n) is itself a polynomial, then so is P(T(n)).

This last remark implies that there is no loss of generality in restricting attention to Turing machines.

We say that a problem P_1 is *polynomially transformable* to a problem P_2 if and only if there is an algorithm (running in polynomial time) that converts inputs for problem P_1 to inputs for problem P_2, and this translation has the property that I_1 is a solution for P_1 if and only if I_2 is a solution for P_2 (I_2 is the encoded I_1).

For example, consider the problem P_1, which is "n is odd," and the problem P_2, which is "m and n are relatively prime." If we encode inputs "n" for P_1 as "2,n" for P_2, then it is clear that using an algorithm for P_2 we can solve P_1. Notice that the encoding is trivially polynomial.

The Classes **P** and **NP**

A problem is said to be in **P** if and only if it can be solved by a Turing machine that has polynomial complexity. Here, we mean a *deterministic* Turing machine. If we relax this to mean a *nondeterministic* Turing machine, then we have the class **NP**.

Amazingly enough, no one knows whether **P** = **NP**! It might be that allowing a Turing machine to "guess" (the nondeterminstic moves) provides absolutely no new power in terms of polynomial computability.

A problem P ϵ **NP** is *NP-complete* if and only if for any problem P' ϵ **NP**, P' is polynomially transformable to P.

It may not be obvious that this definition is satisfied by any problem P. But there do exist such problems.

Theorem (Cook):

The problem of determining whether a set of clauses is satisfiable is NP-complete.

First, let's look at the problem. A *clause* is a Boolean expression of the form

$A \vee B \vee \overline{C}$ [here "v" is the logical "or" and the bar is negation] in which the terms are the *literals* A, B, and $\overline{C}$ (note that some terms can be negated). A set of such clauses is *satisfiable* if and only if there is an assignment of "true" and "false" to the literals so that *all* of the clauses are "true."

Proof Sketch:

Suppose that we have an arbitrary problem $P \in \mathbf{NP}$. Then there is some nondeterministic machine M which solves P in polynomial time P(n). We want to construct an input for the satisfiability problem.

If M requires P(n) steps to solve a problem of size n in **P**, then M does not use more that P(n) squares of its tape.

We can define literals C_{ijt} which have the meaning "cell i has symbol j at time t." By the above remarks, i and t extend from 1 to P(n). The values of j depend only on M.

Similarly we can define $S_{k,t}$ to be "M is in state k at time t," and $H_{i,t}$ to be "M is scanning cell i at time t."

It is a fact that there are only $C \cdot [P(n)]^2$ literals so defined and they can be used to build a set of clauses that is satisfiable if and only if M successfully accepts the original input.

The details may be found in Ref. 1.

To sum up, satisfiability captures *all* of the problems in **NP**, since an algorithm for satisfiability can be converted to an algorithm for *any* problem in **NP**.

There are many other NP-complete problems.

Theorem:

All of the following are NP-complete.

(1) Satisfiability.
(2) Does a graph have a clique of size k?
(3) Does a directed graph have a Hamilton circuit?
(4) Is a graph k colorable?
(5) Is there an efficient scheduling algorithm for a multiprocessor?
(6) Traveling salesman's problem.

(7) Knapsack problem.

(8) Subgraph isomorphism.

•

•

•

There are so many NP-complete problems, none of which are known to have polynomial solutions, that it is *very* unlikely that any of them has a polynomial solution. This follows from the corollary below.

Corollary:

P = **NP** if and only if a single NP-complete problem is in **P**.

The lesson to be learned from this is that there are probably many problems of interest that have solutions but cannot be solved on any feasible computer within the foreseeable future.

APPENDIX 2 - ANSWERS TO EXERCISES

(1) Statement (b) is the only correct one. The modified statements are:

(a) $P(A \cup B) \supset P(A) \cup P(B)$

(b) $P(A \cap B) = P(A) \cap P(B)$

(c) $P(A \setminus B) \subset P(A) \setminus P(B)$

(2) (a) $A \setminus (A \cap B) = (A \setminus A) \cup (A \setminus B) = A \setminus B$.

(b) $A = A \setminus \underbrace{(A \cap B)}_{= \emptyset} = A \setminus B$ [by (a)].

(3) $f : \mathbf{Z}^{+} \to \mathbf{Z}$ is given by

$f(2n + 1) = -n$ for $n = 0, 1, 2, \ldots$

$f(2n) = n$ for $n = 1,2,3,\ldots.$

(4) (a) If $g \cdot f = 1_A$, then f must be injective and g must be surjective. Suppose $f(a_1) = f(a_2)$ for $a_1, a_2 \in A$. Then $a_1 = g(f(a_1)) = g(f(a_2)) = a_2$, so f is injective. Also, for every $a \in A$, $a = g(f(a))$, so g is surjective.

(b) If $f \cdot g = 1_B$, then f is surjective and g is injective (by part (a)). Thus, $g \cdot f = 1_A$, *and* $f \cdot g = 1_B$ implies that both f and g are one-to-one and onto, and inverse to each other. They therefore establish an isomorphism between sets A and B.

(5) $A \in P(A) = P(B)$ implies $A \in P(B)$, i.e., $A \subset B$.
$B \in P(B) = P(A)$ implies $B \in P(A)$, i.e., $B \subset A$.
Thus, $A = B$.

(6) L(G) contains all strings of matched parentheses, i.e., all strings of parentheses that one could encounter in arithmetic expressions. The grammar is context-free.

(7) Modify the sets V and P by introducing non-terminal symbols $A_1, A_2, \ldots, A_{k-1}$, and defining P' by

$$P' = \left\{ \begin{array}{rcl} \sigma & \rightarrow & x_1 A_1 \\ A_1 & \rightarrow & x_2 A_2 \\ A_2 & \rightarrow & x_3 A_3 \\ & \vdots & \\ & \vdots & \\ A_{k-1} & \rightarrow & x_k A \\ A_{k-1} & \rightarrow & x_k \end{array} \right\}$$

Clearly this grammar G' is regular, and L(G) = L(G').

(8) The language is empty: L(G) = $\emptyset$. This follows from the fact that the first and third productions will never allow us to replace the nonterminal "B" by a terminal string - any string we derive will always end with .....B?. Therefore, L(G) is empty.

(9) Define the grammar G = (V,Σ,P,σ) by

$\Sigma = \{+, -, *, /, (,), a, b, c, \ldots, y, z\}$,

$V = \Sigma \cup \{\sigma\}$, and the productions P:

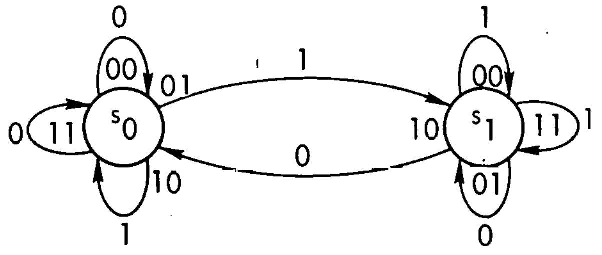
$$\begin{array}{l} \sigma \rightarrow \sigma + \sigma \\ \sigma \rightarrow \sigma - \sigma \\ \sigma \rightarrow \sigma * \sigma \\ \sigma \rightarrow \sigma / \sigma \\ \sigma \rightarrow \sigma ** \sigma \\ \sigma \rightarrow (\sigma) \\ \sigma \rightarrow a \\ \quad \vdots \\ \sigma \rightarrow z \end{array}$$

Clearly, L(G) will contain all FORTRAN arithmetic statements (and more).

(10) Using the above productions, we derive the statement $a - (a^*((b + c)/d)**e)$ as follows:

$$\sigma \to \sigma - \sigma \to a - \sigma \to a - (\sigma) \to a - (\sigma{*}{*}\sigma)$$
$$\to a - (\sigma{*}{*}e) \to a - (\sigma{*}\sigma{*}{*}e) \to a - (a{*}\sigma{*}{*}e) \to$$
$$\to a - (a{*}(\sigma){*}{*}e) \to a - (a{*}(\sigma/\sigma){*}{*}e) \to$$
$$\to a - (a{*}(\sigma/d){*}{*}e) \to a - (a{*}((\sigma)/d){*}{*}e) \to$$
$$\to a - (a{*}((\sigma+\sigma)/d){*}{*}e) \to a - (a{*}((\sigma + c)/d){*}{*}e) \to$$
$$\to a - (a{*}((b + c)/d){*}{*}e)$$

(11) This is the state-diagram for a Mealy-machine that subtracts binary numbers.



(12) Let $A = (K,\Sigma,F,s_0,f)$, where $K = \{s_0,s_1,s_2\}$, $\Sigma = \{0,1\}$, $F = \{s_0\}$, and

$$f(s_0,0) = s_1 \qquad f(s_1,0) = s_2 \qquad f(s_2,0) = s_2$$
$$f(s_0,1) = s_0 \qquad f(s_1,1) = s_0 \qquad f(_2,1) = s_2 \ .$$

Note that inputting two or more 0's in a row will kick us into state s_2 from which we can never return.

(13) (a) Let $B = (K',\Sigma,0,s_0',f',g')$ be a Moore automaton. Define a Mealy machine $A = (K,\Sigma,0,s_0,f,g)$ by:

$K = K'$, $s_0 = s_0'$, $f = f'$, $g = g' \cdot f' : K' \times \Sigma \to 0$.

Now $g(x_1) = g(s_0', x_1) = g'(f'(s_0', x_1)) = g'(x_1)$

and $g(x_1 \ldots x_n) = g(f(s_0', x_1 \ldots x_{n-1}), x_n)$

$= g'(f'(f(s_0', x_1 \ldots x_{n-1}), x_n))$

$= g'(f'(s_0', x_1 \ldots x_n)) = g'(x_1 \ldots x_n)$.

Thus, A is equivalent to B.

(b) Start out with a Mealy machine $A = (K, \Sigma, 0, s_0,f,g)$.
Define a Moore-machine $B = (K',\Sigma,0,s_0',f',g')$ by:

$K' = K \times \Sigma$, $s_0' = (s_0,\varepsilon)$;

let $f': K' \times \Sigma \to K'$ be $f'((s,x),y) = (f(s,x),y)$

and let $g': K' \to 0$ be $g'((s,x)) = g(s,x)$, where $x,y \in \Sigma$, $s \in K$.

Then:

$$g'(x_1) = g'(f'(s_0',x_1)) = g'(f'((s_0,\varepsilon),x_1))$$
$$= g'((f(s_0,\varepsilon),x_1)) = g(fs_0,\varepsilon),x_1)$$
$$= g(s_0,x_1) = g(x_1).$$

Also:

$$g'(x_1\ldots x_n) = g'(f'(s_0',x_1\ldots x_n)) = g'(f'(f'(s_0',x_1\ldots x_{n-1}),x_n))$$
$$= g'(f'(f'((s_0,\varepsilon),x_1\ldots x_{n-1}),x_n)) = g'(f'((s_0,x_1\ldots x_{n-1}),x_n))$$
$$= g'((f(s_0,x_1\ldots x_{n-1}),x_n)) = g(f(s_0,x_1\ldots x_{n-1}),x_n)$$
$$= g(x_1\ldots x_n).$$

Thus, we have shown that B is a Moore machine equivalent to A. Note that if A has n states ($|K| = n$) and m input symbols ($|\Sigma| = m$), then B has m•n states ($|K'| = |K \times \Sigma| = m\cdot n$). In going from a Mealy to a Moore model we have to increase the number of states to make up for the loss of input-dependence in the output function $g' : K' \to O$.

(14) Define $A^R = (K',\Sigma,F',S_0',f')$ as follows:

$K' = K;\quad F' = \{s_0\};\quad S_0' = F;$

$f'(s,x) = \{\tilde{s} \,/\, \tilde{s} \in K \text{ and } f(\tilde{s},x) = s\}$

To show that $T(A^R) = L^R$, let $w = x_1x_2\ldots x_{n-1}x_n$ be a string in $L = T(A)$, and suppose that $f(s_0,x_1) = s_1$, $f(s_1,x_2) = s_2,\ldots, f(s_{n-1},x_n) = s_n \in F$ for some states $s_1,s_2,\ldots,s_{n-1},s_n \in K$. Now we input the reversal $w^R = x_nx_{n-1}\ldots x_2x_1$ to A^R, and we must find that $w^R \in T(A^R)$, i.e., $f'(S_0', w^R) \cap F' \neq 0$, but $F' = \{s_0\}$, so we must show that $s_0 \in f'(S_0',w^R)$.

Now $\quad f'(S_0', x_n) = \{s \in K \,/\, f(s,x_n) \in F\} \neq \emptyset$,

since $\quad s_{n-1} \in f'(S_0',x_n)$;

and $\quad f'(s_{n-1},x_{n-1}) = \{s \in K \,/\, f(s,x_{n-1}) = s_{n-1}\} \neq \emptyset$,

since $\quad s_{n-2} \in f'(s_{n-1}, x_{n-1})$.

Continuing this argument, we see that

$$s_{i-1} \in f'(s_i,x_i) = \{s \in K \mid f(s,x_i) = s_i\}.$$

Finally, $s_0 \in f'(s_1,x_1) = \{s \in K \mid f(s,x_1) = s_1\}$.

Informally, we can draw this diagram:

$$\begin{tikzcd}
F \arrow[r, "x_n"] & \{\ldots, s_{n-1}, \ldots\} \arrow[dr, "x_{n-1}"] & & & \\
& & \{\ldots, s_{n-2}, \ldots\} \arrow[r, "x_{n-2}"] & \cdots \arrow[r, "x_2"] & \{\ldots, s_1, \ldots\} \arrow[d, "x_1"] \\
& & & & \{\ldots, s_0, \ldots\}
\end{tikzcd}$$

As we see, by inputting $w^R = x_n x_{n-1} \ldots x_2 x_1$ to A^R, we can finally arrive in state s_0, so that $s_0 \in f'(S_0', w^R)$, i.e., $w^R \in T(A^R)$.

It is not hard to show also, that if $v \in T(A^R)$, i.e., $s_0 \in f'(S_0',v)$, then $v = w^R$ for some $w \in T(A)$.

(15) $A^R = (K',\Sigma,F',S_0',f')$, where $K' = \{s_0,s_1,s_2\}$, $F' = \{s_0\}$, $S_0' = F = \{s_2\}$, and the next-state function f' is given by:

$$f'(s_0,0) = \{s \in K \mid f(s,0) = s_0\} = \{s_0,s_1,s_2\}$$
$$f'(s_0,1) = \{s \in K \mid f(s,1) = s_0\} = \emptyset$$
$$f'(s_1,0) = \{s \in K \mid f(s,0) = s_1\} = \emptyset$$
$$f'(s_1,0) = \{s \in K \mid f(s,1) = s_1\} = \{s_0\}$$
$$f'(s_2,0) = \{s \in K \mid f(s,0) = s_2\} = \emptyset$$
$$f'(s_2,1) = \{s \in K \mid f(s,1) = s_2\} = \{s_1,s_2\}$$

For example, the string 111010 ϵ $T(A^R)$ can be drawn as follows:

$$\begin{tikzcd}
s'_0 \arrow[r, "1"] & \{s_1, s_2\} \arrow[dr, "1"'] & & & & & & \\
& & \{s_1, s_2\} \arrow[dr, "1"'] & & & & & \\
& & & \{s_0\} \arrow[dr, "0"'] & & & & \\
& & & & \{s_0, s_1, s_2\} \arrow[dr, "1"'] & & & \\
& & & & & \{s_0\} \arrow[dr, "0"'] & & F' \arrow[dl] \\
& & & & & & \{s_0, s_1, s_2\} &
\end{tikzcd}$$

The string 1011 $\notin$ $T(A^R)$, since we have:

$$\begin{tikzcd}
s'_0 \arrow[r, "1"] & \{s_1, s_2\} \arrow[d, "0"'] \arrow[dr, "0"] & \\
& \emptyset & \emptyset
\end{tikzcd}$$

(16) The Turing machine whose input tape and state diagram is given in problem (17) will add the numbers m and n, and write the result (m + n 1's) after the square with the "Z."

(17) The above answer solves this problem.

(18) Assume T is an arbitrary Turing-machine and $(s_i,x_j,s_{ij},x_{ij},L)$ and $(s_k,x_l,s_{kl},s_{kl},R)$ are two quintuples with $s_{ij} = s_{kl}$ (so T is *not* a directed state machine). Add two states, say s'_{ij} and s''_{ij}, and replacc the quintuple $(s_k,x_l,s_{kl},x_{kl},R)$ by the 3 quintuples:

$(s_k,x_l,s'_{ij},x_{kl},R)$

(s'_{ij},x,s''_{ij},x,R), for any x.

(s''_{ij},x,s_{kl},x,L), for any x.

Repeating this procedure for all pairs of "noncompatible" quintuples (adding 2 new states, and replacing one of the quintuples by 3 new quintuples as shown) will result in a directed-state machine that is equivalent to the original machine.

(19) The finite-state machine that is equivalent to a "one-way" Turing machine has the same states and the same input and output alphabets. Furthermore, the next-state function F and output function G are

$F(s_i,x_j) = s_{ij}$

$G(s_i,x_j) = x_{ij}$

where $(s_i,x_j,s_{ij},x_{ij},R)$ is a quintuple. This completely defines the finite-state automaton.

APPENDIX 3 - ABBREVIATIONS AND GLOSSARY OF SYMBOLS

Abbreviations

BNF	Backus-Naur form of a grammar
DFSA	Deterministic finite-state automaton
DLBA	Deterministic linear-bounded automaton
DPDA	Deterministic pushdown automaton
DTM	Deterministic Turing machine
LBA	Linear-bounded automaton
NDFSA	Nondeterministic finite-state automaton
NDLBA	Nondeterministic linear-bounded automaton
NDPDA	Nondeterministic pushdown automaton
NDTM	Nondeterministic Turing machine
PDA	Pushdown automaton

Glossary of Symbols

Set Theory

$\{a,b,c,d\}$	Set of elements a, b, c, and d
$\{x / x \text{ has } P\}$	Set of all elements (x) that have property P
$a \in S$	a is an element of (belongs to) the set S
$a \notin S$	a does not belong to the set S
$\emptyset$	Empty set (containing no elements)
$A \subset B$ $B \supset A$	A is a subset of B
$A \cup B$	Union of sets A and B
$A \cap B$	Intersection of sets A and B
$A \setminus B$	Difference of sets A and B
$A \times B$	Cartesian product of sets A and B
$P(A)$ 2^A	Power-set (set of all subsets) of the set A
$\|S\|$	Cardinality (size) of the set S
$\aleph_0$	Cardinality of all countable sets (Aleph zero)
(a,b)	Ordered pair of elements a and b
$\mathbf{R}$	Set of all real numbers
$\mathbf{Q}$	Set of all rational numbers (fractions)
$\mathbf{Z}$	Set of all integers

Z^+	Set of all non-negative integers
$f:A \to B$	f is a function from set A to set B
$f(a)$	Function-value of f evaluated at element a
$g \cdot f$	Composition of functions f and g (first carry out f, then g)
D_f	Domain of function f
R_f	Range of function f
$a \sim b$	Equivalence relation (a is equivalent to b)
$[a]$	Equivalence class (set of all elements that are equivalent to the element a)
$a \equiv b \pmod{c}$	a is congruent to b modulo c; means "a and b have same remainder if divided by c."
$A/\sim$	Quotient set (set of all equivalence classes)

Logical Symbols

$\Rightarrow$	"implies"
$\Leftrightarrow$	"if and only if"
$\exists x$	"there exists an x such that...."
$\forall x$	"for all elements x, ..."

Languages and Grammars

Σ	Finite alphabet (set of terminals)
Σ^*	Set of all finite strings formed from elements of Σ
V	Finite vocabulary (set of terminals and non-terminals)
ε	Empty string
$\|w\|$	Length (number of symbols) of string w
σ	Start variable (always a nonterminal)
$w \to v$	Production (w produces the string v)
$w \Rightarrow v$, $w \overset{*}{\underset{G}{\Rightarrow}} v$	Derivation (w derives the string v)
$L(G)$	Language generated by the grammar G
w^R	String w in reverse order
L^R	Language of all reversed strings
<A>	BNF-description for a non-terminal
::=	BNF for "$\to$" (production symbol)
|	BNF for "or"

Machines

K	Finite set of states
F	Subset of final (or accepting) states
S_0	Subset of initial (start-) states
s_0	Initial (start-) state
O	Finite set of output symbols
t	Time
MN	Product (concatenation) of subsets of Σ^*
M^*	$\{\varepsilon\} \cup M \cup M^2 \cup M^3 \ldots$
$T(A)$	Language (tapes) accepted by machine A
τ	Input tape
d_T	Description of Turing machine T
μ_Y	Minimization operator (partial-recursive function)
Z_0	Symbol initially at top of pushdown-stack
Γ	Finite pushdown alphabet
$L_f(P)$	Language accepted by PDA P by final state
$L_\varepsilon(P)$	Language accepted by PDA P by empty pushdown store

REFERENCE

1. Alfred Aho and John E. Hopcroft, The Design and Analysis of Computer Algorithms (Addison-Wesley, Reading, Mass., 1974), p. 379.

BIBLIOGRAPHY

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

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

M. Minsky, Computation: Finite and Infinite Machines (Prentice-Hall, Englewood Cliffs, N.J., 1967).

J. B. Tremblay and R. Manohar, Discrete Mathematical Structures with Applications in Computer Science (McGraw-Hill, New York, N.Y., 1975).

Printed in the United States of America
Available from
National Technical Information Service
U. S. Department of Commerce
5285 Port Royal Road
Springfield, Virginia 22151
Price: Printed Copy $___*; Microfiche $2.25

*Pages	NTIS Selling Price
1-50	$4.00
51-150	$5.45
151-325	$7.60
326-500	$10.60
501-1000	$13.60

PO/gw