2

SET THEORY

INTRODUCTION

The concept of a set is used in various disciplines and particularly often in mathematics. In this chapter, we introduce elementary set theory. An axiomatic approach to the discussion and questions of a philosophical nature will be avoided. Although our presentation remains informal, we try to indicate formal proofs which use the notation and the rules of inference given in Chap. 1. As we proceed, an analogy will be drawn between the statement calculus and the set operations leading to a set algebra which is similar to the statement algebra given earlier. Initially, the notation of set theory is introduced and certain operations are defined. Then follows an introduction to the representation of discrete structures. The concepts of relations, orderings, and functions are presented after a discussion of the algebra of sets. A particular type of function known as a binary operation prepares us for a discussion of algebraic structures, which form the subject matter of Chap. 3. A special function known as a hashing function, which maps a name into an integer and permits us to handle nonnumeric data, is discussed. The natural numbers are introduced, and the principle of mathematical induction is given. A discussion of recursive functions then follows. Certain applications

meant to motivate the reader and relate the key theoretical concepts to practical situations are given throughout the chapter. Some of these are presented as algorithms which can easily be implemented on a computer. An algorithm based on the discussion of automatic theorem proving in the statement calculus (Sec. 1-4.4) is also given.

2-1 BASIC CONCEPTS OF SET THEORY

In this section first we introduce the notation used for specifying sets. The concepts of membership and inclusion are given. Certain special sets such as the universal set, empty set, and the power set of a given set are introduced. Next, various set operations are defined. Finally, some of the basic identities of set algebra are derived.

2-1.1 Notation

Rather than defining the term "set," here we give only an intuitive idea of what a set is. By a *set* we mean a collection of objects of any sort. The word "object" is used here in a very broad sense to include even abstract objects. The following examples will illustrate the concept of a set.

The *set* of all Canadians

A *pair* of shoes

A *bouquet* of flowers

The *set* of all ideas contained in a book

A *flock* of sheep

A *collection* of rocks

Note that we encounter many words which convey the same idea as that of a set. Some of these words, however, are used in a more restricted sense than the term "set," while the others are synonyms. We shall use the words "class," "aggregate," and "collection" as synonyms of the word "set," particularly to avoid using the same word repeatedly in a given sentence. For example, a set of sets may be called a collection of sets.

Generally speaking, we think of a set as a collection of objects which share some common property. For example, in mathematics it is common to consider a set of lines, a set of triangles, a set of real numbers, etc. However, this restriction is not necessary, and a set consisting of a horse, the letter A, a jacket, and Mr. Smith is an acceptable example of a set, although it may be uninteresting and not important.

A fundamental concept of set theory is that of membership or belonging to a set. Any object belonging to a set is called a *member* or an *element* of that set. A set is said to be *well defined* if it is possible to determine, by means of certain rules, whether any given object is a member of the set.

Scanned with OKEN Scanner

Capital letters, with or without subscripts, will be used throughout to denote sets, and lowercase letters will be used to denote the elements. Some exceptions to these rules will be made in order to conform to standard practice.

If an element p belongs to a set A, then we write

$$p \in A$$

which is read as "p is an element of the set A" or "p belongs to the set A," or "p is in A." If there exists an object q which does not belong to the set A, then we express this fact as

$$q \notin A$$

which is equivalent to the negation of the statement "q is in A," that is,

$$\neg(q \in A) \Leftrightarrow q \notin A$$

There are several ways in which a set can be specified. For example, a set consisting of the elements 1, 3, and a is generally written as

$$\{1, 3, a\}$$

The names of the elements are enclosed in braces and separated by commas. If we wish to denote this set as S, then we write

$$S = \{1, 3, a\}$$

where the equality sign is understood to mean that S is the set $\{1, 3, a\}$. Obviously,

$$1 \in S \qquad 3 \in S \qquad a \in S \qquad \text{and} \qquad 2 \notin S$$

This method of specifying a set is not convenient, and it is not always possible to use it. In general, a set can be defined or characterized by a predicate. Examples of such sets are

$$S_1 = \{x \mid x \text{ is an odd positive integer}\}$$
$$S_2 = \{x \mid x \text{ is a province of Canada}\}$$
$$S_3 = \{x \mid x \text{ is a river}\}$$
$$S_4 = \{x \mid x = a \text{ or } x = b\} .$$

If we let $P(x)$ denote any predicate, then $\{x \mid P(x)\}$ defines a set. An element b belongs to this set if $P(b)$ is true; otherwise b does not belong to the set. This statement would be written symbolically as

$$(y)\,(P(y) \Leftrightarrow y \in \{x \mid P(x)\})$$

If we denote the set $\{x \mid P(x)\}$ by A, then $A = \{x \mid P(x)\}$ and

$$(y)\,(y \in A \Leftrightarrow y \in \{x \mid x \in A\})$$

Other sets which are specified by listing the elements can also be characterized by means of predicates. For example, the set $\{1, 3, a\}$ could be defined as

$$\{x \mid (x = 1) \vee (x = 3) \vee (x = a)\}$$

Although it is possible to characterize any set by a predicate, it is sometimes convenient to specify sets by yet another method, such as

$$S_5 = \{1, 3, 5, \ldots\}$$
$$S_6 = \{2, 4, 6, \ldots, 18\}$$
$$S_7 = \{a, a^2, a^3, \ldots\}$$

In this representation the missing elements can be determined from the elements present and from the context.

The number of distinct elements present in a set may be finite or infinite. We shall call a set *finite* if it contains a finite number of distinguishable elements; otherwise, a set is *infinite*. Precise definitions of a finite and an infinite set are given in Sec. 2-5.2.

Note that no restriction has been placed on the objects that can be members of a set. It is not unusual to have sets whose members are themselves sets, such as

$$S = \{a, \{1, 2\}, p, \{q\}\}$$

However, it is important to distinguish between the set $\{q\}$, which is an element of S, and the element q, which is a member of $\{q\}$ but not a member of S.

2-1.2 Inclusion and Equality of Sets

In Sec. 2-1.1, we discussed the notion of membership of an element in a set. Another basic concept in set theory is that of inclusion.

Definition 2-1.1 Let A and B be any two sets. If every element of A is an element of B, then A is called a *subset* of B, or A is said to be *included* in B, or B *includes* A. Symbolically, this relation is denoted by $A \subseteq B$, or equivalently by $B \supseteq A$. Alternatively,

$$A \subseteq B \Leftrightarrow (x)\,(x \in A \rightarrow x \in B) \Leftrightarrow B \supseteq A$$

It is important at this stage to distinguish between membership and inclusion. We illustrate the difference between these two. Let

$$A = \{1, 2, 3\} \qquad B = \{1, 2\} \qquad C = \{1, 3\} \qquad \text{and} \qquad D = \{3\}$$

then

$$B \subseteq A \qquad C \subseteq A \qquad \text{and} \qquad D \subseteq A$$

or

$$\{1, 2\} \subseteq \{1, 2, 3\} \qquad \{1, 3\} \subseteq \{1, 2, 3\} \qquad \{3\} \subseteq \{1, 2, 3\}$$

On the other hand, $1 \in \{1, 2, 3\}$, and 1 is not included in $\{1, 2, 3\}$, though $\{1\} \subseteq \{1, 2, 3\}$. Only a set can be included in or can be a subset of another set, while only elements can be members of a set. Of course, a set may sometimes have other sets as elements. For example, if $A = \{\{1\}, 2, 3\}$, then

$$\{1\} \in A \qquad \{\{1\}, 2\} \subseteq A \qquad \{\{1\}\} \subseteq A \qquad 2 \in A \qquad \{2, 3\} \subseteq A$$

The following are some of the important properties of set inclusion. For any sets A, B, and C

$$A \subseteq A \qquad \text{(reflexive)} \qquad (1)$$

$$(A \subseteq B) \wedge (B \subseteq C) \Rightarrow (A \subseteq C) \qquad \text{(transitive)} \qquad (2)$$

Scanned with OKEN Scanner

It is enough to note at this stage that set inclusion is reflexive and transitive. These terms are explained in Sec. 2-3.2. The proof of Statement (1) is obvious, while Statement (2) can be proved by using the implication

$$(x)(x \in A \rightarrow x \in B) \wedge (x)(x \in B \rightarrow x \in C) \Rightarrow (x)(x \in A \rightarrow x \in C)$$

(see Example 2, Sec. 1-6.4). For two sets A and B, note that $A \subseteq B$ does not necessarily imply $B \subseteq A$ except for the following case.

Definition 2-1.2 Two sets A and B are said to be *equal* (extensionally equal) iff $A \subseteq B$ and $B \subseteq A$, or symbolically,

$$A = B \Leftrightarrow (A \subseteq B \wedge B \subseteq A)$$

From the equivalence

$$(x)((P(x) \rightarrow Q(x)) \wedge (Q(x) \rightarrow P(x))) \Leftrightarrow (x)(P(x) \rightleftarrows Q(x))$$

we can alternatively define the equality of two sets as

$$A = B \Leftrightarrow (x)(x \in A \rightleftarrows x \in B)$$

We now give some examples of sets that are equal and sets that are not equal.

$\{1, 2, 4\} = \{1, 2, 2, 4\}$.

$\{1, 4, 2\} = \{1, 2, 4\}$.

If $P = \{\{1, 2\}, 4\}$ and $Q = \{1, 2, 4\}$, then $P \neq Q$.

$\{\{1\}\} \neq \{1\}$ because $\{1\} \in \{\{1\}\}$ while $1 \in \{1\}$.

If $A = \{x \mid x(x-1) = 0\}$ and $B = \{0, 1\}$, then $A = B$.

$\{1, 3, 5, \ldots\} = \{x \mid x$ is an odd positive integer$\}$.

From the definition of equality of sets it is clear that

$$A = B \Leftrightarrow B = A$$

The equality of sets is reflexive, symmetric, and transitive.

Definition 2-1.3 A set A is called a *proper subset* of a set B if $A \subseteq B$ and $A \neq B$. Symbolically it is written as $A \subset B$, so that

$$A \subset B \Leftrightarrow (A \subseteq B \wedge A \neq B)$$

$A \subset B$ is also called a proper inclusion.

A proper inclusion is not reflexive; however, it is transitive, i.e.,

$$(A \subset B) \wedge (B \subset C) \Rightarrow (A \subset C)$$

We shall now introduce two special sets, of which one includes every set under discussion while the other is included in every set under discussion.

Definition 2-1.4 A set is called a *universal set* if it includes every set under discussion. A universal set will be denoted by E.

It follows from the definition that for any set A, we have $A \subseteq E$. Thus every element $x \in E$, that is, $(x)(x \in E)$ is identically true. One could specify E in a variety of ways, e.g.,

$$E = \{x \mid P(x) \vee \neg P(x)\}$$

where $P(x)$ is any predicate. It is easy to show that all such sets are equal to the set E. The introduction of the universal set makes the notion of $b \notin A$ more definite in the sense that b is assumed to be in E. The universal set is the same as the universe of discourse given in Sec. 1-5.5.

Definition 2-1.5 A set which does not contain any element is called an *empty set* or a *null set*. An empty set will be denoted by $\varnothing$.

It follows from the definition that for an empty set $\varnothing$ and any $x, x \in \varnothing$ is a contradiction, that is, $(x)(x \in \varnothing)$ is a contradiction. Thus for any set A, $\varnothing \subseteq A$, because $(x)(x \in \varnothing \Rightarrow x \in A)$. One could specify $\varnothing$ in a variety of ways, e.g.,

$$\varnothing = \{x \mid P(x) \wedge \neg P(x)\}$$

where $P(x)$ is any predicate. It is easy to show that all such sets are identical to the set $\varnothing$.

2-1.3 The Power Set

Given any set A, we know that the null set $\varnothing$ and the set A are both subsets of A. Also for any element $a \in A$, the set $\{a\}$ is a subset of A. Similarly, we can consider other subsets of A. Rather than finding individual subsets of A, we would like to say something about the set of all subsets of A.

Definition 2-1.6 For a set A, a collection or family of all subsets of A is called the *power set* of A. The power set of A is denoted by $\rho(A)$ or 2^A, so that

$$\rho(A) = 2^A = \{x \mid x \subseteq A\}$$

Let us consider some finite sets and their power sets. The power set of the null set $\varnothing$ has only the element $\varnothing$; hence $\rho(\varnothing) = \{\varnothing\}$. For a set $S_1 = \{a\}$, the power set $\rho(S_1) = \{\varnothing, \{a\}\} = \{\varnothing, S_1\}$. For $S_2 = \{a, b\}$, $\rho(S_2) = \{\varnothing, \{a\}, \{b\}, S_2\}$, and for $S_3 = \{a, b, c\}$, $\rho(S_3) = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, S_3\}$.

We now introduce a notation by which one can designate every subset of a finite set in a unique manner. Before we describe this notation, it would be useful to assume that the elements of the given set are ordered in some way, so that a particular element may be called the first element, the next element the second, and so on. No such ordering of the elements of a set is implied in the

Scanned with OKEN Scanner

definition of a set. However, for the purpose of representing a set on a computer it is generally necessary to prescribe some arbitrary order; i.e., to each element is attached a label which describes the position of the element with respect to other elements of the set. As an example, let us assume that in the set S_2 given earlier we let a be the first element and b be the second element. Now, in any subset of a given set, some elements of the set are present while the remaining ones are absent. We shall use this idea, together with the ordering prescribed on the elements of a given set, to designate the subsets. For example, the subsets of S_2 may be designated as

$$\varnothing = B_{00} \qquad \{a\} = B_{10} \qquad \{b\} = B_{01} \qquad \text{and} \qquad \{a, b\} = B_{11}$$

where the subscripts of B contain 1 or 0 in the first position on the left depending on whether the first element, viz., a, is present (or not). Similarly, the subscript has 1 or 0 in the second position from the left depending on whether b, the second member, is present (or not). As there are only two elements in S_2, we need only the subscripts 00, 01, 10, and 11. Conversely, given any one of these $2^2 = 4$ subscripts, we can determine the elements of the corresponding subset. For example, $B_{01} = \{b\}$. Note that it is only the subscript that determines the elements of the subset. The use of the letter B in naming the subsets is incidental. A similar technique has been used earlier in assigning names to maxterms and minterms. This method will also be used in the representation of data on a computer.

Consider the set $J = \{00, 01, 10, 11\}$ or $J = \{i \mid i \text{ is a binary integer}, 00 \leq i \leq 11\}$; then $\rho(S_2) = \{B_i \mid i \in J\}$. Similarly,

$$\rho(S_3) = \{B_i \mid i \in J\}$$

where $J = \{i \mid i \text{ is a binary integer}, 000 \leq i \leq 111\}$. From $S_3 = \{a, b, c\}$ we have $B_{001} = \{c\}$, $B_{101} = \{a, c\}$, and $B_{011} = \{b, c\}$.

The above notation can be generalized to designate the subsets of a set having n distinct elements. Obviously, there are 2^n such subsets. The subscripts designating the subsets range over the binary representations of the decimal integers 0 to $2^n - 1$. Care must be taken to insert as many zeros on the left of this binary integer as necessary in order to have exactly n digits in all. One can use decimal integers from 0 to $2^n - 1$ and convert them only at the time when the elements of the corresponding subsets are to be determined. As an illustration, let $S_6 = \{a_1, a_2, \ldots, a_6\}$. Obviously, there are 2^6 subsets of S_6, which we shall designate by $B_0, B_1, \ldots B_{2^6-1}$. The following examples illustrate the method to determine the elements of any subset.

$$B_7 = B_{111} = B_{000111} = \{a_4, a_5, a_6\}$$

$$B_{12} = B_{1100} = B_{001100} = \{a_3, a_4\}$$

The method of employing subscripts to designate the elements of a family of sets is used very often. Here we have used it to designate the members of a power set. It is convenient to introduce the concept of an indexed set at this stage.

Definition 2-1.7 Let $J = \{s_1, s_2, s_3, \ldots\}$ and A be a family of sets $A = \{A_{s_1}, A_{s_2}, A_{s_3}, \ldots\}$ such that for any $s_i \in J$ there corresponds a set $A_{s_i} \in A$, and also $A_{s_i} = A_{s_j}$ iff $s_i = s_j$, then A is called an *indexed set*, J the *index set*, and any subscript such as s_i in A_{s_i} is called an *index*.

An indexed family of sets can also be written as

$$A = \{A_i\}_{i \in J}$$

In particular, if $J = \mathbf{I} = \{1, 2, 3, \ldots\}$, then $A = \{A_1, A_2, A_3, \ldots\}$. Also, if $J = \mathbf{I}_n = \{1, 2, \ldots, n\}$, then $A = \{A_1, A_2, \ldots, A_n\} = \{A_i\}_{i \in \mathbf{I}_n}$. For a set S containing n elements, the power set $\rho(S)$ is written as the indexed set

$$\rho(S) = \{B_i\}_{i \in J} \qquad J = \{0, 1, 2, \ldots, 2^n - 1\}$$

EXERCISES 2-1.3

1 Give another description of the following sets and indicate those which are infinite sets.
   (a) $\{x \mid x \text{ is an integer and } 5 \leq x \leq 12\}$.
   (b) $\{2, 4, 8, \ldots\}$.
   (c) All the countries of the world.

2 Given $S = \{2, a, \{3\}, 4\}$ and $R = \{\{a\}, 3, 4, 1\}$, indicate whether the following are true or false.
   (a) $\{a\} \in S$
   (b) $\{a\} \in R$
   (c) $\{a, 4, \{3\}\} \subseteq S$
   (d) $\{\{a\}, 1, 3, 4\} \subset R$
   (e) $R = S$
   (f) $\{a\} \subseteq S$
   (g) $\{a\} \subseteq R$
   (h) $\varnothing \subset R$
   (i) $\varnothing \subseteq \{\{a\}\} \subseteq R \subseteq E$
   (j) $\{\varnothing\} \subseteq S$
   (k) $\varnothing \in R$
   (l) $\varnothing \subseteq \{\{3\}, 4\}$

3 Show that

$$(R \subseteq S) \wedge (S \subset Q) \Rightarrow R \subset Q$$

Is it correct to replace $R \subset Q$ by $R \subseteq Q$? Explain your answer.

4 Give the power sets of the following.
   (a) $\{a, \{b\}\}$
   (b) $\{1, \varnothing\}$
   (c) $\{X, Y, Z\}$

5 Given $S = \{a_1, a_2, \ldots, a_8\}$, what subsets are represented by B_{17} and B_{31}? Also, how will you designate the subsets $\{a_2, a_6, a_7\}$ and $\{a_1, a_8\}$?

2-1.4 Some Operations on Sets

In this section we introduce certain basic operations on sets. Using these operations, one can construct new sets by combining the elements of given sets. While the term "operation" and its properties are discussed in Sec. 2-4.4, it suffices to

Scanned with OKEN Scanner

say here that operations on one or more sets produce other sets according to certain rules.

Definition 2-1.8 The *intersection* of any two sets A and B, written as $A \cap B$, is the set consisting of all the elements which belong to both A and B. Symbolically,

$$A \cap B = \{x \mid (x \in A) \wedge (x \in B)\}$$

From the definition of intersection it follows that for any sets A and B,

$$A \cap B = B \cap A \qquad A \cap A = A \qquad \text{and } A \cap \varnothing = \varnothing \tag{1}$$

The first of these equalities shows that the intersection is commutative. The importance of the other two will be discussed later. The commutativity of intersection can be proved in the following manner. For any x,

$$\begin{aligned} x \in A \cap B &\Leftrightarrow x \in \{x \mid (x \in A) \wedge (x \in B)\} \\ &\Leftrightarrow (x \in A) \wedge (x \in B) \\ &\Leftrightarrow (x \in B) \wedge (x \in A) \\ &\Leftrightarrow x \in \{x \mid (x \in B) \wedge (x \in A)\} \\ &\Leftrightarrow x \in B \cap A \end{aligned}$$

The other two equalities in Eq. (1) can be proved in a similar manner.

Since $A \cap B$ is a set, we can consider its intersection with another set C, so that

$$(A \cap B) \cap C = \{x \mid x \in A \cap B \wedge x \in C\}$$

Using $(x \in A \wedge x \in B) \wedge x \in C \Leftrightarrow x \in A \wedge (x \in B \wedge x \in C)$, we can easily show that

$$(A \cap B) \cap C = A \cap (B \cap C) \qquad \text{(associative)} \tag{2}$$

In view of Eq. (2), we can write $(A \cap B) \cap C$ as $A \cap B \cap C$.

For an indexed set $A = \{A_1, A_2, \ldots, A_n\} = \{A_i\}_{i \in \mathbf{I}_n}$ where $\mathbf{I}_n = \{1, 2, \ldots, n\}$, we write

$$A_1 \cap A_2 \cap \cdots \cap A_n = \bigcap_{i=1}^{n} A_i = \bigcap_{i \in \mathbf{I}_n} A_i$$

In general, for any index set J,

$$\bigcap_{i \in J} A_i = \{x \mid x \in A_i \text{ for all } i \in J\}$$

Definition 2-1.9 Two sets A and B are called *disjoint* iff $A \cap B = \varnothing$, that is, A and B have no element in common.

Definition 2-1.10 A collection of sets is called a *disjoint collection* if, for every pair of sets in the collection, the two sets are disjoint. The elements of a disjoint collection are said to be *mutually disjoint*.

Let A be an indexed set $A = \{A_i\}_{i \in J}$. The set A is a disjoint collection iff $A_i \cap A_j = \varnothing$ for all $i, j \in J$, $i \neq j$.

EXAMPLE 1 If $A_1 = \{\{1, 2\}, \{3\}\}$, $A_2 = \{\{1\}, \{2, 3\}\}$, and $A_3 = \{\{1, 2, 3\}\}$, then show that A_1, A_2, and A_3 are mutually disjoint.

SOLUTION Note that $A_1 \cap A_2 = \varnothing$, $A_1 \cap A_3 = \varnothing$ and $A_2 \cap A_3 = \varnothing$. ////

EXAMPLE 2 Show that $A \subseteq B \Leftrightarrow A \cap B = A$.

SOLUTION Note that for any x,

$$x \in A \rightarrow x \in B \Leftrightarrow (x \in A \wedge x \in B) \rightleftarrows x \in A$$

which follows from $P \rightarrow Q \Leftrightarrow ((P \wedge Q) \rightleftarrows P)$. Now

$$A \subseteq B \Leftrightarrow (x)(x \in A \rightarrow x \in B)$$

while

$$A \cap B = A \Leftrightarrow (x)(x \in A \wedge x \in B \rightleftarrows x \in A) \qquad ////$$

Definition 2-1.11 For any two sets A and B, the *union* of A and B, written as $A \cup B$, is the set of all elements which are members of the set A or the set B or both. Symbolically, it is written as

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

From the definition, it follows that

$$A \cup B = B \cup A \qquad \text{(commutative)} \qquad A \cup \varnothing = A \qquad A \cup A = A$$
$$(A \cup B) \cup C = A \cup (B \cup C) \qquad \text{(associative)} \tag{3}$$

The last equality in Eq. (3) suggests that we can write $(A \cup B) \cup C$ as $A \cup B \cup C$. Note that

$$A \cup B \cup C = \{x \mid x \in A \vee x \in B \vee x \in C\}$$

We shall now prove one of the equalities in Eq. (3), viz., $A \cup A = A$. The proofs of the other equalities are similar. For any x,

$$\begin{aligned} x \in A \cup A &\Leftrightarrow x \in \{x \mid x \in A \vee x \in A\} \\ &\Leftrightarrow x \in A \vee x \in A \\ &\Leftrightarrow x \in A \\ &\Leftrightarrow x \in \{x \mid x \in A\} \\ &\Leftrightarrow x \in A \end{aligned}$$

For any indexed set $A = \{A_i\}_{i \in J}$,

$$\bigcup_{i \in J} A_i = \{x \mid x \in A_i \text{ for at least one } i \in J\}$$

For $J = \mathbf{I}_n = \{1, 2, \ldots, n\}$, we may write

$$\bigcup_{i=1}^{n} A_i = A_1 \cup A_2 \cup \cdots \cup A_n$$

Scanned with OKEN Scanner

EXAMPLE 3 What are $S \cup Q$ and $S \cap Q$ if $S = \{a, b, p, q\}$ and $Q = \{a, p, t\}$?

SOLUTION
$$S \cup Q = \{a, b, p, q, t\} \qquad S \cap Q = \{a, p\} \qquad ////$$

EXAMPLE 4 If $A_1 = \{1, 2\}$, $A_2 = \{2, 3\}$, and $A_3 = \{1, 2, 3, 6\}$, what are $\bigcup_{i=1}^{3} A_i$ and $\bigcap_{i=1}^{3} A_i$?

SOLUTION
$$\bigcup_{i=1}^{3} A_i = \{1, 2, 3, 6\} \qquad \bigcap_{i=1}^{3} A_i = \{2\} \qquad ////$$

Definition 2-1.12 Let A and B be any two sets. The *relative complement* of B in A (or of B with respect to A), written as $A - B$, is the set consisting of all elements of A which are not elements of B, that is,
$$A - B = \{x \mid x \in A \wedge x \notin B\} = \{x \mid x \in A \wedge \neg(x \in B)\}$$
The relative complement of B in A is also called the *difference* of A and B.

Definition 2-1.13 Let E be the universal set. For any set A, the relative complement of A with respect to E, that is, $E - A$, is called the *absolute complement* of A. The absolute complement of a set A is often called the *complement* of A and is denoted by $\sim A$. Symbolically,
$$\sim A = E - A = \{x \mid x \in E \wedge x \notin A\} = \{x \mid x \notin A\} = \{x \mid \neg(x \in A)\}$$

The following equalities follow from the definition of the complement.
$$\sim(\sim A) = \sim\sim A = A \quad \sim\varnothing = E \quad \sim E = \varnothing \quad A \cup \sim A = E \quad A \cap \sim A = \varnothing \tag{4}$$

We now prove one of these equalities, viz., that $A \cup \sim A = E$:
$$(A \cup \sim A) = \{x \mid x \in A \vee x \notin A\} = E$$

EXAMPLE 5 Given $A = \{2, 5, 6\}$, $B = \{3, 4, 2\}$, $C = \{1, 3, 4\}$, find $A - B$ and $B - A$. Show that $A - B \neq B - A$ and $A - C = A$.

SOLUTION $A - B = \{5, 6\}$, $B - A = \{3, 4\}$, and $A - C = \{2, 5, 6\}$ ////

EXAMPLE 6 Show that (a) $A - B = A \cap \sim B$ and (b) $A \subseteq B \Leftrightarrow \sim B \subseteq \sim A$.

SOLUTION

(a) For any x,
$$\begin{aligned} x \in A - B &\Leftrightarrow x \in \{x \mid x \in A \wedge x \notin B\} \\ &\Leftrightarrow x \in (A \cap \sim B) \end{aligned}$$

(b)
$$\begin{aligned} A \subseteq B &\Leftrightarrow (x)(x \in A \rightarrow x \in B) \\ &\Leftrightarrow (x)(\neg(x \in B) \rightarrow \neg(x \in A)) \\ &\Leftrightarrow (x)(x \notin B) \rightarrow x \notin A) \\ &\Leftrightarrow \sim B \subseteq \sim A \end{aligned}$$ ////

EXAMPLE 7 Show that for any two sets A and B
$$A - (A \cap B) = A - B$$

SOLUTION For any x,
$$\begin{aligned} x \in A - (A \cap B) &\Leftrightarrow x \in \{x \mid x \in A \wedge x \notin (A \cap B)\} \\ &\Leftrightarrow x \in A \wedge \sim(x \in A \wedge x \in B) \\ &\Leftrightarrow x \in A \wedge (x \notin A \vee x \notin B) \\ &\Leftrightarrow (x \in A \wedge x \notin A) \vee (x \in A \wedge x \notin B) \\ &\Leftrightarrow x \in A \wedge x \notin B \\ &\Leftrightarrow x \in \{x \mid x \in A \wedge x \notin B\} \end{aligned}$$

Definition 2-1.14 Let A and B be any two sets. The *symmetric difference* (or *Boolean sum*) of A and B is the set $A + B$ defined by
$$A + B = (A - B) \cup (B - A) \text{ or } x \in A + B \Leftrightarrow x \in \{x \mid x \in A \ \overline{\vee}\ x \in B\}$$
where $\overline{\vee}$ is the exclusive disjunction.

The following equalities are interesting and easy to prove.
$$A + B = B + A \qquad (A + B) + C = A + (B + C) \qquad A + \varnothing = A$$
$$A + A = \varnothing \quad \text{and} \quad A + B = (A \cap \sim B) \cup (B \cap \sim A) \tag{5}$$

We shall now prove one of these, viz., $A + \varnothing = A$. For any x,
$$\begin{aligned} x \in A + \varnothing &\Leftrightarrow x \in \{x \mid (x \in A \wedge x \notin \varnothing) \vee (x \in \varnothing \wedge x \notin A)\} \\ &\Leftrightarrow (x \in A \wedge x \notin \varnothing) \vee (x \in \varnothing \wedge x \notin A) \\ &\Leftrightarrow (x \in A) \vee \mathbf{F} \\ &\Leftrightarrow x \in A \\ &\Leftrightarrow x \in \{x \mid x \in A\} \\ &\Leftrightarrow x \in A \end{aligned}$$

The programming of these set operations is discussed in Sec. 2-2.

EXERCISES 2-1.4

1 Prove the equalities in Eqs. (1).

Given $A = \{x \mid x \text{ is an integer and } 1 \leq x \leq 5\}$, $B = \{3, 4, 5, 17\}$, and $C = \{1, 2, 3, \ldots\}$, find $A \cap B$, $A \cap C$, $A \cup B$, and $A \cup C$.

Show that $A \subseteq A \cup B$ and $A \cap B \subseteq A$.

Scanned with OKEN Scanner

4 Show that $A \subseteq B \Leftrightarrow A \cup B = B$.
5 If $S = \{a, b, c\}$, find nonempty disjoint sets A_1 and A_2 such that $A_1 \cup A_2 = S$. Find other solutions to this problem.
6 Prove the equalities in Eqs. (4) and (5).
7 Given $A = \{2, 3, 4\}$, $B = \{1, 2\}$, and $C = \{4, 5, 6\}$, find $A + B$, $B + C$, $A + B + C$, and $(A + B) + (B + C)$.

2-1.5 Venn Diagrams

Introduction of the universal set permits the use of a pictorial device to study the connection between the subsets of a universal set and their intersection, union, difference, and other operations. The diagrams used are called Venn diagrams. A *Venn diagram* is a schematic representation of a set by a set of points. The universal set E is represented by a set of points in a rectangle (or any other figure), and a subset, say A, of E is represented by the interior of a circle or some other simple closed curve inside the rectangle. In Fig. 2-1.1 the shaded areas represent the sets indicated below each figure. The Venn diagram for $A \subseteq B$ and $A \cap B = \varnothing$ are also given. From some of the Venn diagrams it is

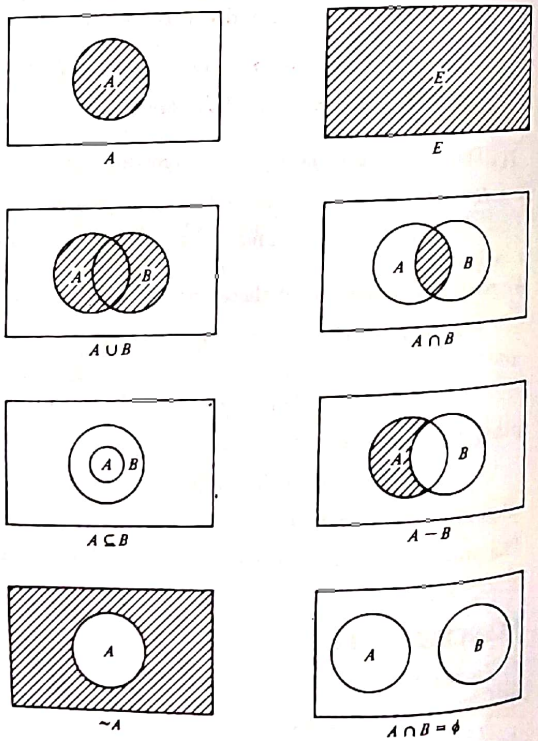


FIGURE 2-1.1 Venn diagrams.

easy to see that the following hold:

$$A \cup B = B \cup A \qquad A \cap B = B \cap A \qquad \sim(\sim A) = A.$$

Furthermore, if $A \subseteq B$, then

$$A - B = \varnothing \qquad A \cap B = A \qquad \text{and} \qquad A \cup B = B$$

It should be emphasized that the above relations between the subsets are only suggested by the Venn diagram. Venn diagrams do not provide proofs that such relations are true in general for all subsets of E. We shall demonstrate this point by a particular example.

Consider the Venn diagrams given in Fig. 2-1.2. From the first two Venn diagrams it appears that

$$A \cup B = (A \cap \sim B) \cup (B \cap \sim A) \cup (A \cap B) \tag{1}$$

From the third Venn diagram it appears that

$$A \cup B = (A \cap \sim B) \cup (B \cap \sim A)$$

This equality is not true in general, although it happens to be true for the two disjoint sets A and B.

A formal proof of Eq. (1) will now be outlined. For any x,

$$x \in A \cup B \Leftrightarrow x \in \{x \mid x \in A \vee x \in B\}$$

$$\begin{aligned}
&x \in (A \cap \sim B) \cup (B \cap \sim A) \cup (A \cap B) \\
&\quad \Leftrightarrow x \in \{x \mid x \in (A \cap \sim B) \vee x \in (B \cap \sim A) \vee x \in (A \cap B)\} \\
&\quad \Leftrightarrow x \in (A \cap \sim B) \vee x \in (B \cap \sim A) \vee x \in (A \cap B) \\
&\quad \Leftrightarrow (x \in A \wedge x \in \sim B) \vee (x \in B \wedge x \in \sim A) \vee (x \in A \wedge x \in B) \\
&\quad \Leftrightarrow (x \in A \wedge (x \in \sim B \vee x \in B)) \vee (x \in B \wedge x \in \sim A) \\
&\quad \Leftrightarrow (x \in A \vee x \in B) \wedge (x \in A \vee x \in \sim A) \\
&\quad \Leftrightarrow (x \in A \vee x \in B) \\
&\quad \Leftrightarrow x \in \{x \mid x \in A \vee x \in B\} \\
&\quad \Leftrightarrow x \in A \cup B
\end{aligned}$$

Consider the Venn diagrams in Fig. 2-1.3. From the third and fifth Venn diagrams it appears that

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \tag{2}$$

Similarly, one can show that for any three sets A, B, and C,

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C) \tag{3}$$

Equations (2) and (3) are known as the *distributive laws of union and intersection.*

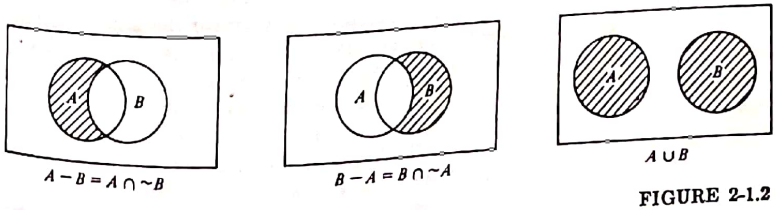


FIGURE 2-1.2

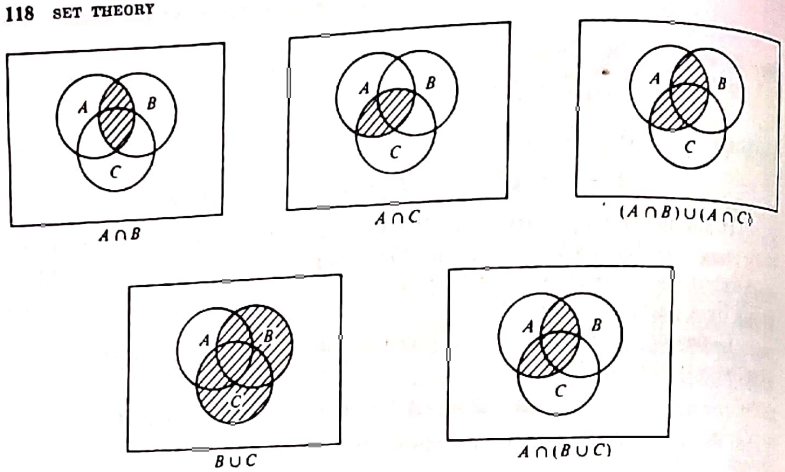


FIGURE 2-1.3

We shall now prove Eq. (3). For any x,

$$\begin{aligned} x \in A \cup (B \cap C) &\Leftrightarrow x \in \{x \mid x \in A \vee x \in (B \cap C)\} \\ &\Leftrightarrow x \in \{x \mid x \in A \vee (x \in B \wedge x \in C)\} \\ &\Leftrightarrow x \in \{x \mid (x \in A \vee x \in B) \wedge (x \in A \vee x \in C)\} \\ &\Leftrightarrow x \in (A \cup B) \cap (A \cup C) \end{aligned}$$

EXERCISES 2-1.5

1 Prove Eq. (2).
2 Draw a Venn diagram to illustrate Eq. (3).

2-1.6 Some Basic Set Identities

Set operations such as union, intersection, complementation, etc., have been defined. With the help of these operations one can construct new sets from given sets. Capital letters have been used to denote definite sets. These letters have also been used as set variables. This practice is similar to the one employed in the statement calculus. Capital letters such as $A, B, C, \ldots$ are used as set variables; they are not exactly sets, but *set formulas*. The operations on sets can also be extended to set formulas, so that $A \cup B$, $A \cap B$, $\sim A$, etc., are all set formulas. Any well-formed string involving set variables, the operations $\cap$, $\cup$, $\sim$, and parentheses is a set formula which will also be called a set for the sake of brevity.

In fact, one obtains a set from a set formula by replacing the variables by definite sets. Two set formulas are said to be equal if they are equal as sets whenever the set variables appearing in both the formulas are replaced by any sets. It is assumed that any particular variable is replaced by the same set throughout both formulas. Since the equality of set formulas does not depend upon the sets which replace the variables, these equalities are called set *identities*. Some of the basic identities describe certain properties of the operations involved and are

given special names. These properties describe an algebra called set algebra. We shall see in Chap. 4 that both the statement algebra and the set algebra are particular cases of an abstract algebra called a Boolean algebra. This fact also explains why one could see similarities between the operators in the statement calculus and the operations of set theory. For all the identities listed in this section, we have also listed the corresponding equivalences from the statement calculus. Similar equivalences hold for the predicate calculus.

Not all the identities listed here are independent. Some of the identities can be proved by assuming certain other identities. However, we have listed these identities in order to include all those identities which exhibit some basic and useful properties. Most of these identities have been proved earlier in this section, and the others can be proved either by using the same technique or by using the identities already known to be true.

In our discussion here we assume that all the sets involved are subsets of a universal set E. Although such an assumption is not necessary for many of the identities, there is no loss of generality. Furthermore, some of the identities do require such an assumption, particularly those involving complementation.

Set Algebra	*Statement Algebra*	
	Idempotent laws	
$A \cup A = A$	$P \vee P \Leftrightarrow P$	(1)
$A \cap A = A$	$P \wedge P \Leftrightarrow P$	
	Associative laws	
$(A \cup B) \cup C = A \cup (B \cup C)$	$(P \vee Q) \vee R \Leftrightarrow P \vee (Q \vee R)$	(2)
$(A \cap B) \cap C = A \cap (B \cap C)$	$(P \wedge Q) \wedge R \Leftrightarrow P \wedge (Q \wedge R)$	
	Commutative laws	
$A \cup B = B \cup A$	$P \vee Q \Leftrightarrow Q \vee P$	(3)
$A \cap B = B \cap A$	$P \wedge Q \Leftrightarrow Q \wedge P$	
	Distributive laws	
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	$P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (Q \vee R)$	(4)
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$	$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$	(5)
$A \cup \varnothing = A$	$P \vee \mathbf{F} \Leftrightarrow P$	
$A \cap E = A$	$P \wedge \mathbf{T} \Leftrightarrow P$	(6)
$A \cup E = E$	$P \vee \mathbf{T} \Leftrightarrow \mathbf{T}$	
$A \cap \varnothing = \varnothing$	$P \wedge \mathbf{F} \Leftrightarrow \mathbf{F}$	(7)
$A \cup {\sim}A = E$	$P \vee \neg P \Leftrightarrow \mathbf{T}$	
$A \cap {\sim}A = \varnothing$	$P \wedge \neg P \Leftrightarrow \mathbf{F}$	
	Absorption laws	(8)
$A \cup (A \cap B) = A$	$P \vee (P \wedge Q) \Leftrightarrow P$	
$A \cap (A \cup B) = A$	$P \wedge (P \vee Q) \Leftrightarrow P$	
	De Morgan's laws	(9)
${\sim}(A \cup B) = {\sim}A \cap {\sim}B$	$\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$	
${\sim}(A \cap B) = {\sim}A \cup {\sim}B$	$\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$	(10)
${\sim}\varnothing = E$	$\neg \mathbf{F} \Leftrightarrow \mathbf{T}$	
${\sim}E = \varnothing$	$\neg \mathbf{T} \Leftrightarrow \mathbf{F}$	
${\sim}({\sim}A) = A$	$\neg\neg P \Leftrightarrow P$	

Scanned with OKEN Scanner

All the identities just given are presented in pairs except for the identity (11). This pairing is done because a principle of duality similar to the one given for statement algebra (see Sec. 1-2.10) also holds in the case of set algebra. In fact, the principle of duality holds for any Boolean algebra. At present it is sufficient to note that given any identity of the set algebra, one can obtain another identity by interchanging $\cup$ with $\cap$ and E with $\varnothing$.

Assuming identities (4) to (6), we shall prove the absorption laws. First note that

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

from the distributive and idempotent laws. Now

$$\begin{aligned} A \cup (A \cap B) &= (A \cap E) \cup (A \cap B) && \text{from (5)} \\ &= A \cap (E \cup B) && \text{from (4)} \\ &= A \cap E && \text{from (6)} \\ &= A && \text{from (5)} \end{aligned}$$

Alternatively one can prove it as follows. For any x,

$$\begin{aligned} x \in A \cup (A \cap B) &\Leftrightarrow x \in \{x \mid (x \in A) \vee ((x \in A) \wedge (x \in B))\} \\ &\Leftrightarrow x \in \{x \mid x \in A\} \\ &\Leftrightarrow x \in A \end{aligned}$$

using the absorption laws of predicate calculus.

In order to complete our discussion, we list some implications and certain set inclusions

$$(A \cup B \neq \varnothing) \Rightarrow (A \neq \varnothing) \vee (B \neq \varnothing) \tag{12}$$

$$(A \cap B \neq \varnothing) \Rightarrow (A \neq \varnothing) \wedge (B \neq \varnothing) \tag{13}$$

To prove the implication (12), let us assume that $A \neq \varnothing \vee B \neq \varnothing$ is *false*. This requires that $A \neq \varnothing$ is *false* and also that $B \neq \varnothing$ is *false*, that is, $A = B = \varnothing$. But then $A \cup B = \varnothing$, so that $A \cup B \neq \varnothing$ is also *false*. Hence the implication is proved. One could also have proved (12) by assuming that $A \cup B \neq \varnothing$ is *true* and showing that this assumption requires $A \neq \varnothing \vee B \neq \varnothing$ to be *true*. Implication (13) can be proved in a similar manner.

The following inclusions follow from the definition and have been proved earlier in this section.

$$A \cap B \subseteq A \qquad A \cap B \subseteq B \qquad A \subseteq A \cup B \qquad A - B \subseteq A \tag{14}$$

Let A be a family of indexed sets over an index set $\mathbf{I}$ such that $A = \{A_1, A_2, \ldots\} = \{A_i\}_{i \in \mathbf{I}}$. Then

$$\bigcup_{i \in \mathbf{I}} A_i = \{x \mid x \in A_i \text{ for some } i \in \mathbf{I}\} \tag{15}$$

$$\bigcap_{i \in \mathbf{I}} A_i = \{x \mid x \in A_i \text{ for every } i \in \mathbf{I}\} \tag{16}$$

The associative laws and the distributive laws can be generalized in the following manner.

$$B \cup (\bigcap_{i \in \mathbf{I}} A_i) = \bigcap_{i \in \mathbf{I}} (B \cup A_i) \tag{17}$$

$$B \cap (\bigcup_{i \in \mathbf{I}} A_i) = \bigcup_{i \in \mathbf{I}} (B \cap A_i)$$

The identities (17) can be proved using mathematical induction discussed later in Sec. 2-5.1. We now give some examples illustrating the above operations.

EXAMPLE 1 Verify the identities (17) for

$$A_1 = \{1, 5\} \qquad A_2 = \{1, 2, 4, 6\} \qquad A_3 = \{3, 4, 7\}$$

$$B = \{2, 4\} \quad \text{and} \quad I = \{1, 2, 3\}$$

SOLUTION

$$\bigcup_{i \in I} A_i = \{1, 2, 3, 4, 5, 6, 7\}$$

$$\bigcap_{i \in I} A_i = \varnothing$$

$$B \cup (\bigcap_{i \in I} A_i) = \{2, 4\}$$

$$\bigcap_{i \in I} (B \cup A_i) = \{1, 2, 4, 5\} \cap \{1, 2, 4, 6\} \cap \{2, 3, 4, 7\} = \{2, 4\}$$

$$B \cap (\bigcup_{i \in I} A_i) = \{2, 4\}$$

$$\bigcup_{i \in I} (B \cap A_i) = \varnothing \cup \{2, 4\} \cup \{4\} = \{2, 4\}$$

////

2-1.7 The Principle of Specification

The idea of a set was discussed in the beginning of this chapter, although we had used the notion earlier in Chap. 1 while discussing the universe of discourse or the domain of the object variable (see Sec. 1-5.5). The universe of discourse was defined as the set of all objects under consideration, and this set is the same as the universal set defined in Sec. 2-1.2.

A set is usually defined by means of a predicate. The connection between a predicate and a set defined by it is known as the *principle of specification*, which states that every predicate specifies a set which is a subset of a universal set. The subset specified by a predicate is called an *extension* of the predicate in the universal set. This method has been used extensively in defining sets. For example, if $P(x)$ is a predicate, then a set A is called an extension of $P(x)$ if

$$A = \{x \mid P(x)\}$$

A predicate can be considered as a condition, and any object of the universal set satisfying the condition is then an element of the set which is an extension of the predicate. Obviously, if two predicates are equivalent, then they have the same extension, and the two sets specified by equivalent predicates are

Scanned with OKEN Scanner

equal. In other words, if $P(x) \Leftrightarrow Q(x)$, then $A = B$ where A and B are the extensions of $P(x)$ and $Q(x)$, respectively. We now have an analogy between the equality of sets and the equivalence of predicates. A similar analogy exists between set inclusion and implication. If $P(x) \Rightarrow Q(x)$, then $A \subseteq B$ where, again, A and B are extensions of $P(x)$ and $Q(x)$ respectively.

If $P(x)$ is identically *true* for all x in E, then the extension of $P(x)$ in the universal set is the universal set itself. Similarly, if $P(x)$ is identically *false* for all x in E, then the extension of $P(x)$ in E is the null set. Recall that the universal set and the null set were defined as extensions of $P(x) \vee \neg P(x)$ and $P(x) \wedge \neg P(x)$ respectively. However, any other identically *true* (valid) and *false* predicates could have been used to define them.

If A and B are extensions of the predicates $P(x)$ and $Q(x)$, respectively, in a universal set E, then it is easy to see that $A \cup B$ and $A \cap B$ are the extensions of $P(x) \vee Q(x)$ and $P(x) \wedge Q(x)$ respectively. Similarly $\sim A$ is the extension of $\neg P(x)$. The extension of $P(x) \rightarrow Q(x)$ is the set $\sim A \cup B$, and that of $P(x) \rightleftarrows Q(x)$ is the set $(\sim A \cup B) \cap (A \cup \sim B)$. Thus the new sets formed from the sets A and B can be interpreted in terms of extensions of formulas containing $P(x)$ and $Q(x)$.

From the above discussion it is clear that all the identities of set theory given in the previous section should follow from the corresponding equivalence of predicate formulas. Similarly, the inclusions of sets should follow from the corresponding implications of predicates. If we replace the predicates by their extensions—$\wedge$ by $\cap$, $\vee$ by $\cup$, and $\neg$ by $\sim$—in any predicate formula, then we obtain the corresponding formula of set theory. Also, the equivalences and implications are replaced by equality and inclusions of sets. In fact, this technique has often been used in proving the identities and other relations of set theory so far. For example, let us consider

$$\neg(P(x) \vee Q(x)) \Leftrightarrow \neg P(x) \wedge \neg Q(x)$$

If A and B denote the extensions of $P(x)$ and $Q(x)$ respectively, then we can write

$$\sim(A \cup B) = \sim A \cap \sim B$$

Similarly, from

$$P(x) \vee (Q(x) \wedge R(x)) \Leftrightarrow (P(x) \vee Q(x)) \wedge (P(x) \vee R(x))$$

we get

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C) \qquad [C \text{ is the extension of } R(x)]$$

2-1.8 Ordered Pairs and n-tuples

So far we have been solely concerned with sets, their equality, and operations on sets to form new sets. We now introduce the notion of an ordered pair. Although it is possible to define ordered pairs rigorously, we shall give an intuitive definition.

An *ordered pair* consists of two objects in a given fixed order. Note that an ordered pair is not a set consisting of two elements. The ordering of the two objects is important. The two objects need not be distinct. We shall denote an ordered pair by $\langle x, y \rangle$. A familiar example of an ordered pair is the representation of a point in a two-dimensional plane in cartesian coordinates. Accordingly, the ordered pairs $\langle 1, 3 \rangle$, $\langle 2, 4 \rangle$, $\langle 1, 2 \rangle$, and $\langle 2, 1 \rangle$ represent different points in a plane.

The *equality* of two ordered pairs $\langle x, y \rangle$ and $\langle u, v \rangle$ is defined by

$$\langle x, y \rangle = \langle u, v \rangle \Leftrightarrow ((x = u) \wedge (y = v)) \tag{1}$$

so that $\langle 1, 2 \rangle \neq \langle 2, 1 \rangle$ and $\langle 1, 1 \rangle \neq \langle 2, 2 \rangle$. A distinction between ordered pairs and sets containing two elements will be clear from the following examples:

$$\{a, b\} = \{b, a\} = \{a, a, b\} \qquad \{a, a\} = \{a\} \qquad \langle a, b \rangle \neq \langle b, a \rangle \qquad \langle a, a \rangle \neq \{a\}$$

The idea of an ordered pair can be extended to define an ordered triple, and, more generally, an n-tuple.

An *ordered triple* is an ordered pair whose first member is itself an ordered pair. Thus an ordered triple can be written as $\langle \langle x, y \rangle, z \rangle$. From the definition of the equality of an ordered pair, we can arrive at the equality of ordered triples $\langle \langle x, y \rangle, z \rangle$ and $\langle \langle u, v \rangle, w \rangle$:

$$\langle \langle x, y \rangle, z \rangle = \langle \langle u, v \rangle, w \rangle \qquad \text{iff} \qquad \langle x, y \rangle = \langle u, v \rangle \wedge z = w$$

But, $\langle x, y \rangle = \langle u, v \rangle$ if $(x = u \wedge y = v)$. Therefore

$$\langle \langle x, y \rangle, z \rangle = \langle \langle u, v \rangle, w \rangle \Leftrightarrow ((x = u) \wedge (y = v) \wedge (z = w)) \tag{2}$$

From the above definition of equality of an ordered triple, we may write an ordered triple as $\langle x, y, z \rangle$ with an understanding that $\langle x, y, z \rangle$ stands for $\langle \langle x, y \rangle, z \rangle$. Note that

$$\langle x, y, z \rangle \neq \langle y, x, z \rangle \neq \langle x, z, y \rangle$$

An ordered quadruple can be defined as an ordered pair whose first member is an ordered triple. Thus, an ordered quadruple is written as $\langle \langle x, y, z \rangle, u \rangle$ which is actually $\langle \langle \langle x, y \rangle, z \rangle, u \rangle$. It is easy to show that two ordered quadruples $\langle \langle x, y, z \rangle, u \rangle$ and $\langle \langle p, q, r \rangle, s \rangle$ are equal provided that

$$(x = p) \wedge (y = q) \wedge (z = r) \wedge (u = s) \tag{3}$$

In view of this fact, we shall write an ordered quadruple as $\langle x, y, z, u \rangle$.

Continuing this process, an ordered n-tuple is defined to be an ordered pair whose first member is an ordered $(n - 1)$-tuple. We write an ordered n-tuple as $\langle \langle x_1, x_2, \ldots, x_{n-1} \rangle, x_n \rangle$. Further, given two ordered n-tuples $\langle \langle x_1, x_2, \ldots, x_{n-1} \rangle, x_n \rangle$ and $\langle \langle u_1, u_2, \ldots, u_{n-1} \rangle, u_n \rangle$, we have

$$\langle \langle x_1, x_2, \ldots, x_{n-1} \rangle, x_n \rangle = \langle \langle u_1, u_2, \ldots, u_{n-1} \rangle, u_n \rangle$$
$$\Leftrightarrow ((x_1 = u_1) \wedge (x_2 = u_2) \wedge \cdots \wedge (x_n = u_n))$$

Therefore, an ordered n-tuple will be written as $\langle x_1, x_2, \ldots, x_n \rangle$.

2-1.9 Cartesian Products

Definition 2-1.15 Let A and B be any two sets. The set of all ordered pairs such that the first member of the ordered pair is an element of A and the second member is an element of B is called the *cartesian product* of A

Scanned with OKEN Scanner

and B and is written as $A \times B$. Accordingly,

$$A \times B = \{\langle x, y\rangle \mid (x \in A) \wedge (y \in B)\}$$

EXAMPLE 1 If $A = \{\alpha, \beta\}$ and $B = \{1, 2, 3\}$, what are $A \times B$, $B \times A$, $A \times A$, $B \times B$, and $(A \times B) \cap (B \times A)$?

SOLUTION

$$A \times B = \{\langle \alpha, 1\rangle, \langle \alpha, 2\rangle, \langle \alpha, 3\rangle, \langle \beta, 1\rangle, \langle \beta, 2\rangle, \langle \beta, 3\rangle\}$$
$$B \times A = \{\langle 1, \alpha\rangle, \langle 2, \alpha\rangle, \langle 3, \alpha\rangle, \langle 1, \beta\rangle, \langle 2, \beta\rangle, \langle 3, \beta\rangle\}$$
$$A \times A = \{\langle \alpha, \alpha\rangle, \langle \alpha, \beta\rangle, \langle \beta, \alpha\rangle, \langle \beta, \beta\rangle\}$$
$$B \times B = \{\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle, \langle 2, 3\rangle, \langle 3, 1\rangle, \langle 3, 2\rangle, \langle 3. 3\rangle\}$$
$$(A \times B) \cap (B \times A) = \varnothing \qquad ////$$

EXAMPLE 2 If $A = \varnothing$ and $B = \{1, 2, 3\}$ what are $A \times B$ and $B \times A$?

SOLUTION

$$A \times B = \varnothing = B \times A \qquad ////$$

Before we consider the cartesian product of more than two sets let us consider the expressions $(A \times B) \times C$ and $A \times (B \times C)$. From the definition it follows that

$$(A \times B) \times C = \{\langle\langle a, b\rangle, c\rangle \mid (\langle a, b\rangle \in A \times B) \wedge (c \in C)\}$$
$$= \{\langle a, b, c\rangle \mid (a \in A) \wedge (b \in B) \wedge (c \in C)\} \qquad (1)$$

The last step follows from our definition of the ordered triple given in Sec. 2-1.8. Next,

$$A \times (B \times C) = \{\langle a, \langle b, c\rangle\rangle \mid (a \in A) \wedge (\langle b, c\rangle \in B \times C)\}$$

Here $\langle a, \langle b, c\rangle\rangle$ is not an ordered triple. If we consider $(A \times B) \times C$ as an ordered pair, then the first member is an ordered pair and the second member is an element of C. On the other hand, $A \times (B \times C)$ is an ordered pair in which the first member is an element of A while the second member is an ordered pair. This fact shows that

$$(A \times B) \times C \neq A \times (B \times C)$$

Before defining the cartesian product of any finite number of sets, we shall show that the cartesian product satisfies the following distributive properties. For any three sets A, B, and C

$$A \times (B \cup C) = (A \times B) \cup (A \times C)$$
$$A \times (B \cap C) = (A \times B) \cap (A \times C) \qquad (2)$$

We now prove the first of these two identities.

$$\begin{aligned} A \times (B \cup C) &= \{\langle x\ y\rangle \mid (x \in A) \wedge (y \in B \cup C)\} \\ &= \{\langle x, y\rangle \mid (x \in A) \wedge ((y \in B) \vee (y \in C))\} \\ &= \{\langle x, y\rangle \mid ((x \in A) \wedge (y \in B)) \vee ((x \in A) \wedge (y \in C))\} \\ &= (A \times B) \cup (A \times C) \end{aligned}$$

The second equality in Eq. (2) can be proved in a similar manner.

Let $A = \{A_i\}_{i \in \mathbf{I}_n}$ be an indexed set and $\mathbf{I}_n = \{1, 2, \ldots, n\}$. We denote the cartesian product of the sets $A_1, A_2, \ldots, A_n$ by

$$\bigtimes_{i \in \mathbf{I}_n} A_i = A_1 \times A_2 \times \cdots \times A_n$$

which is defined by

$$\bigtimes_{i \in \mathbf{I}_1} A_i = A_1 \quad \text{and} \quad \bigtimes_{i \in \mathbf{I}_m} A_i = \left(\bigtimes_{i \in \mathbf{I}_{m-1}} A_i\right) \times A_m \quad \text{for } m = 2, 3, \ldots, n$$

According to the above definition,

$$A_1 \times A_2 \times A_3 = (A_1 \times A_2) \times A_3$$

and

$$\begin{aligned} A_1 \times A_2 \times A_3 \times A_4 &= (A_1 \times A_2 \times A_3) \times A_4 \\ &= ((A_1 \times A_2) \times A_3) \times A_4 \end{aligned}$$

Our definition of cartesian product of n sets is related to the definition of n-tuples in the sense that

$$A_1 \times A_2 \times \ldots \times A_n = \{\langle x_1, x_2, \ldots, x_n\rangle \mid (x_1 \in A_1) \wedge (x_2 \in A_2) \wedge \cdots \wedge (x_n \in A_n)\}$$

The cartesian product $A \times A$ is also written as A^2, and similarly $A \times A \times A$ as A^3, and so on.

EXERCISES 2-1

1 Give examples of sets A, B, C such that $A \cup B = A \cup C$, but $B \neq C$.

2 Write the sets

$$\varnothing \cap \{\varnothing\} \qquad \{\varnothing\} \cap \{\varnothing\} \qquad \{\varnothing, \{\varnothing\}\} - \varnothing$$

3 Write the members of $\{a, b\} \times \{1, 2, 3\}$.

4 Write $A \times B \times C$, B^2, A^3, $B^2 \times A$, and $A \times B$ where $A = \{1\}$, $B = \{a, b\}$, and $C = \{2, 3\}$.

5 Show by means of an example that $A \times B \neq B \times A$ and $(A \times B) \times C \neq A \times (B \times C)$.

6 Show that for any two sets A and B

$$\rho(A) \cup \rho(B) \subseteq \rho(A \cup B)$$
$$\rho(A) \cap \rho(B) = \rho(A \cap B)$$

Show by means of an example that

$$\rho(A) \cup \rho(B) \neq \rho(A \cup B)$$

Scanned with OKEN Scanner

7 Prove the identities

$$A \cap A = A \qquad A \cap \varnothing = \varnothing \qquad A \cap E = A \qquad \text{and } A \cup E = E$$

8 Show that $A \times (B \cap C) = (A \times B) \cap (A \times C)$.

9 Prove that

$$(A \cap B) \cup (A \cap {\sim}B) = A$$

and

$$A \cap ({\sim}A \cup B) = A \cap B$$

10 Show that $A \times B = B \times A \Leftrightarrow (A = \varnothing) \vee (B = \varnothing) \vee (A = B)$.

11 Show that $(A \cap B) \cup C = A \cap (B \cup C)$ iff $C \subseteq A$.

12 Draw Venn diagrams showing

$$A \cup B \subset A \cup C \quad \text{but} \quad B \not\subset C$$
$$A \cap B \subset A \cap C \quad \text{but} \quad B \not\subset C$$
$$A \cup B = A \cup C \quad \text{but} \quad B \neq C$$
$$A \cap B = A \cap C \quad \text{but} \quad B \neq C$$

13 Draw Venn diagrams and show the sets

$${\sim}B \qquad {\sim}(A \cup B) \qquad B - ({\sim}A) \qquad {\sim}A \cup B \qquad {\sim}A \cap B$$

where $A \cap B \neq \varnothing$.

14 Show that $(A + B) + C = A + (B + C)$.

15 Prove that $A + A = \varnothing$ and $A + \varnothing = A$.

16 Show that $(A - B) - C = (A - C) - (B - C)$.

17 Prove that $(A \cap B) \times (C \cap D) = (A \times C) \cap (B \times D)$.

2-2 REPRESENTATION OF DISCRETE STRUCTURES

A number of applications involving discrete mathematical structures will be discussed throughout this book. Under discrete mathematical structures we include sets, ordered sets, and other structures such as trees and graphs which will be discussed in Chap. 5. These applications require that discrete structures be represented in some suitable manner. This will be the topic of the present section.

2-2.1 Data Structures

If the representation for a discrete structure does not exist in the programming language being used, then the program for a particular algorithm may be quite complex. For example, in a payroll (data processing) application a treelike representation of information for an employee such as in Fig. 2-2.1 may be required. This structure does not exist in certain programming languages, such as FORTRAN. It does not mean that we cannot program a payroll application in FORTRAN. It could be done by writing programs to construct and manipulate trees, but the programs would be complex. It would be more suitable to use a data processing language such as COBOL in this case. Ideally, the programming language chosen for the implementation of an algorithm should possess the particular representations chosen for the discrete structures in the problem being

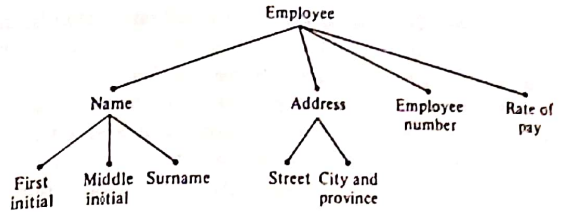


FIGURE 2-2.1 A treelike representation of information.

solved. In practice, the choice of a language may be dictated by what is available at a particular computing center, what language is preferred by some key personnel, etc.

The task of writing a computer program for a problem is made simpler if the problem can be analyzed in terms of subproblems. This structuring process in problem solving is usually reflected in the program. The program tends to be modular or consists of a number of small parts, an approach to problem solving and organization that has had a profound impact on the design of many programming languages. Processes, or modules, concerned with operations performed on data structures are frequently represented by subroutines or functions, and for the program implementation of any significant problem the organization of a program into suitable modules or subroutines is indispensible. Indeed, the simplest way to write programs is to organize them in such a modular fashion.

In organizing the solution to a problem, one is concerned with two classes of concepts. The first class deals with particular *data structures*, which are the result of an individual's interpretation of a problem, expressed by the preparation and recording of data which must be manipulated in some way according to certain processes. The second class of concepts deals with the operations that must be performed on such data structures. The terms "data structure" and "discrete structure" are considered to be equivalent in our discussions.

The class of concepts dealing with data structures has become increasingly important in recent years. Initially, computers were used to solve primarily numerical scientific problems, but this situation has changed drastically with the emerging interest in many nonnumerical problems. Associated with the solution of numerical problems were rather primitive data structures such as variables, vectors, and arrays. These structures were, for most cases, adequate for the solution of numerical problems. However, in the solution of nonnumerical problems these primitive data structures were clearly not sufficiently powerful to specify the more complex structural relationships in the data.

Data in a particular problem consist of a set of elementary items or atoms of data. An atom usually consists of single elements such as integers, bits, characters, or a set of such items. A person solving a particular problem is concerned with establishing certain paths of accessibility between atoms of data. The choice of atoms of data is a necessary and key step in defining and, therefore, solving a problem. The possible ways in which the data items or atoms are structured define different *data structures*. By choosing a particular structure for the data items, certain items become immediate neighbors while others are related in a

Scanned with OKEN Scanner

weaker way. The interpretation of two items being immediate neighbors is that of adjacency relative to the ordering relation that may be imposed by the structure. Ordering relations are discussed in Sec. 2-3.9.

Let us now consider a general classification of data structures. In so doing, we will introduce well-known data structures and a number of associated operations which are used to manipulate them.

A frequently used data structure operation is one which *creates* a structure. This creation is usually accomplished in a computer program by using a declaration statement. Data structures can be *destroyed* or *erased*. Many programming languages do not permit the destruction of data structures once they have been created since all creations are performed at compile time. Another operation used in conjunction with data structures is one which changes data in the structure. This operation is called *updating* and can change, add, or delete an element in a structure.

Data can be classified as *batches*, *vectors*, and *plexes*. A *batch* is an unordered set of objects which can be of fixed or variable size. The *size* of a structure here is defined to be its number of data items. Batches are very frequently used in the solution of data processing problems. A *vector* is an ordered set which consists of a fixed number of objects. No deletion or addition operations are performed on vectors. At best, certain elements are changed to some value which we have previously decided to use as representing an element to be ignored. The setting of an element in an array to zero to "delete" it is an example. A *plex*, on the other hand, is an ordered set consisting of a variable number of elements to which additions and deletions can be made.

A plex which displays the relationship of adjacency between elements is said to be *linear*. Any other plex is said to be *nonlinear*. The term "list" is also used in the literature to refer to what we have called a plex, and the term "linear list" is used instead of linear plex. We shall concern ourselves with linear plexes in the remainder of this section.

Operations performed on plexes include those which are performed on vectors. However, there is one important difference in that the size of a plex may be changed by updating. Indeed, updating may add or delete elements as well as change existing elements. The addition and deletion of elements in a plex are specified by position, so that we may want to delete the ith element of a plex or add a new element before or after the ith existing element. Frequently, it may be required to add or delete an element whose position in a plex is based on the value of some element (as in sorting). It may be required to add or delete a given element to or from a plex respectively preceding or following an element that has a specified value or satisfies a particular relationship.

Certain plexes permit the deletion of any of its elements and the addition of an element in any position. An important subclass of plexes permits the addition or deletion of an element to occur only at one end. A plex belonging to this subclass is called a *stack*. The addition operation is referred to as "push," the deletion operation as "pop." The most and least accessible elements in a stack are known as the *top* and *bottom* of the stack respectively. Since addition and deletion operations are performed at the same end of a stack, the elements can be removed only in the opposite order from that in which they were added to the stack. This phenomenon is observed in conjunction with recursive functions in

Sec. 2-6.2, and such a plex is frequently referred to as a LIFO (last in, first out) plex.

A common example of a stack phenomenon, which permits the selection of only its end element, is a pile of trays in a cafeteria. These are supported by some kind of spring action in such a manner that a person desiring a tray finds that only one is available at a time at the surface of the tray counter. The removal of the top tray causes the load on the spring to be lighter, and the next tray to appear at the surface of the counter. A tray which is placed on the pile causes the entire pile to be pushed down and that tray to appear above the tray counter.

Another important subclass of plexes permits deletions to be performed at one end of a plex and additions at the other. The information in such a plex is processed in the same order that it was received, that is, on a first in, first out (FIFO) basis. This type of plex is frequently referred to as a *queue*. The updater may be restricted to the examination of elements at only one end in the case of a queue. If no such restriction is made, any element in a plex can be selected. A familiar example of a queue is a checkout line at a supermarket cash register. The first person in line is (usually) the first to be checked out.

A similar type of restriction may be specified for changing elements in a plex. It may be required to examine the top element in a stack or the front element of a queue with a view to altering (not adding or deleting) it. This operation is sometimes extended to other elements of the structure (besides the top or front elements of a stack or queue respectively).

We shall deal with examples of nonlinear plexes in subsequent chapters. The following section is concerned with the representation of data structures in the memory of a computer.

2-2.2 Storage Structures

It is very important in solving a problem to distinguish explicitly between data structures, on the one hand, and the ways by which these structures are represented in the memory of a particular computer dictated by a specific hardware and software system, on the other hand. The way in which a particular data structure is represented in the memory of a computer is known as a *storage structure*. The distinction between a data structure and its corresponding storage structure is often confused. This leads to a loss of efficiency, and prevents the problem solver from making optimal use of available tools and resources. There are many possible memory configurations or storage structures corresponding to a particular data structure. For example, there are a number of possible storage structures for a data structure such as an array. It is also possible for two data structures to be represented by the same storage structure. In many instances, almost exclusive attention is directed toward the storage structures for certain given data, and little attention is given to the data structure per se. In essence, there is significant confusion between the properties that belong to the interpretation and meaning of the data on the one hand and the storage structures that can be selected to represent them in a programming system on the other.

In discussing storage structures we will be primarily concerned with the core memory of the conventional digital computer. The main memory of such a

Scanned with OKEN Scanner

computer is organized into an ordered sequence of words. Each word contains from 8 to 64 bits, and its contents can be referenced by using an address. For efficiency reasons, it is desirable to arrange data in such a manner that a particular element of the data can be referenced by computing its address rather than by searching for it. It is therefore preferable to have a memory organization which can be addressed by location.

There are two possible ways that can be used to obtain an address of an element.

The first method of obtaining an address is by using the description of the data being sought. This type of address is known as a *computed address*. This method of obtaining an address is used very extensively in programming languages to compute the address of an element of an array and in the acquisition of the next instruction to be executed in the object program.

The second method of obtaining an address is to store it somewhere in the memory of the computer. This type of address is referred to as a *link* or *pointer address*. In programming languages the addresses of the actual arguments of a procedure are stored in the computer memory. The return address which is used by a procedure to return to the calling program is also stored, not computed. Certain structures require a combination of computed and link addresses.

2-2.3 Sequential Allocation

Integers, real numbers, and character strings are considered to be primitive data structures because the instruction repertoire of a computer has instructions which will manipulate these primitive structures. We can perform the common arithmetic operations on numbers. A word which contains a character string can be modified by using a number of machine language instructions.

We now turn to the representation in storage of more complex data structures. Actually a complex (nonprimitive) data structure can be considered to consist of a structured set of primitive data structures. A vector may consist of an ordered set of integers, or a plex may consist of a set of elements, each element (node) consisting of two basic fields such as an integer and a real number.

The most straightforward way of representing a vector or a plex in the memory of a computer is to store their elements or nodes one after the other. A vector will have its elements adjacent to one another in memory. This method of allocating memory to a structure is called *sequential allocation*.

One of the simplest data structures which makes use of computed addresses to locate its elements is the vector. Normally a sequence of (contiguous) memory locations are sequentially allocated to the vector. Assuming that each element requires one word of memory, an n-element vector will occupy n consecutive words in memory. The size of a vector is fixed and therefore requires a fixed number of memory locations. In general, a vector A can be represented as in Fig. 2-2.2 where L_0 is the address of the first word allocated to the first element of A and where m represents the number of words allocated to each element.

In certain programming languages, memory allocation is performed at compile time where the size of the vector obtained in the dimensioning statement is saved along with the starting address L_0.

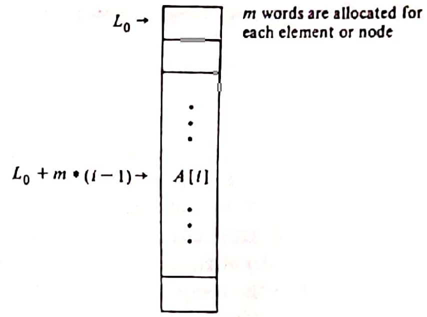


FIGURE 2-2.2 Representation of a vector.

A multidimensional array can be represented by an equivalent one-dimensional array. For example, a two-dimensional array consisting of three rows and columns could be stored sequentially by columns as

$$A[1,1]A[2,1]A[3,1]A[1,2]A[2,2]A[3,2]A[1,3]A[2,3]A[3\ 3]$$

↑ L_0

The address of element $A[i, j]$ can be obtained by evaluating the expression

$$L_0 + (j-1) * 3 + i - 1$$

For element $A[2, 3]$, the address is given as $L_0 + 7$. In general, for a two-dimensional array consisting of n rows and m columns, which is stored by *columns*, the address of element $A[i, j]$ is given by the linear expression

$$L_0 + (j-1) * n + (i-1)$$

In many programming languages a two-dimensional array will be stored row by row (sometimes referred to as row major order) instead of column by column (column major order). An array consisting of n rows and m columns will be stored sequentially as

$$A[1,1]A[1,2]\cdots A[1,m]A[2,1]A[2,2]\cdots A[2,m]\cdots A[n,1]A[n,2]\cdots A[n,m]$$

The address of matrix element $A[i, j]$ is given by the expression

$$L_0 + (i-1) * m + (j-1)$$

In discussions of sequential allocation, arrays are one of the first data structures that come to mind. The adjacency of primitive elements, however, may be described at the lowest possible level of machine organization. Sequences of bits, or bit strings, are often allocated storage sequentially. For example, in a computer with a word size of 32 bits and 8-bit bytes, if a bit string is to represent n bits, then $\lceil n/8 \rceil$ (the least integer greater than or equal to $n/8$) consecutive bytes are allocated as its storage structure. Not only are these bytes adjacent, but we also describe the bits as being arranged in a sequential manner. An important application of bit strings to set theory is the subject of Sec. 2-2.5.

Scanned with OKEN Scanner

2-2.4 Pointers and Linked Allocation

The previous section discussed at some length how the address of an element in a data structure could be obtained by direct computation. The data structures discussed were linearly ordered, and this ordering relation was specified in the corresponding storage structures by using sequential allocation. There was no need for an element to specify where the next element would be found.

Consider a list consisting of elements which individually vary in size. The task of directly computing the address of a particular element becomes much more difficult. An obvious method of obtaining the address of a node (element) is to store it in the computer memory. This address was previously defined as a link or pointer address. If the list in question has n nodes, we can store the address of each node in a vector consisting of n elements. The first element of the vector will contain the address of the first node of the list, the second element the address of the second node, and so on.

There are many applications which, by their very nature, have data which are continually being updated (additions, deletions, etc.). Each time a change occurs, significant manipulation of the data is required. The representation of the data by sequentially allocated lists in some of these cases results in an inefficient use of memory and wasted computational time, and, indeed, for certain problems this method of allocation is totally unacceptable.

The interpretation of a pointer as an address is a natural one. Most computers use addresses to find the next instruction to be executed and its operand(s). In many hardware configurations special registers are used to store such addresses.

A pointer can be regarded as a general type of structure because when a pointer to a data structure is given, then its contents become accessible. Pointers are always of the same length (usually no longer than a half-word), and this property enables the manipulation of pointers to be performed in a uniform manner by using simple allocation techniques regardless of the configurations of the structures they may point to.

In the sequential-allocation method one is able to compute an address of an element provided that the storage structure is organized in some uniform manner. Pointers permit the referencing of structures in a uniform way regardless of the organization of the structure being referenced. Pointers are capable of representing a much more complex relationship between elements of a structure than a linear order.

The use of pointers or links to refer to elements of a data structure (which is linearly ordered) implies that elements which are adjacent because of the linear ordering need not be physically adjacent in memory. This type of allocation scheme is called *linked allocation*. We now turn to the problem of representing structures by this type of allocation.

A plex has been defined to consist of an ordered set of elements which may vary in number. The previous subsection was concerned with representation of the relationship of adjacency between elements in a plex. There are many other structures that can be represented by a plex where the relationships that exist between elements are much more complex than that of adjacency.

The simplest form that can be used to represent a linear plex is to expand each node to contain a link or pointer to the next node. This representation will be called a *one-way chain* or *singly linked linear list*, which can be displayed as in Fig. 2-2.3*a* in which the variable *FIRST* contains an address or pointer which gives the location of the first node of the list. Each node is divided into two parts. The first part represents the original information contents of the element, and the second part contains the address of the next node. The last node does not have a successor, and, consequently, no actual address is stored in the pointer field. In such a case, a null value is stored as the address. The "arrow" emanating from the link field of a particular node indicates its successor node in the structure. For example, the linked list in Fig. 2-2.3*b* represents a five-node list whose elements are located in memory locations 1000, 1002, 1010, 1007, and 1005 respectively. We again emphasize that the only purpose of the links is to specify which node is next in the linear ordering. The link address of *NULL* (indicated by the slash) in the last node signals the end of the list. *NULL* is not an address of any possible node, but is a special value which cannot be mistaken for an address. For this reason, *NULL* is used as a special list delimiter. It is possible for a list to have no nodes at all. Such a list is called an empty list and is denoted by assigning a value of *NULL* to *FIRST* in our example.

Let us compare the operations commonly performed on sequentially allocated and linked lists. Consider the operations of insertion and deletion in the case of a sequentially allocated list. If we have a five-element list and it is required to insert a new element between the first and second elements, then the last four elements of the list must be moved so as to make room for the new element. For a list that contains many nodes, this way of performing an insertion is rather inefficient especially if many insertions are to be performed. The principle applies in the case of a deletion where all elements after the element being deleted must be moved up so as to take up the vacant space caused by the element removed from the list.

In the case of linked allocation, an addition is performed in a straightfor-

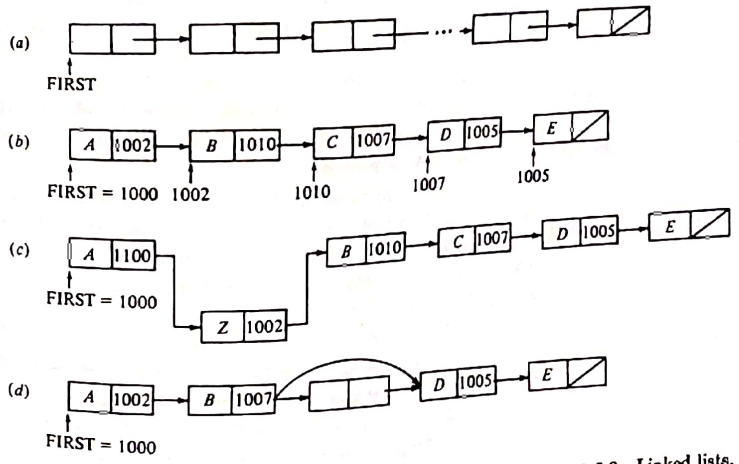


FIGURE 2-2.3 Linked lists.

ward manner. If a new element is to be inserted following the first element, in the preceding five-element linked list this insertion can be accomplished by merely interchanging pointers, as shown in Fig. 2-2.3c. The deletion of the third element from the original list can be performed in a similar fashion, as shown in Fig. 2-2.3d. It is clear that the insertion and deletion operations are more efficient when performed on linked lists than when performed on sequentially allocated lists.

If a particular node in a linked list is required, one has to follow the links from the first node onward until it is found. This method is clearly inferior to the computed-address technique associated with sequential allocation.

If, in a particular application, it is required to examine every node in the list, then it is only slightly more time-consuming to go through a linked list than a sequential list.

It is easier to join or split two linked lists than it is in the case of sequential allocation. This task can be accomplished merely by interchanging pointers and does not require the movement of nodes.

The pointers or links consume additional memory, but if only part of a memory word is being used, then a pointer can be stored in the remaining part. It is possible to group nodes so as to require only one link per several nodes. The cost of storing pointers may not be too expensive in many cases.

Since the memory address number in the link field has been used for illustration purposes in the present discussion and is of no concern (and indeed unknown) to the programmer in practice, the arrow symbol will be used exclusively to denote a successor node.

We mentioned the use of pointers in this section only to specify the linear ordering (adjacency) among elements, but pointers can be used to specify more complex relations between nodes such as that of a tree or a directed graph. This relation is difficult and indeed for certain graphs impossible to specify by using sequential allocation. These more complex structures can be specified by placing in the nodes a number of pointers. It is possible for a particular node to belong to several lists with this technique. A further discussion of this topic is given in Chap. 5.

From this discussion and some of the applications which follow, it will become clear that for certain operations linked allocation is more efficient than sequential allocation, and yet for other operations the opposite holds. In a typical application, both types of allocations are used.

Consider the familiar symbol manipulation problem of performing various operations on polynomials such as addition, subtraction, multiplication, division, differentiation, etc. Let us direct our attention, in particular, to the manipulation of polynomials in two variables. It may be required, for example, to write a program which subtracts polynomial $x^2 + 3xy + y^2 + y - x$ from polynomial $2x^2 + 5xy + y^2$ to give a result of $x^2 + 2xy - y + x$. We are interested in finding a suitable representation for polynomials so that the operations just mentioned can be performed in a reasonably efficient manner. If we are to manipulate polynomials, it is clear that individual terms must be selected. In particular, we must distinguish between variables, coefficients, and exponents within each term.

It is possible to represent a polynomial as a character string and solve the problem by searching for the various parts of a term. This approach tends to be

complex, especially if one tries to program in a language with only primitive string processing capabilities.

A two-dimensional array can be used to represent a polynomial in two variables. In a programming language that permits subscripts to have zero values, the coefficients of the term x^iy^j would be stored in the element identified by row i and column j of the array (assuming the existence of a zeroth row and zeroth column). If we restrict the size of an array to a maximum of 5 rows and 5 columns, then the powers of x and y in any term of the polynomial must not exceed a value of 4. The array representing the polynomial $2x^2 + 5xy + y^2$ is given as

$$\begin{matrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$$

and the array for $x^2 + 3xy + y^2 + y - x$ is

$$\begin{matrix} 0 & 1 & 1 & 0 & 0 \\ -1 & 3 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Once we have an algorithm for converting the input data to an array representing a polynomial and another for converting an array to an appropriate output form, then addition and subtraction of polynomials reduce to the adding and subtracting of corresponding elements in the two arrays respectively. This problem is left as an exercise. A number of disadvantages are evident in using this representation. In the first case, the array tends to be sparsely filled with nonzero elements. Second, the powers of the polynomials are restricted. Furthermore, the memory requirements associated with certain operations were not always predictable, as in the case of polynomial division, and so one was faced with the situation of not knowing how much memory to reserve for the polynomial generated by such an operation. This reason is why linked allocation should be used in this case. Before describing how it can be used, we will consider the different classes of operations we can perform on linear linked lists.

Data in any type of application are required to be manipulated according to certain operations. If this processing is to be performed by the use of a computer, the first task to be accomplished is the adequate representation of the data in the computer memory. The difficulty and complexity of this task depend to a large extent on the particular programming language that is used to program the algorithms associated with an application. Certain languages have been specifically designed for manipulating linked lists. One such prominent language is LISP 1.5. In other cases, common procedures or functions to be performed on linked lists have been written as subprograms in a simple "host" language. An example of such a case is the list processing language SLIP which consists of

Scanned with OKEN Scanner

a number of subprograms which are written in the FORTRAN language. We will make use of the PL/I language to represent certain common operations performed on lists.

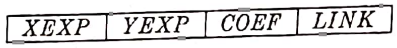
There are a number of classes of operations which are associated with linked lists. The first class of such operations is independent of the data contained in the nodes of a list. These operations include the insertion, deletion, and selection of nodes. Programming languages which possess list processing capabilities usually have these operations built in.

Another class of operations associated with list structures is the operation which converts the raw data from a human-readable form to a corresponding machine form. The inverse operation of converting an internal structure to a suitable human-readable form is also required. These operations are clearly data-dependent, and attention must be given to the interpretation that is associated with the structures. List processing languages will have some standard basic routines for such operations, but any additional routines must be programmed.

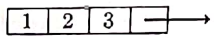
Finally, there are operations that must be programmed to manipulate the data according to what is required in a particular application at hand. In the case of polynomial manipulation, for example, such operations would include the addition, subtraction, multiplication, division, differentiation, and integration of polynomials. Once a programmer has access to all the routines for the three classes mentioned above, the task of programming an algorithm is much simpler.

Let us consider the problem of describing singly linked linear list representations and operations in an algorithmic notation and also in the PL/I language. A node consists of a number of fields, each of which can represent an integer, a real number, etc., except for one field (usually the last for purposes of illustration), called a pointer, which contains the location of the next node in the list. This location is specified as a computer memory address.

Consider the example of representing a term of a polynomial in the variables x and y. A typical node will be represented as



which consists of four sequentially allocated fields that we will collectively refer to as *TERM*. The first two fields represent the power of the variables x and y respectively. The third and fourth fields represent the coefficient of the term in the polynomial and the address of the next term in the polynomial, respectively. For example, the term $3xy^2$ would be represented as

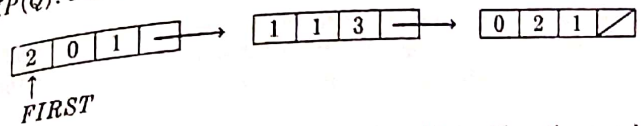


The selection of a particular field within a node for the example of our polynomial is an easy matter. Our algorithmic notation will allow the referencing of any field of a node, given the pointer P to that node. $COEF(P)$ denotes the coefficient field of a node pointed to by P. Similarly, the exponents of x and y are given by $XEXP(P)$ and $YEXP(P)$ respectively, and the pointer to the next node is given by $LINK(P)$.

Consider as an example the representation of the polynomial

$$x^2 + 3xy + y^3$$

as a linked list. Assume that the nodes in the list are to be stored so that a term pointed to by P will precede another term indicated by Q if $XEXP(P)$ is greater than $XEXP(Q)$; or, if they are equal, then $YEXP(P)$ must be greater than $YEXP(Q)$. For our example the list can be represented as



Let us now consider the more difficult problem of inserting a node into a linked list. There are a number of steps necessary to accomplish this task. First, the values for the various fields of the new node must be obtained either from an input operation or as the result of a computation. Second, we must somehow obtain a node from available storage. Finally, the values of the fields obtained in the first step are copied in the appropriate field positions of the new node, which is then placed in the linked list. The linking of the new node to its successor in the existing list is accomplished by setting the pointer field of the former to a value that gives the location of the latter.

In our algorithms we will assume that we have an available area of storage and that we can request a node from this area. In our polynomial example, the assignment statement

$$P \leftarrow TERM$$

creates a new node consisting of the four fields previously described, with the location of the first field being copied in the pointer variable P. At this point, the fields $XEXP(P)$, $YEXP(P)$, $COEF(P)$, and $LINK(P)$ have undefined values. When a node is no longer required, we can return it to available storage. This return will be specified simply by stating, for example,

Restore node P to the availability area.

After this action is taken, the node is assumed to be inaccessible, and P is undefined.

We can now formulate an algorithm which will insert a term of a polynomial into a linked list.

Algorithm *INSERT* Given the definition of the node structure *TERM* and an availability area from which we can obtain such nodes, it is required to insert in the linked list a node which will immediately precede the node whose address is designated by the pointer *FIRST*. The fields of the new term have exponents for x and y and a coefficient value represented by the variables NX, NY, and $NCOEF$ respectively. P is an auxiliary pointer variable.

1 [Obtain a node from available storage.] $P \leftarrow TERM$.

2 [Initialize numeric fields.] Set $XEXP(P) \leftarrow NX$, $YEXP(P) \leftarrow NY$, and $COEF(P) \leftarrow NCOEF$.

Scanned with OKEN Scanner

3 [Set link to the list.] Set $LINK(P) \leftarrow FIRST$, $INSERT \leftarrow P$, and Exit.

Algorithm *INSERT* performs all its insertions at one end of the linked list. In general, it is also possible to perform insertions at the other end or in the middle of the list. The zero polynomial (polynomial with no terms) is represented by the null pointer. Before any term of a polynomial is added to a list, its first node pointer, which we will call *POLY*, has a value of *NULL*. It is quite possible in adding and subtracting polynomials to get a cancellation of terms, which will result in a zero polynomial.

The algorithm will be invoked as a function whose name is *INSERT*. The address of the created node is assigned to *INSERT* immediately prior to the exit, and it is this value that will replace the function call. ////

In algorithm *INSERT* we assume the existence of the required structure, *TERM*. In PL/I programs, however, we must define any such datatype. We now turn to the programming aspects of how this is accomplished. For linked list processing applications it is necessary to use a structure declared to be of the BASED storage class. The declaration statement

```
DECLARE
    1 TERM BASED(P),
      2 XEXP BINARY FIXED,
      2 YEXP BINARY FIXED,
      2 COEF BINARY FLOAT,
      2 LINK POINTER;
```

defines the structure of a polynomial term. The node TERM, at level 1, collectively represents the four fields at level 2. The fields XEXP, YEXP, and COEF are capable of containing the required numeric data. LINK is given the POINTER attribute which specifies that the value of LINK will usually denote the address of a BASED datatype such as TERM. The variable P, following BASED, is implicitly defined as a pointer variable, and its purpose will be discussed shortly.

This declaration merely defines the structure TERM; it does not create any nodes. The programmer has complete control over obtaining required nodes from an available area of memory. The execution of the statement

```
ALLOCATE TERM;
```

creates a node and automatically assigns the address of the new node to P. Note that many TERM nodes may be in use at one time, but each one must be generated by the execution of an ALLOCATE statement.

The referencing of a node or a particular field is accomplished by using pointer qualification. For example, P−>TERM would denote the node generated by the preceding ALLOCATE statement. Each field could also be referenced as Q−>XEXP, Q−>YEXP, Q−>COEF, or Q−>LINK. The imitation arrow is a minus sign followed immediately by a "greater than" sign. As an example, we can add the coefficients of two polynomial terms, indicated by P and Q, using the statement

```
X = Q->COEF + P->COEF;
```


PL/I not only allows the allocation of based structures, but also permits the freeing of such datatypes to available storage. All this is accomplished by the statement

```
FREE P->TERM;
```

when the node indicated by pointer P is to be restored to the availability area.

It was mentioned that pointer variables usually specify an address. An exception is that they may be assigned the value returned by PL/I's built-in NULL function. This value cannot be related to any address and therefore cannot be interpreted as a pointer to a node. NULL returns the same value on each invocation, and therefore it can be used as an end delimiter. The comparisons "equal" and "not equal" can be made between NULL and a pointer variable as well as between two pointer variables.

These PL/I programming concepts should provide an adequate background for the list processing programs in this and other sections. A program for algorithm *INSERT* is given in Fig. 2-2.4. In procedure INSERT and also the subprogram of Fig. 2-2.7 (DELETE), the structure TERM is not declared. It is assumed that these procedures are nested within an invoking routine in which the necessary declarations are made.

Now that it is possible to invoke the procedure INSERT, the construction of a linked list for a polynomial is achieved by having a zero polynomial initially and repeatedly invoking the insertion function until all terms of the polynomial are processed. For the polynomial $x^2 + 3xy + y^2$, this process must be done 3 times. Since we want the first element of the list to be x^2, we start by inserting the term y^2 followed by the insertion of $3xy$, etc. (we actually have a linked list here).

If the pointer to the first node of the list is POLY, then the program in Fig. 2-2.5 will construct the above polynomial. A trace of this invoking procedure is given in Fig. 2-2.6.

Let us examine another equally important algorithm, that of deleting a node from a linked list.

Algorithm *DELETE* Given a variable *FIRST* whose value denotes the address of the first node in the linked list, it is required to delete the node whose address

```
INSERT:
        PROCEDURE(NX,NY,NCOEF,FIRST) RETURNS(POINTER);
/*INSERT A NODE IN THE LINKED LIST WHICH WILL IMMEDIATELY PRECEDE THE
  NODE WHOSE ADDRESS IS DESIGNATED BY THE POINTER FIRST.  INITIALIZE
  THE NODE'S FIELDS TO NX, NY, AND NCOEF.  RETURN THE POINTER TO THE
  NEW NODE.
        DECLARE
            (NX,NY) BINARY FIXED,
            NCOEF BINARY FLOAT,
            FIRST POINTER;
        ALLOCATE TERM;
        P->XEXP = NX;
        P->YEXP = NY;
        P->COEF = NCOEF;
        P->LINK = FIRST;
        RETURN(P);
END INSERT;
```

FIGURE 2-2.4 PL/I procedure for algorithm *INSERT*.

Scanned with OKEN Scanner

```
POLYST:
    PROCEDURE OPTIONS(MAIN);
/*CONSTRUCT A LINKED LIST REPRESENTATION OF A POLYNOMIAL USING
  THE INSERT PROCEDURE. */
    DECLARE
        1 TERM BASED(P),
            2 XEXP BINARY FIXED,
            2 YEXP BINARY FIXED,
            2 COEF BINARY FLOAT,
            2 LINK POINTER,
        Q POINTER,
        POLY POINTER,
        INSERT ENTRY(BIN FIXED,BIN FIXED,BIN FLOAT,PTR) RETURNS(PTR);

    POLY = NULL; /* INITIALIZE */

    POLY = INSERT(0,2,1,POLY); /* INSERT LAST TERM OF POLYNOMIAL */

    POLY = INSERT(1,1,3,POLY); /* INSERT SECOND TERM OF POLYNOMIAL */

    POLY = INSERT(2,0,1,POLY); /* INSERT FIRST TERM OF POLYNOMIAL */

END POLYST;
```

FIGURE 2-2.5 Construction of the polynomial $x^2 + 3xy + y^0$.

is given by variable X. Auxiliary pointer variables $NEXT$ and $PRED$ are used The list is composed of nodes with structure previously described as $TERM$.

1 [Empty list?] If $FIRST = NULL$ then write 'underflow' and Exit.

2 [Delete first node?] If $X = FIRST$ then set $FIRST \leftarrow LINK(FIRST)$ and go to step 8.

3 [Initiate search for predecessor of X.] Set $NEXT \leftarrow FIRST$.

4 [Update pointers.] Set $PRED \leftarrow NEXT$ and $NEXT \leftarrow LINK(NEXT)$.

5 [End of list?] If $NEXT = NULL$ then write 'node not found' and Exit.

6 [Is this node X?] If $NEXT \neq X$ then go to step 4.

7 [Delete X.] Set $LINK(PRED) \leftarrow LINK(X)$.

8 [Return node X.] Restore node X to the availability area and Exit.

The first step in the algorithm checks for an underflow. The second step determines whether the node to be deleted is the first node of the list, and if it is, then the second node of the list becomes the new first node. In the case of a list containing a single node, the pointer variable $FIRST$ will assume the null value as a result of the deletion.

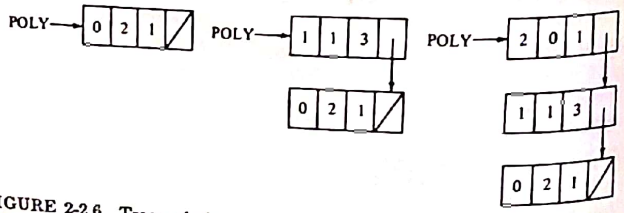


FIGURE 2-2.6 Trace of the construction of a linked list for polynomial $x^2 + 3xy + y^0$.

If X is not the first node in the list, then a search to find the immediate predecessor of X ($PRED$ in the algorithm) is launched. This search is accomplished by chaining through the list and storing the address of the next node in variable $NEXT$ until X is found. A value of "$NULL$" for $NEXT$ indicates that the node to be deleted has not been found in the list and that an error has been made. When variables $NEXT$ and X contain the same value, we are now in a position to change the link field of the predecessor node of X to point to its successor node ($LINK(X)$). This change is accomplished in step 7 of the algorithm, and finally the deleted node is returned to the availability list. ////

A PL/I procedure for the algorithm is given in Fig. 2-2.7.

2-2.5 An Application of Bit Represented Sets

For any representation of a set, it will be necessary to assign an arbitrary ordering to the elements of the set. Such an ordering will be assumed throughout this discussion.

Let us consider the problem of determining whether two sets are equal. Intuitively, the basic operation required for checking the equality of sets is the comparison of their elements. This requirement implies that each individual element of a set must at some time be selected for comparison. We can do this by

```
DELETE:
        PROCEDURE(X,FIRST);
/*FIND AND DELETE NODE X FROM THE
  POLYNOMIAL LIST POINTED TO BY FIRST. */
        DECLARE
            (X,FIRST,NEXT,PRED) POINTER;
        IF FIRST = NULL
        THEN
            DO;/* INDICATE THAT LIST IS EMPTY */
                PUT SKIP LIST('LIST UNDERFLOW');
                RETURN;
            END;
        IF X = FIRST
        THEN
            DO; /* DELETE THE FIRST NODE */
                FIRST = FIRST->LINK;
                GO TO FREE_NODE;
            END;
        NEXT = FIRST; /*INITIATE SEARCH */
LOOP:   PRED = NEXT; /* UPDATE NEXT AND PRED */
        NEXT = NEXT->LINK;
        IF NEXT = NULL
        THEN
            DO; /*INDICATE NODE WAS NOT FOUND */
                PUT SKIP LIST('NODE NOT FOUND');
                RETURN;
            END;
        IF NEXT ¬= X
        THEN GO TO LOOP;
        PRED->LINK = X->LINK; /* DELETE X */
FREE_NODE: FREE X->TERM; /*RESTORE NODE X TO THE AVAILABILITY AREA */
        RETURN;
END DELETE;
```

FIGURE 2-2.7 PL/I procedure for algorithm *DELETE*.

Scanned with OKEN Scanner

using a vector (of integers or strings) to represent a set, since each element could then be selected by appending a subscript to the set name. If vectors are chosen as representations of sets, then the algorithms for testing set equality or performing set operations tend to be complex and inefficient. For these reasons we will not discuss the vector representation any further, but pursue a different approach.

Recall that in Sec. 2-1.3 each subset of a given set was referenced by a letter with a binary subscript. For example, given the set $S_3 = \{a, b, c\}$, the subset $\{a, c\}$ was denoted B_{101}. The 1 bits in 101 indicate that the first and third elements of S_3 are in the subset. Similarly, assume that a universal set, say E, has n elements, each element being distinguishable as first, second, or third, and so on (E is ordered in the sense of Sec. 2-1.8). We will need a sequence of n bits to represent a subset A of this universal set. When we number the bit positions from left to right as 1 to n, each bit corresponds to a particular element in set E. If A contains the ith element of E, then the ith bit of the representative bit sequence will be a 1; otherwise it will be a 0.

Along with this convenient and compact representation of sets, we can achieve simplicity and speed in performing set operations and comparing sets. Obviously, two sets are equal if bit sequences representing them are equal. Using the logical operations *AND*, *NOT*, and *OR* available on most computers and in certain programming languages, we can perform the set operations of intersection, complementation, and union respectively. As an example, consider the two subsets $\{a, c\}$ and $\{c, e\}$ of $\{a, b, c, d, e\}$. They are represented by the bit sequences 10100 and 00101 respectively. To find the intersection of the sets, we perform the operation

$$10100 \wedge 00101 \qquad (AND \text{ corresponding bits})$$

giving the bit sequence 00100, representing $\{c\}$. Similarly,

$$10100 \vee 00101 \qquad (\text{OR corresponding bits})$$

will give 10101, denoting the union $\{a, c, e\}$ of $\{a, c\}$ and $\{c, e\}$, and

$$\neg 10100 \qquad (\text{negate each bit})$$

will give the sequence 01011, denoting $\{b, d, e\}$, the absolute complement of $\{a, c\}$. Bit strings and operations on them will prove useful in the following discussion of the retrieval of information from a file.

Predicate logic and set theory play a very important role in the area of information organization and retrieval. A literature search of a certain topic area in a library system or the number of graduate students who are married and in the College of Engineering in a student record application may be required. This type of application occurs in so many different situations that a special class of languages known as question-answering languages has been created. A typical statement in such languages specifies a predicate whose extension set is required.

We first give a number of definitions which are used in dealing with information, and then proceed to consider as a concrete example a simple student record application.

A collection of information concerning a particular item or individual is called a *record*. Each portion of this record which gives one specific attribute of the item or individual is a *field*. A set of records is a *file*. If a file contains information on a complete set, for example, all books in a library or all employees working for a corporation, then the file is a *master file*. A subset of a file is a *subfile*. In information retrieval, records having specified values in particular fields must be searched for. A *query* is used to formally describe the subfile of records being requested.

Consider a file of student records which have four fields, one each for student number, sex, college, and marital status. If a request is made for information pertaining to students in the College of Engineering, then every record must be scanned to determine which are required. Even greater difficulty would be encountered in searching for information on married students in engineering, as both the college and marital status fields must be considered. Is there an easier way to find the records containing this data?

If we assume that the master file is ordered, that is, we can identify a first record, second record, and so on, then we can specify beforehand a number of subsets in the file. This can be done compactly by using bit represented sets. Obviously many bit strings would be required to directly specify all possible subfiles which may be requested. To resolve this problem, only one bit string should be established for each specific value a field may have. Most information requests will be satisfied by the specified subsets or by logical operations performed on their bit string representations, as previously described.

The purpose of this approach is to avoid the complex and lengthy search often necessary for the retrieval of information. Bit string represented subsets sacrifice a minimum of storage space in order to decrease search time. Note, however, that they should not be used to specify records having a certain value in a field which has many possible values. For example, a student number field will have a different value for each record, and it is therefore not practical to allocate a bit string for each student number. When used in conjunction with a field such as marital status, only several bit strings are necessary to specify each possible status subset. For example, the bit string 10010110... could be used to indicate that the first, fourth, sixth, etc. records pertain to married students.

We will give a PL/I program to demonstrate the construction and use of bit string represented subsets of a master student file. An array of structures denoted by STUDENT is used for the master file. Each element of this array is subdivided into the fields NUMBER, SEX, COLLEGE, and MARTIAL_STATUS, thus representing one record. Information is coded for the latter three fields, as shown in the program comments of Fig. 2-2.8. A bit string array is allocated for each of these fields. One bit string is used for each code a field may contain, and it is simply referenced by that code. This means, for example, that SEX_FILE(1) and SEX_FILE(2) will indicate the subfiles or subsets of records referring to male and female students respectively. SEX_FILE(1) therefore indicates the extension set of the predicate

x is a male student.

Similarly, COLLEGE_FILE(2) is a bit string indicating the extension set of

the predicate

x is a commerce student.

We can find the extension set of the predicate

x is a male commerce student.

by finding the intersection of the two preceding extension sets. This is done by performing the logical operation

SEX_FILE(1)&COLLEGE_FILE(2)

```
BITSETS:
   PROCEDURE OPTIONS(MAIN);
/*THIS PROGRAM INPUTS CODED INFORMATION ABOUT STUDENTS, CREATING A
  MASTER FILE AND THREE BIT STRING ARRAYS.  THE MASTER FILE CONSISTS OF
  AN ARRAY OF STRUCTURES NAMED STUDENT.  EACH ELEMENT OF THIS ARRAY
  HAS FOUR FIELDS NAMED AND CONTAINING CODES AS FOLLOWS:
      NUMBER          - A SIX DIGIT STUDENT NUMBER
      SEX             - 1 - MALE
                      - 2 - FEMALE
      COLLEGE         - 1 - ARTS AND SCIENCE
                      - 2 - COMMERCE
                      - 3 - ENGINEERING
                      - 4 - GRADUATE STUDIES
                      - 5 - HOME ECONOMICS
                      - 6 - AGRICULTURE
      MARITAL_STATUS  - 1 - SINGLE
                      - 2 - MARRIED
                      - 3 - OTHER
  FOR THE FIELDS SEX, COLLEGE, AND MARITAL_STATUS, THE ARRAYS SEX_FILE,
  COLLEGE_FILE, AND STATUS_FILE ARE ESTABLISHED AS REPRESENTATIONS OF
  MUTUALLY EXCLUSIVE SUBSETS OF THE MASTER FILE.  THE ELEMENTS OF THESE
  ARRAYS ARE BIT STRINGS HAVING A LENGTH EQUAL TO THE NUMBER OF RECORDS
  IN THE MASTER FILE.  THERE ARE A TOTAL OF 11 BIT STRINGS, ONE FOR
  EACH OF THE ABOVE CODES.  EACH STRING INITIALLY CONSISTS ENTIRELY OF
  '0' BITS.
  THE FIRST PART OF THE MAIN PROGRAM INPUTS THE STUDENT FILE AND
  CONSTRUCTS THE BIT REPRESENTED SETS AS FOLLOWS:  FOR THE ITH RECORD,
  IF THE CODES J, K, AND M ARE IN THE FIELDS SEX, COLLEGE, AND MARITAL_
  STATUS RESPECTIVELY, THEN THE ITH BIT OF BIT STRINGS SEX_FILE(J),
  COLLEGE_FILE(K), AND STATUS_FILE(M) ARE SET TO '1'B.
  BEGINNING AT THE LABEL QUERY, A NUMBER OF EXTENSION SETS OF CERTAIN
  PREDICATES ARE PRINTED USING PROCEDURE OUTPUT.  THE FIRST INVOCATION
  PRINTS EVERY RECORD AND THE NEXT FIVE PRINT CERTAIN SUBSETS OF THE
  MASTER FILE.  THE USE OF LOGICAL CONNECTIVES IN FINDING
  INTERSECTIONS, UNIONS, OR COMPLEMENTATIONS OF EXTENSION SETS IS
  DEMONSTRATED.                                                    */
   GET LIST(N);
BEGIN; /* AUTOMATIC STORAGE ALLOCATION */
   DECLARE
      1 STUDENT(N),
        2 NUMBER FIXED(6),
        2 SEX FIXED(1),
        2 COLLEGE FIXED(1),
        2 MARITAL_STATUS FIXED(1),
      (SEX_FILE(2),COLLEGE_FILE(6),STATUS_FILE(3)) BIT(N)
         INITIAL((11)(1)'0'B),
      SEX_WORD(2) CHAR(10) INITIAL('MALE','FEMALE'),
      COLLEGE_WORD(6) CHAR(20) INITIAL('ARTS AND SCIENCE','COMMERCE',
         'ENGINEERING','GRADUATE STUDIES','HOME ECONOMICS',
         'AGRICULTURE'),
      STATUS_WORD(3) CHAR(7) INITIAL('SINGLE','MARRIED','OTHER'),
      I BINARY FIXED,
      HEADINGS CHAR(54) INITIAL
         ('NUMBER    SEX       COLLEGE                 MARITAL STATUS'||
```

FIGURE 2-2.8 PL/I program applying bit represented sets.


```
      DO I = 1 TO N; /*INPUT A RECORD AND PUT '1' BITS IN BIT STRINGS.*/
         GET LIST(STUDENT(I));
         SUBSTR(SEX_FILE(SEX(I)),I,1) = '1'B;
         SUBSTR(COLLEGE_FILE(COLLEGE(I)),I,1) = '1'B;
         SUBSTR(STATUS_FILE(MARITAL_STATUS(I)),I,1) = '1'B;
      END;
QUERY:
      CALL OUTPUT('ALL STUDENTS',SEX_FILE(1)|SEX_FILE(2));
      CALL OUTPUT('FEMALE STUDENTS',(SEX_FILE(2)));
      CALL OUTPUT('ARTS STUDENTS',(COLLEGE_FILE(1)));
      CALL OUTPUT('UNMARRIED STUDENTS IN ARTS',COLLEGE_FILE(1)&
         ¬STATUS_FILE(2));
      CALL OUTPUT('MALE STUDENTS IN COMMERCE, ARTS, OR ENGINEERING',
         (COLLEGE_FILE(1)|COLLEGE_FILE(2)|COLLEGE_FILE(3))&SEX_FILE(1));
      CALL OUTPUT('STUDENTS WHO ARE FEMALE AND SINGLE OR IN HOME '||
         'ECONOMICS',(SEX_FILE(2)&STATUS_FILE(1))|COLLEGE_FILE(5));


OUTPUT:
      PROCEDURE (TITLE,STRING);

            O U T P U T
*
TITLE: DESCRIBES RECORDS TO BE OUTPUT.
STRING: BIT STRING INDICATING WHICH RECORDS SHOULD BE OUTPUT.
HEADINGS: FOR THE OUTPUT FIELDS.
I: INDEX TO '1' BITS IN STRING.
ALL OTHER VARIABLES ARE DECLARED IN THE MAIN PROCEDURE.
  SCAN STRING AND OUTPUT THE RECORDS IN THE STUDENT FILE CORRESPONDING
  TO THE '1' BITS.  EACH '1' BIT IS CHANGED TO A '0' BIT AFTER IT IS
  SCANNED.  THIS PROCEDURE DOES NOT PRESERVE STRING SO THE
  CORRESPONDING ARGUMENT MUST BE PASSED BY VALUE.          */
      DECLARE
         TITLE CHAR(60) VARYING,
         STRING BIT(*),
         I BINARY FIXED;
      PUT SKIP(2) LIST(TITLE);
      PUT SKIP EDIT(HEADINGS)(A(54));
      DO WHILE('1'B); /* SCAN STRING FOR '1' BITS */
         I = INDEX(STRING,'1'B);
         IF I = 0  /* ALL REQUIRED RECORDS HAVE BEEN PRINTED */
         THEN RETURN;
         SUBSTR(STRING,I,1)='0'B;
         PUT SKIP EDIT(NUMBER(I),SEX_WORD(SEX(I)),
            COLLEGE_WORD(COLLEGE(I)),STATUS_WORD(MARITAL_STATUS(I)))
            (F(6),X(4),A(10),A(20),A(7));
      END;
END OUTPUT;

END; /* OF BEGIN BLOCK */

END BITSETS;
```

FIGURE 2-2.8 (Continued)

in PL/I. This operation is the required formal query. Similarly, unions and complementations of extension sets can be obtained using the "|" (OR) and "¬" (NOT) operators.

The PL/I program in Fig. 2-2.8 demonstrates the use of these concepts. Notice the built-in functions SUBSTR and INDEX. For example, the statement

SUBSTR(STRING, I, 1) = '0'B;

replaces the Ith bit in STRING by '0'B. The statement

I = INDEX(STRING, '1'B);

assigns to I an integer value which indicates the position in STRING of the leftmost occurrence of '1'B. If there is no such occurrence, zero is assigned to I.

Scanned with OKEN Scanner

```
ALL STUDENTS
NUMBER    SEX       COLLEGE                 MARITAL STATUS
596426    MALE      COMMERCE                MARRIED
600868    MALE      GRADUATE STUDIES        MARRIED
621656    MALE      ENGINEERING             MARRIED
640621    MALE      ARTS AND SCIENCE        MARRIED
652079    FEMALE    ARTS AND SCIENCE        MARRIED
672281    MALE      ARTS AND SCIENCE        OTHER
672915    MALE      ENGINEERING             OTHER
672919    FEMALE    COMMERCE                SINGLE
681242    MALE      ENGINEERING             SINGLE
681614    FEMALE    HOME ECONOMICS          SINGLE
683369    FEMALE    ARTS AND SCIENCE        OTHER
690528    FEMALE    HOME ECONOMICS          MARRIED
702136    FEMALE    HOME ECONOMICS          SINGLE
703062    MALE      AGRICULTURE             SINGLE
720153    MALE      ARTS AND SCIENCE        SINGLE

FEMALE STUDENTS
NUMBER    SEX       COLLEGE                 MARITAL STATUS
652079    FEMALE    ARTS AND SCIENCE        MARRIED
672919    FEMALE    COMMERCE                SINGLE
681614    FEMALE    HOME ECONOMICS          SINGLE
683369    FEMALE    ARTS AND SCIENCE        OTHER
690528    FEMALE    HOME ECONOMICS          MARRIED
702136    FEMALE    HOME ECONOMICS          SINGLE

ARTS STUDENTS
NUMBER    SEX       COLLEGE                 MARITAL STATUS
640621    MALE      ARTS AND SCIENCE        MARRIED
652079    FEMALE    ARTS AND SCIENCE        MARRIED
672281    MALE      ARTS AND SCIENCE        OTHER
683369    FEMALE    ARTS AND SCIENCE        OTHER
720153    MALE      ARTS AND SCIENCE        SINGLE

UNMARRIED STUDENTS IN ARTS
NUMBER    SEX       COLLEGE                 MARITAL STATUS
672281    MALE      ARTS AND SCIENCE        OTHER
683369    FEMALE    ARTS AND SCIENCE        OTHER
720153    MALE      ARTS AND SCIENCE        SINGLE

MALE STUDENTS IN COMMERCE, ARTS, OR ENGINEERING
NUMBER    SEX       COLLEGE                 MARITAL STATUS
596426    MALE      COMMERCE                MARRIED
621656    MALE      ENGINEERING             MARRIED
640621    MALE      ARTS AND SCIENCE        MARRIED
672281    MALE      ARTS AND SCIENCE        OTHER
672915    MALE      ENGINEERING             OTHER
681242    MALE      ENGINEERING             SINGLE
720153    MALE      ARTS AND SCIENCE        SINGLE

STUDENTS WHO ARE FEMALE AND SINGLE OR IN HOME ECONOMICS
NUMBER    SEX       COLLEGE                 MARITAL STATUS
672919    FEMALE    COMMERCE                SINGLE
681614    FEMALE    HOME ECONOMICS          SINGLE
690528    FEMALE    HOME ECONOMICS          MARRIED
702136    FEMALE    HOME ECONOMICS          SINGLE
```

FIGURE 2-2.8 (Continued)

Note that the records were manually sorted by student number. This assumption is not necessary for the proper functioning of the program.

Special care must be taken in the use of procedure OUTPUT. It prints the records indicated by the value of STRING. TITLE is a character description of the records printed. Note that the value, not the name, of STRING's corresponding argument is passed. This passing of value is necessary because OUTPUT replaces each detected '1'B by '0'B, thus destroying the original string passed.

EXERCISES 2-2

1 Consider a two-dimensional array A whose subscript limits are

$$-3 \le i \le 6 \qquad 3 \le j \le 10$$

Give the addressing function for the element $A[i, j]$ where the storage representation is in row major order.

2 Consider a two-dimensional array A whose subscript limits are

$$-2 \le i \le 5 \qquad 2 \le j \le 7$$

Give the addressing function for the element $A[i, j]$ where the storage representation is in column major order.

3 Based on the discussion in the text of representing polynomials in two variables by the use of matrices, formulate an input algorithm to convert a polynomial to its matrix representation and a converse output algorithm. Assume that each polynomial in variables x and y is given on *one* card and has the form for polynomial

$$2x^2 + 5xy + y^2$$

$$2x2+5xy+y2$$

of

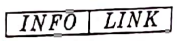
with *no* blanks used for spacing the terms. If a variable is present but its exponent is omitted, it is assumed to have a value of 1. If the coefficient of a polynomial term is omitted, its value is also assumed to be 1. The coefficients are assumed to be real numbers.

4 Using the algorithms developed in Problem 3, write an algorithm which will add two polynomials.

5 Using the algorithms developed in Problem 3, devise an algorithm which will multiply two polynomials. Note that the resulting polynomial will, in general, have higher-order terms than the polynomials being multiplied and therefore an appropriate size matrix must be declared.

6 Formulate an algorithm which appends (concatenates) a list to another list.

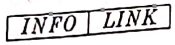
7 Assume that one is given an ordered linked linear list whose typical node consists of an information field and a link field as shown below:



The list is ordered on the field $INFO$ in the sense that node P precedes node Q in the list if and only if $INFO(P) < INFO(Q)$. It is required to devise an algorithm which inserts a new node whose information field is denoted by X, such that the updated list will remain ordered. Write a program corresponding to the algorthim devised.

8 Given a simple linked list whose first node is denoted by the pointer variable $FIRST$, it is required to deconcatenate (or split) this list into two simple linked lists. The node denoted by the pointer variable $SPLIT$ is to be the first element in the second linked list. Formulate a step-by-step algorithm to perform this task.

9 Given a simple linked list whose typical node is represented by

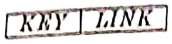


and whose first node is denoted by the pointer variable $FIRST$, it is required to devise an algorithm that will copy this list. The new list has nodes of the form

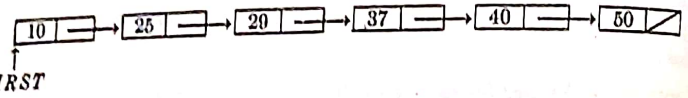
Scanned with OKEN Scanner

and the pointer variable *BEGIN* is to denote the first node. The variables *INFO* and *FIELD* represent the information content of a node, while *LINK* and *PTR* are variables containing the address of the next node. Give the name *COPY* to the algorithm.

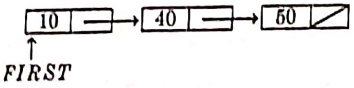
10 Suppose that you are given a simple linked list whose first node is denoted by the pointer variable *FIRST* and whose typical node is represented by



where the variables *KEY* and *LINK* represent the information and link fields of the node, respectively. The list is ordered on the field *KEY* such that the first and last nodes contain the smallest and largest values of the field. It is desired to delete a number of consecutive nodes whose *KEY* values are greater than or equal to *KMIN* and less than *KMAX*. For example, an initial list with *KMIN* and *KMAX* having values of 25 and 40, respectively, could look like this:



After deleting the designated nodes, the updated list would reduce to the following:



In this example, we dropped the nodes whose *KEY* field values are 25, 20, and 37. Formulate an algorithm and write a program which will accomplish the deletion operation for an arbitrary linked list.

2-3 RELATIONS AND ORDERING

The concept of a relation is a basic concept in everyday life as well as in mathematics. We have already used various relations. Associated with a relation is the act of comparing objects which are related to one another. The ability of a computer to perform different tasks based upon the result of a comparison is one of its most important attributes which is used several times during the execution of a typical program. In this section we first formalize the concept of a relation and then discuss methods of representing a relation by using a matrix or its graph. The relation matrix is useful in determining the properties of a relation and also in representing a relation on a computer. Various basic properties of relations are given, and certain important classes of relations are introduced. Among these, the compatibility relation and the equivalence relation have useful applications in the design of digital computers and other sequential machines. Partial ordering and its associated terminology are introduced next. The material in Chap. 4 is based upon these notions. Several relations given as examples in this section are used throughout the book. Algorithms to determine certain properties of relations are also given.

2-3.1 Relations

The word "relation" suggests some familiar examples of relations such as the relation of father to son, mother to son, brother to sister, etc. Familiar examples in arithmetic are relations such as "greater than," "less than," or that of equality between two real numbers. We also know the relation between the area of a circle and its radius and between the area of a square and its side. These examples suggest relationships between two objects. The relation between parents and child, the coincidence of three lines, and that of a point lying between two points are examples of relations among three objects. Similar examples exist for relations among four or more objects.

Here, we shall only consider relations, called binary relations, between a pair of objects. Before we give a set-theoretic definition of a relation, we note that a relation between two objects can be defined by listing the two objects as an ordered pair. A set of all such ordered pairs, in each of which the first member has some definite relationship to the second, describes a particular relationship. Of course, we have been motivated by relationships which are familiar and could be given a name. However, this is an undue restriction which will not appear in our definition of a relation.

Definition 2-3.1 Any set of ordered pairs defines a *binary relation*.

We shall call a binary relation simply a relation. It is sometimes convenient to express the fact that a particular ordered pair, say $\langle x, y \rangle \in R$, where R is a relation, by writing $x\ R\ y$ which may be read as "x is in relation R to y."

In mathematics, relations are often denoted by special symbols rather than by capital letters. A familiar example is the relation "greater than" for real numbers. This relation is denoted by $>$. In fact, $>$ should be considered as the name of a set whose elements are ordered pairs. Each member of any of the ordered pairs in the set is a real number, and if a and b are two real numbers such that $a > b$, then we say that $\langle a, b \rangle \in >$, or $a > b$. More precisely the relation $>$ is

$$> = \{\langle x, y \rangle \mid x, y \text{ are real numbers and } x > y\} \tag{1}$$

The relation of father to his child can be described by a set, say F, of ordered pairs in which the first member is the name of the father and the second the name of his child. That is,

$$F = \{\langle x, y \rangle \mid x \text{ is the father of } y\} \tag{2}$$

The definition of relation permits any set of ordered pairs to define a relation. For example, the set S given by

$$S = \{\langle 2, 4 \rangle, \langle 1, 3 \rangle, \langle \lambda, 6 \rangle, \langle \text{Joan}, \mu \rangle\} \tag{3}$$

can be considered as a relation. Of course, such a relation may not be familiar or interesting.

Scanned with OKEN Scanner

Scanned with OKEN Scanner

Let R denote the set of real numbers. Then

$$Q = \{\langle x^2, x\rangle \mid x \in \mathbf{R}\} \tag{4}$$

defines the relation of the square of a real number.

Definition 2-3.2 Let S be a binary relation. The set $D(S)$ of all objects x such that for some y, $\langle x, y\rangle \in S$ is called the *domain* of S, that is,

$$D(S) = \{x \mid (\exists y)(\langle x, y\rangle \in S)\}$$

Similarly, the set $R(S)$ of all objects y such that for some x, $\langle x, y\rangle \in S$ is called the *range* of S, that is,

$$R(S) = \{y \mid (\exists x)(\langle x, y\rangle \in S)\}$$

For the relation S described in Eq. (3) we have

$$D(S) = \{2, 1, \lambda, \text{Joan}\} \quad \text{and} \quad R(S) = \{4, 3, 6, \mu\}$$

Let X and Y be any two sets. A subset of the cartesian product $X \times Y$ defines a relation, say C. For any such relation C, we have $D(C) \subseteq X$ and $R(C) \subseteq Y$, and the relation C is said to be from X to Y. If $Y = X$, then C is said to be a relation from X to X. In such a case, C is called a relation in X. Thus any relation in X is a subset of $X \times X$. The set $X \times X$ itself defines a relation in X and is called a *universal relation* in X, while the empty set which is also a subset of $X \times X$ is called a *void relation* in X.

The relations given in Eqs. (1) and (4) are in the set R of real numbers. The relation in Eq. (2) is in the set of all human beings. It could also be considered as a relation from the set of all males to the set of human beings. The relation S in Eq. (3) can be considered from a set X to a set Y where $\{2, 1, \lambda, \text{Joan}\} \subseteq X$ and $\{4, 3, 6, \mu\} \subseteq Y$. In fact any relation from a set X to a set Y can also be considered as a relation in $X \cup Y$.

If R is the set of real numbers, then the elements of $\mathbf{R} \times \mathbf{R}$ can be represented by points in a plane, as shown in Sec. 2-1.8. Some of the subsets of $\mathbf{R} \times \mathbf{R}$ define familiar relations which can be shown graphically. For example, consider the relations in which $x * y$ means xy

$$\begin{aligned} R_1 &= \{\langle x, y\rangle \mid \langle x, y\rangle \in \mathbf{R} \times \mathbf{R} \wedge x * y \geq 1\} \\ R_2 &= \{\langle x, y\rangle \mid \langle x, y\rangle \in \mathbf{R} \times \mathbf{R} \wedge x^2 + y^2 \leq 9\} \\ R_3 &= \{\langle x, y\rangle \mid \langle x, y\rangle \in \mathbf{R} \times \mathbf{R} \wedge y^2 < x\} \end{aligned} \tag{5}$$

R_1 can be represented by points on one side of a hyperbola, R_2 by points inside or on a circle of radius 3, and R_3 by points on one side of a parabola. These relations are displayed in Fig. 2-3.1.

A relation has been defined as a set of ordered pairs. It is therefore possible to apply the usual operations of sets to relations as well. The resulting sets will also be ordered pairs and will define some relation. If R and S denote two relations, then $R \cap S$ defines a relation such that

$$x\,(R \cap S)\,y \Leftrightarrow x\,R\,y \wedge x\,S\,y$$

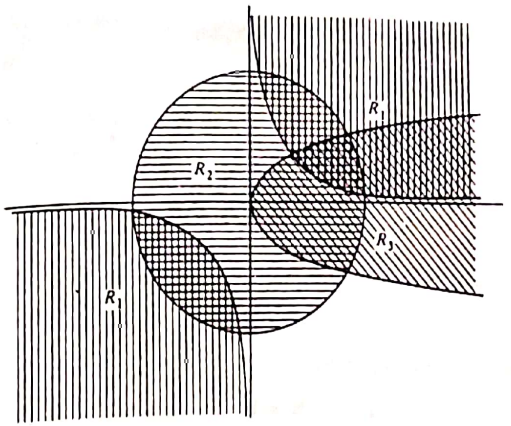


FIGURE 2-3.1

Similarly, $R \cup S$ is a relation such that

$$x\,(R \cup S)\,y \Leftrightarrow x\,R\,y \vee x\,S\,y$$

Also

$$x\,(R - S)\,y \Leftrightarrow x\,R\,y \wedge x\,\not S\,y$$

and

$$x\,(\sim R)\,y \Leftrightarrow x\,\not R\,y$$

EXAMPLE 1 (*a*) Let $X = \{1, 2, 3, 4\}$. If

$R = \{\langle x, y\rangle \mid x \in X \wedge y \in X \wedge ((x - y) \text{ is an integral nonzero multiple of } 2)\}$
$= \{\langle 1, 3\rangle, \langle 3, 1\rangle, \langle 2, 4\rangle, \langle 4, 2\rangle\}$
$S = \{\langle x, y\rangle \mid x \in X \wedge y \in X \wedge ((x - y) \text{ is an integral nonzero multiple of } 3)\}$
$= \{\langle 1, 4\rangle, \langle 4, 1\rangle\}$

Find $R \cup S$ and $R \cap S$.

(*b*) If $X = \{1, 2, 3, \ldots\}$, what is $R \cap S$ for R and S as defined in (*a*)?

SOLUTION

(*a*) $R \cup S = \{\langle 1, 3\rangle, \langle 3, 1\rangle, \langle 2, 4\rangle, \langle 4, 2\rangle, \langle 1, 4\rangle, \langle 4, 1\rangle\}$ and $R \cap S = \varnothing$.
(*b*) $R \cap S = \{\langle x, y\rangle \mid x \in X \wedge y \in X \wedge ((x - y) \text{ is a nonzero multiple of } 6)\}$. ////

Let us return to the relations R_1, R_2, R_3 given in Eq. (5). The predicates P_1, P_2, and P_3 defined by

$$P_1: x * y \geq 1 \qquad P_2: x^2 + y^2 \leq 9 \qquad P_3: y^2 < x$$

describe the relations R_1, R_2, and R_3 respectively. Recall from Sec. 2-1.7 that associated with each predicate is an extension set. A more complex predicate such as $P_1 \wedge P_2 \wedge P_3$ having an extension set $R_1 \cap R_2 \cap R_3$ can be written. Given the coordinates of a point $\langle x, y\rangle$, it is instructive to determine whether $\langle x, y\rangle$ is a

member of each of the following sets:

$$R_4 = R_1 \cap R_2 \cap R_3$$
$$= \{\langle x, y\rangle \mid \langle x, y\rangle \in \mathrm{R} \times \mathrm{R} \wedge x * y \geq 1 \wedge x^2 + y^2 \leq 9 \wedge y^2 < x\}$$

$$R_5 = R_2 \cap (R_1 \cup R_3) \cap \sim(R_1 \cap R_3)$$
$$= \{\langle x, y\rangle \mid \langle x, y\rangle \in \mathrm{R} \times \mathrm{R} \wedge x^2 + y^2 \leq 9 \wedge (x * y \geq 1 \vee y^2 < x) \wedge \sim(x * y \geq 1 \wedge y^2 < x)\}$$

$$R_6 = R_1 \cap \sim R_2 \cap R_3$$
$$= \{\langle x, y\rangle \mid \langle x, y\rangle \in \mathrm{R} \times \mathrm{R} \wedge x * y \geq 1 \wedge \sim(x^2 + y^2 \leq 9) \wedge y^2 < x\}$$

$$R_7 = \sim(R_1 \cup R_3) \cap R_2$$
$$= \{\langle x, y\rangle \mid \langle x, y\rangle \in \mathrm{R} \times \mathrm{R} \wedge \sim(x * y \geq 1 \vee y^2 < x) \wedge x^2 + y^2 \leq 9\}$$

R_4 includes all points lying within the circle and the parabola and above the hyperbola of the first quadrant. R_5 includes all points within the circle which lie either within the parabola or above the hyperbola of the first quadrant, but not both, and all points within the circle and below the hyperbola in the third quadrant. R_6 includes all points lying above the hyperbola and within the parabola in the first quadrant. R_7 includes all points lying within the circle and between the hyperbolic curves but not within the parabola.

These newly defined sets can pictorially be represented as shown in Fig. 2-3.2. The program given in Fig. 2-3.3 reads a number of coordinate points and determines whether these points lie in the sets R_4 to R_7. Note that the relations R_4 to R_7 are written as predicates P_4 to P_7 in the program.

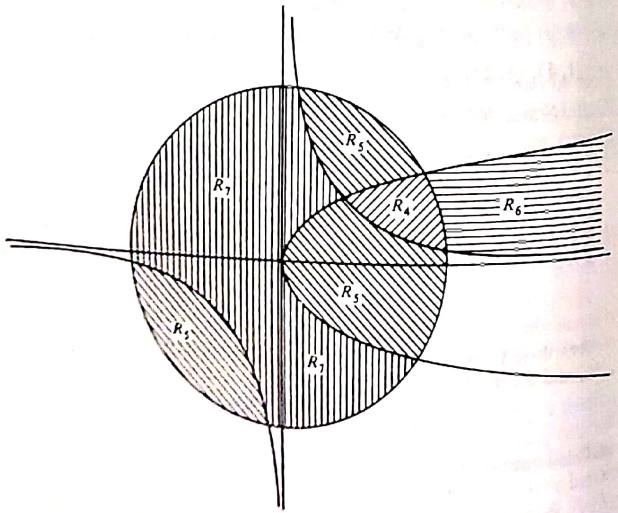


FIGURE 2-3.2


```
C  THIS PROGRAM READS PAIRS OF NUMBERS, X AND Y, AND DETERMINES THE
C  TRUTH VALUES OF THE PREDICATES P(1), P(2), AND P(3).
C  IF THE TRUTH VALUE OF A PREDICATE IS TRUE THEN THE POINT (X,Y) IS
C  AN ELEMENT OF THE RELATION WHICH IS AN EXTENSION SET OF THE
C  PREDICATE, OTHERWISE IT IS NOT.
C  IN ADDITION, THE TRUTH VALUES OF PREDICATES P(4), P(5), P(6), AND
C  P(7) ARE FOUND.  THESE PREDICATES DEFINE RELATIONS WHICH ARE OBTAINED
C  BY PERFORMING THE OPERATIONS OF UNION, INTERSECTION, AND
C  COMPLEMENTATION ON THE EXTENSION SETS OF P(1), P(2) AND P(3).
C
C  P MUST BE A LOGICAL ARRAY
      LOGICAL P(7)

C  PRINT HEADINGS
      WRITE(6,10)
   10 FORMAT('1',5X,'(X,Y) POINTS',22X,'PREDICATES',//,' ',5X,'X',9X,
     1'Y',9X,'P1     P2     P3     P4     P5     P6     P7')

C  READ X AND Y COORDINATES
    1 READ(5,15,END=2)X,Y
   15 FORMAT(F5.2,2X,F5.2)

C  EVALUATE PREDICATES P(1), P(2), AND P(3)
      P(1) = X * Y .GE. 1.
      P(2) = X ** 2 + Y ** 2 .LE. 9
      P(3) = Y ** 2 .LT. X

C  EVALUATE THE PREDICATES WHICH DEFINE SETS OBTAINED THROUGH
C  OPERATIONS ON THE EXTENSION SETS OF P(1), P(2) AND P(3).
      P(4) = P(1) .AND. P(2) .AND. P(3)
      P(5) = P(2) .AND. (P(1) .OR. P(3)) .AND. .NOT.(P(1) .AND .P(3))
      P(6) = P(1) .AND. .NOT.P(2) .AND. P(3)
      P(7) = .NOT.(P(1) .OR. P(3)) .AND. P(2)

C  OUTPUT X, Y AND THE TRUTH VALUES OF THE PREDICATES.
      WRITE(6,20)X,Y,(P(I),I = 1,7)
   20 FORMAT('0',3X,F5.2,5X,F5.2,3X,7(L6))
      GO TO 1
    2 STOP
      END
```

(X,Y) POINTS		PREDICATES						
X	Y	P1	P2	P3	P4	P5	P6	P7
-0.60	0.70	F	T	F	F	F	F	T
1.11	0.50	F	T	T	F	T	F	F
2.20	1.30	T	T	T	T	F	F	F
4.80	1.90	T	F	T	F	F	T	F
-5.00	-1.00	T	F	F	F	F	F	F
1.00	2.30	T	T	F	F	T	F	F
5.00	0.15	F	F	T	F	F	F	F

FIGURE 2-3.3 FORTRAN program for testing membership of coordinates in relations.

EXERCISES 2-3.1

1 Let

$$P = \{\langle 1, 2\rangle, \langle 2, 4\rangle, \langle 3, 3\rangle\} \quad \text{and} \quad Q = \{\langle 1, 3\rangle, \langle 2, 4\rangle, \langle 4, 2\rangle\}$$

Scanned with OKEN Scanner

Find $P \cup Q$, $P \cap Q$, $D(P)$, $D(Q)$, $D(P \cup Q)$, $R(P)$, $R(Q)$, and $R(P \cap Q)$. Show that

$$D(P \cup Q) = D(P) \cup D(Q)$$
$$R(P \cap Q) \subseteq R(P) \cap R(Q)$$

and

2 What are the ranges of the relations

$$S = \{\langle x, x^2 \rangle \mid x \in \mathbf{N}\} \quad \text{and} \quad T = \{\langle x, 2x \rangle \mid x \in \mathbf{N}\}$$

where $\mathbf{N} = \{0, 1, 2, \ldots\}$? Find $R \cup S$ and $R \cap S$.

3 Let L denote the relation "less than or equal to" and D denote the relation "divides," where $x \, D \, y$ means "x divides y." Both L and D are defined on the set $\{1, 2, 3, 6\}$. Write L and D as sets, and find $L \cap D$.

2-3.2 Properties of Binary Relations in a Set

Definition 2-3.3 A binary relation R in a set X is *reflexive* if, for every $x \in X$, $x \, R \, x$, that is, $\langle x, x \rangle \in R$, or

$$R \text{ is reflexive in } X \Leftrightarrow (x)(x \in X \rightarrow x \, R \, x)$$

The relation $\leq$ is reflexive in the set of real numbers since, for any x, we have $x \leq x$. Similarly, the relation of inclusion is reflexive in the family of all subsets of a universal set. The relation of equality of sets is also reflexive. However, the relation $<$ is not reflexive in the set of real numbers, and the relation of proper inclusion is not reflexive in the family of subsets of a universal set.

Definition 2-3.4 A relation R in a set X is *symmetric* if, for every x and y in X, whenever $x \, R \, y$, then $y \, R \, x$. That is,

$$R \text{ is symmetric in } X \Leftrightarrow (x)(y)(x \in X \wedge y \in X \wedge x \, R \, y \rightarrow y \, R \, x)$$

The relations $\leq$ and $<$ are not symmetric in the set of real numbers, while the relation of equality is. The relation of similarity in the set of triangles in a plane is both reflexive and symmetric. The relation of being a brother is not symmetric in the set of all people. However, in the set of all males it is symmetric.

Definition 2-3.5 A relation R in a set X is *transitive* if, for every x, y, and z in X, whenever $x \, R \, y$ and $y \, R \, z$, then $x \, R \, z$. That is,

$$R \text{ is transitive in } X \Leftrightarrow (x)(y)(z)(x \in X \wedge y \in X \wedge z \in X \wedge x \, R \, y \wedge y \, R \, z \rightarrow x \, R \, z)$$

The relations $\leq$, $<$, and $=$ are transitive in the set of real numbers. The relations $\subseteq$, $\subset$, and equality are also transitive in the family of subsets of a universal set. The relation of similarity of triangles in a plane is transitive, while the relation of being a mother is not.

Definition 2-3.6 A relation R in a set X is *irreflexive* if, for every $x \in X$, $\langle x, x \rangle \notin R$.

Note that any relation which is not reflexive is not necessarily irreflexive, and vice versa. The relation $<$ in the set of real numbers is irreflexive because for no x do we have $x < x$. Similarly, the relation of proper inclusion in the set of all nonempty subsets of a universal set is irreflexive. The following is a simple example of a relation in $\{1, 2, 3\}$ which is not reflexive and not irreflexive:

$$S = \{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 3, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 3 \rangle\}$$

Definition 2-3.7 A relation R in a set X is *antisymmetric* if, for every x and y in X, whenever $x \, R \, y$ and $y \, R \, x$, then $x = y$. Symbolically, R is antisymmetric in x iff

$$(x)(y)(x \in X \wedge y \in X \wedge x \, R \, y \wedge y \, R \, x \rightarrow x = y)$$

Note that it is possible to have a relation which is both symmetric and antisymmetric. This is obviously the case when each element is either related to itself or not related to any element.

Some known relations and their properties are now given.

Let R be the set of real numbers. The relations $>$ (greater than) and $<$ (less than) in R are both irreflexive and transitive. Also the relation $=$ (equality) in R is reflexive, symmetric, and transitive.

Let X be the set of all courses offered at a university, and for $x \in X$ and $y \in X$, $x \, R \, y$ if x is a prerequisite for y. The relation of being a prerequisite is irreflexive and transitive.

Let X be the set of all male Canadians and let $x \, R \, y$, where $x \in X$ and $y \in X$, denote the relation "x is a brother of y." The relation R is irreflexive and symmetric but not transitive. In general, any relation which is irreflexive and symmetric cannot be transitive because $x \, R \, y \wedge y \, R \, x \Rightarrow x \, R \, x$, which is not true.

Let X be the collection of the subsets of a universal set. The relation of inclusion in X is reflexive, antisymmetric, and transitive. Also, the relation of proper inclusion in X is irreflexive, antisymmetric, and transitive.

Several important classes of relations having one or more of the properties given here will be discussed later in this section.

EXERCISES 2-3.2

1 Give an example of a relation which is neither reflexive nor irreflexive.

2 Give an example of a relation which is both symmetric and antisymmetric.

3 If relations R and S are both reflexive, show that $R \cup S$ and $R \cap S$ are also reflexive.

4 If relations R and S are reflexive, symmetric, and transitive, show that $R \cap S$ is also reflexive, symmetric, and transitive.

5 Show whether the following relations are transitive:

$$R_1 = \{\langle 1, 1 \rangle\} \qquad R_2 = \{\langle 1, 2 \rangle, \langle 2, 2 \rangle\}$$
$$R_3 = \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 1, 3 \rangle, \langle 2, 1 \rangle\}$$

6 Given $S = \{1, 2, 3, 4\}$ and a relation R on S defined by

$$R = \{\langle 1, 2 \rangle, \langle 4, 3 \rangle, \langle 2, 2 \rangle, \langle 2, 1 \rangle, \langle 3, 1 \rangle\}$$

Scanned with OKEN Scanner

show that R is not transitive. Find a relation $R_1 \supseteq R$ such that R_1 is transitive. Can you find another relation $R_2 \supseteq R$ which is also transitive?

7 Given $S = \{1, 2, \ldots, 10\}$ and a relation R on S where

$$R = \{\langle x, y\rangle \mid x + y = 10\}$$

what are the properties of the relation R?

8 Let R be a relation on the set of positive real numbers so that its graphical representation consists of points in the first quadrant of the cartesian plane. What can we expect if R is (*a*) reflexive, (*b*) symmetric, and (*c*) transitive?

9 Show that the relations L and D given in Problem 3 of Exercises 2-3.1 are both reflexive, antisymmetric, and transitive. Give another example of such a relation. Draw the graphs of these relations as defined in Sec. 2-3.3.

2-3.3 Relation Matrix and the Graph of a Relation

A relation R from a finite set X to a finite set Y can also be represented by a matrix called the *relation matrix* of R.

Let $X = \{x_1, x_2, \ldots, x_m\}$, $Y = \{y_1, y_2, \ldots, y_n\}$, and R be a relation from X to Y. The relation matrix of R can be obtained by first constructing a table whose columns are preceded by a column consisting of successive elements of X and whose rows are headed by a row consisting of the successive elements of Y. If $x_i \, R \, y_j$, then we enter a 1 in the ith row and jth column. If $x_k \not R \, x_l$, then we enter a zero in the kth row and lth column. As a special case, consider $m = 3$, $n = 2$, and R given by

$$R = \{\langle x_1, y_1\rangle, \langle x_2, y_1\rangle, \langle x_3, y_2\rangle, \langle x_2, y_2\rangle\} \tag{1}$$

The required table for R is Table 2-3.1.

If we assume that the elements of X and Y appear in a certain order, then the relation R can be represented by a matrix whose elements are 1s and 0s. This matrix can be written down from the table constructed or can be defined in the following manner.

$$r_{ij} = \begin{cases} 1 & \text{if } x_i \, R \, y_j \\ 0 & \text{if } x_i \not R \, y_j \end{cases}$$

where r_{ij} is the element in the ith row and jth column. The matrix obtained in this way is called the relation matrix. If X has m elements and Y has n elements, then the relation matrix is an $m \times n$ matrix. For the relation R given in Eq. (1), the relation matrix is

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}$$

Table 2-3.1

	y_1	y_2
x_1	1	0
x_2	1	1
x_3	0	1

One can not only write a relation matrix when a relation R is given but also obtain the relation if the relation matrix is given.

Throughout the remainder of this subsection we shall assume that the relations are defined in a set, say X. A relation matrix reflects some of the properties of a relation in a set. If a relation is reflexive, then all the diagonal entries must be 1. If a relation is symmetric, then the relation matrix is symmetric. If a relation is antisymmetric, then its matrix is such that if $r_{ij} = 1$, then $r_{ji} = 0$ for $i \neq j$.

A relation can also be represented pictorially by drawing its *graph*. Although we shall introduce some of the concepts of graph theory which are discussed in Chap. 5, here we shall use graphs only as a tool to represent relations. Let R be a relation in a set $X = \{x_1, \ldots, x_m\}$. The elements of X are represented by points or circles called *nodes*. The nodes corresponding to x_i and x_j are labeled x_i and x_j respectively. These nodes may also be called vertices. If $x_i \, R \, x_j$, that is, if $\langle x_i, x_j\rangle \in R$, then we connect nodes x_i and x_j by means of an arc and put an arrow on the arc in the direction from x_i to x_j. When all the nodes corresponding to the ordered pairs in R are connected by arcs with proper arrows, we get a graph (directed graph) of the relation R. If $x_i \, R \, x_j$ and $x_j \, R \, x_i$, then we draw two arcs between x_i and x_j. For the sake of simplicity, we may replace the two arcs by one arc with arrows pointing in both directions. If $x_i \, R \, x_i$, we get an arc which starts from node x_i and returns to node x_i. Such an arc is called a *loop*. In Fig. 2-3.4 we show some arcs.

From the graph of a relation it is possible to observe some of its properties. If a relation is reflexive, then there must be a loop at each node. On the other hand, if the relation is irreflexive, then there is no loop at any node. If a relation is symmetric and if one node is connected to another, then there must be a return arc from the second node to the first. For antisymmetric relations no such direct return path should exist (see Fig. 2-3.5). If a relation is transitive, the situation is not so simple. In any case, its graph must have loops of the type shown in Fig. 2-3.6.

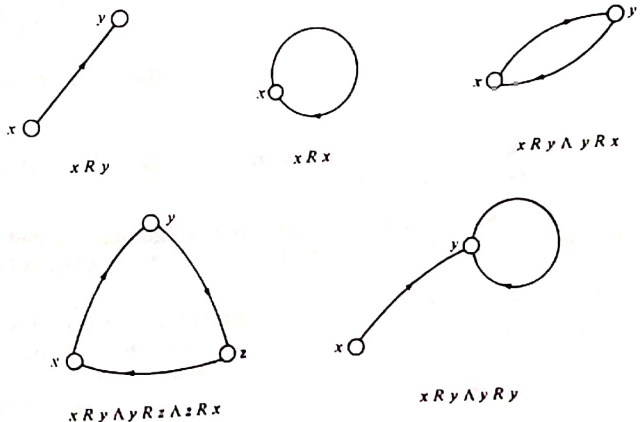


FIGURE 2-3.4 Graphs of relations.

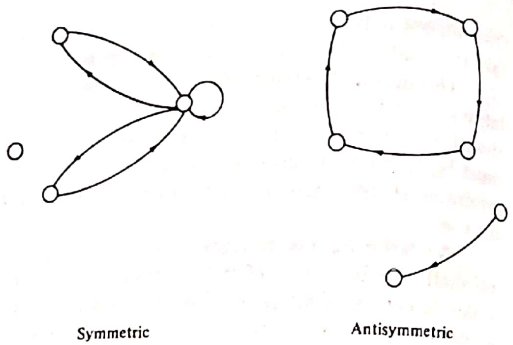


FIGURE 2-3.5 Symmetric and antisymmetric relations.

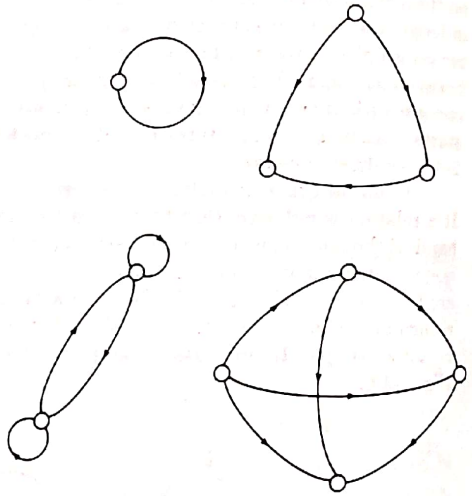
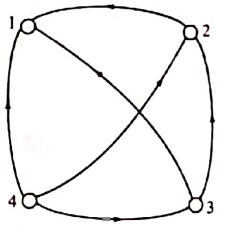


FIGURE 2-3.6 Transitive relations.

EXAMPLE 1 Let $X = \{1, 2, 3, 4\}$ and $R = \{\langle x, y\rangle \mid x > y\}$. Draw the graph of R and also give its matrix.

SOLUTION The graph and the corresponding relation matrix for the relation $R = \{\langle 4, 1\rangle, \langle 4, 2\rangle, \langle 4, 3\rangle, \langle 3, 1\rangle, \langle 3, 2\rangle, \langle 2, 1\rangle\}$ is given in Fig. 2-3.7. ////

EXAMPLE 2 Let $A = \{a, b, c\}$ and denote the subsets of A by $B_0, \ldots, B_7$ according to the convention given in Sec. 2-1.3. Thus $B_0 = \varnothing$, $B_1 = \{c\}$, $B_2 = \{b\}$, $B_3 = \{b, c\}$, $B_4 = \{a\}$, $B_5 = \{a, c\}$, $B_6 = \{a, b\}$, and $B_7 = \{a, b, c\}$. If R is the relation of *proper inclusion* on the subsets $B_0, \ldots, B_7$, then give the matrix of the relation.



$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

FIGURE 2-3.7

SOLUTION

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The relations given in Examples 1 and 2 are both transitive, antisymmetric, and irreflexive. This class of relations will be discussed in Sec. 2-3.8.

EXAMPLE 3 Determine the properties of the relations given by the graphs shown in Fig. 2-3.8, and also write the corresponding relation matrices.

SOLUTION The relation given by the graph in (a) is antisymmetric, in (b) is reflexive, in (c) it is reflexive and symmetric, while in (d) it is transitive. The required matrices are

$$(a)\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (b)\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (c)\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad (d)\begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

////

When the number of elements in a set X over which a relation R is defined is large (say greater than or equal to 5 or 6), both the graphical and the matrix

Scanned with OKEN Scanner

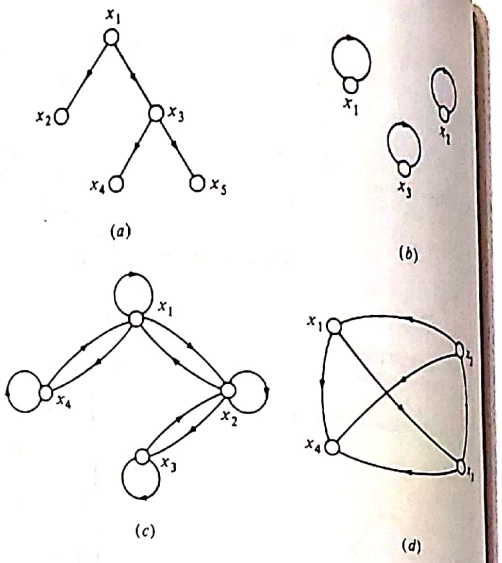


FIGURE 2-3.8

representations of the relation become unwieldy. In these cases, however, the matrix representation can be easily represented on a computer. When a relation matrix is available, it is easy to determine whether a given relation is reflexive or symmetric. It is not always easy to determine from the matrix whether the relation is transitive. We now present two algorithms. The first determines from a relation matrix whether the relation is reflexive and symmetric. The second algorithm then determines whether the relation is also transitive. Relations which are reflexive, symmetric, and transitive are called *equivalence relations*. Equivalence relations are discussed in Sec. 2-3.5.

The entries of the relation matrix are denoted by T and F instead of 1 and 0 in order to conserve storage. Note that in FORTRAN only 1 byte is needed for each logical entry, but at least 2 bytes are required for an integer entry.

Algorithm *REFSYM* Given a relation matrix R representing a relation in the set of positive integers from 1 to n inclusive, it is required to determine if the relation represented by R is symmetric and reflexive. If it is, the variable *FLAG* which is initially *false* is given the truth value *true*; otherwise *FLAG* remains *false*.

1 [Scan each row.] Repeat steps 2 and 3 for $i = 1, 2, \ldots, n$.
2 [Reflexive?] If $R[i, i] = F$ then Exit.
3 [Symmetric?] Repeat for $j = i + 1, i + 2, \ldots, n$:
  If $R[i, j] \neq R[j, i] = F$ then Exit.
4 [Successful test.] Set $FLAG \leftarrow T$ and Exit.

This algorithm scans each row of the matrix from the diagonal element to the right. If a diagonal element has the truth value F, then the algorithm is terminated in step 2 with *FLAG* remaining *false*. Step 3 scans each row in the upper triangle of the matrix. If an entry has a truth value which differs from the truth value of its reflection, then the algorithm is terminated with *FLAG* remaining *false*.

If the algorithm completely scans all the rows, then *FLAG* is set to T, indicating that the relation represented by the relation matrix R is symmetric and reflexive. A subprogram REFSYM is given in Fig. 2-3.9. An example will be given in Sec. 2-3.5. ////

We now give an algorithm to determine whether a given relation which is both reflexive and symmetric is also transitive. The test for transitivity is simplified when a relation is symmetric because ordinarily $i\,R\,j$ and $j\,R\,k$ imply $i\,R\,k$, but in the case of a symmetric relation $i\,R\,j$ and $i\,R\,k$ imply $j\,R\,k$. Algorithm *TRANS* determines whether a given symmetric relation is transitive.

Algorithm *TRANS* Given a relation matrix R which represents a symmetric and reflexive relation, it is required to determine if this relation is transitive. The relation is in the set $\{1, 2, \ldots, n\}$. If it is transitive, then FLAG, which is initially *false*, is set to *true*.

1 [Scan each row.] Repeat steps 2 and 3 for $i = 1, 2, \ldots, n - 2$.
2 [Scan to right of diagonal.] Repeat step 3 for $j = i + 1, i + 2, \ldots, n - 1$.
3 [Transitive?] If $R[i, j] = T$ then repeat for $k = j + 1, j + 2, \ldots, n$: If $R[i, k] = T$ and $R[j, k] = F$ then Exit.
4 [Successful test.] Set $FLAG \leftarrow T$ and Exit.

This algorithm scans only the rows in the upper triangular part of the matrix. The first element to the right of the diagonal which has the truth value T is found. This element is $R[i, j]$. All the following *true* elements are found,

```
      SUBROUTINE REFSYM(A,N,FLAG)
C  THIS SUBROUTINE RECEIVES A LOGICAL ARRAY A WHICH REPRESENTS THE
C  RELATION MATRIX OF A RELATION ON THE SET OF POSITIVE INTEGERS FROM
C  ONE TO N INCLUSIVE.  IF THE RELATION REPRESENTED BY A IS REFLEXIVE
C  AND SYMMETRIC, FLAG, WHICH IS INITIALLY FALSE IS SET TO TRUE.
C  OTHERWISE FLAG IS STILL FALSE WHEN RETURNED TO THE MAINLINE.
C
      LOGICAL*1 A(N,N),FLAG
C
C  I IS THE ROW COUNTER.
      DO 10 I = 1,N
C
C  REFLEXIVE?
      IF(.NOT.A(I,I)) RETURN
      IF(I .EQ. N) GO TO 20
C
C  J IS THE COLUMN COUNTER AND RANGES FROM I+1 TO N.
      K = I + 1
      DO 10 J = K,N
C
C  SYMMETRIC?  A(I,J) IFF A(J,I).
      IF(.NOT.((.NOT.A(I,J).OR.A(J,I)).AND.(.NOT.A(J,I).OR.A(I,J))))
     1RETURN
   10 CONTINUE
   20 FLAG = .TRUE.
      RETURN
      END
```

FIGURE 2-3.9 FORTRAN subroutine for algorithm *REFSYM*.

Scanned with OKEN Scanner

and for each such $R[i, k]$, if $R[j, k]$ is *false*, then the relation is not transitive and the algorithm is finished; otherwise transitivity can be tested further. This test must be performed for all *true* $R[i, j]$ such that $i < j < n$.

Each row, excepting the last, is scanned in this manner. After row $n - 2$ is completed, *FLAG* is set to *true*, indicating that the relation is transitive. A FORTRAN subroutine for this algorithm is given in Fig. 2-3.10. ////

A program which first puts a relation given as a set of ordered pairs into a relation matrix and then determines whether the relation is reflexive, symmetric, and transitive is given in Sec. 2-3.5 along with a further subprogram which obtains equivalence classes.

2-3.4 Partition and Covering of a Set

Definition 2-3.8 Let S be a given set and $A = \{A_1, A_2, \ldots, A_m\}$ where each A_i, $i = 1, \ldots, m$, is a subset of S and

$$\bigcup_{i=1}^{m} A_i = S$$

```
      SUBROUTINE TRANS(A,N,FLAG)
C  THIS SUBROUTINE IS INVOKED AFTER THE RELATION MATRIX A HAS BEEN
C  PROVED TO REPRESENT A REFLEXIVE AND SYMMETRIC RELATION.  IF THE
C  RELATION IS TRANSITIVE, THEN FLAG, WHICH WAS RESET TO FALSE IS SET TO
C  TRUE, INDICATING THAT THE RELATION IS AN EQUIVALENCE RELATION.
C  OTHERWISE, FLAG REMAINS FALSE ON RETURN.
C
      LOGICAL*1 A(N,N),FLAG
C
C  CHECK FOR TRIVIAL CASE:  N = 1 OR N = 2
      IF(N.EQ.1.OR.N.EQ.2) GO TO 20
C
C  I IS THE ROW COUNTER.
      NMIN2 = N - 2
      DO 10 I = 1,NMIN2
C
C  J IS A COLUMN COUNTER RANGING FROM I+1 TO N-1.
      L = I + 1
      NMIN1 = N - 1
      DO 10 J = L,NMIN1
C
C  FIND A PAIR (I,J) FOR WHICH I AND J ARE RELATED.
      IF(.NOT.A(I,J)) GO TO 10
C  K SCANS FROM COLUMN J+1 TO N.
      M = J + 1
      DO 5 K = M,N
C
C  FIND THE FOLLOWING RELATED PAIRS IN ROW I.
      IF(.NOT.A(I,K)) GO TO 5
C
C  TRANSITIVE?  A(I,J) AND A(I,K) IMPLIES A(K,J).
      IF(.NOT.A(K,J)) RETURN
    5 CONTINUE
   10 CONTINUE
C
C  THE RELATION IS AN EQUIVALENCE RELATION.
   20 FLAG = .TRUE.
      RETURN
      END
```

FIGURE 2-3.10 FORTRAN subroutine for algorithm *TRANS*.

Then the set A is called a *covering* of S, and the sets $A_1, A_2, \ldots, A_m$ are said to *cover* S. If, in addition, the elements of A, which are subsets of S, are mutually disjoint, then A is called a *partition* of S, and the sets $A_1, A_2, \ldots, A_m$ are called the *blocks* of the partition.

For example, let $S = \{a, b, c\}$ and consider the following collections of subsets of S

$$A = \{\{a, b\}, \{b, c\}\} \qquad B = \{\{a\}, \{a, c\}\} \qquad C = \{\{a\}, \{b, c\}\}$$
$$D = \{\{a, b, c\}\} \qquad E = \{\{a\}, \{b\}, \{c\}\} \qquad F = \{\{a\}, \{a, b\}, \{a, c\}\}$$

The sets A and F are coverings of S while C, D, and E are partitions of S. Of course, every partition is also a covering. The set B is neither a partition nor a covering of S. The partition D has only one block while E has three. In the case of the given set S, we cannot have more than three blocks in any partition. In fact, for any finite set, the smallest partition consists of the set itself as a block while the largest partition consists of blocks containing only single elements.

Two partitions are said to be equal if they are equal as sets. For a finite set, every partition is a finite partition, i.e., every partition contains only a finite number of blocks.

It will be shown in Sec. 2-3.5 that an equivalence relation on a set partitions the set. Another relation on a set known as a compatibility relation as described in Sec. 2-3.6 defines certain coverings of the set.

Now we discuss some partitions of the universal set E which are generated by the subsets of E. Let us first consider a subset A of E. The subsets A and $\sim A$ generate a partition of E (see Fig. 2-3.11*a*) since

$$E = A \cup \sim A$$

Next let A and B be any two subsets of E, and consider the sets

$$I_0 = \sim A \cap \sim B \qquad I_1 = \sim A \cap B \qquad I_2 = A \cap \sim B \qquad \text{and} \qquad I_3 = A \cap B$$

The sets I_0, I_1, I_2, and I_3 are called the complete intersections or the *minterms* generated by the subsets A and B. It is easy to see that I_0, I_1, I_2, and I_3 are mutually disjoint and

$$E = I_0 \cup I_1 \cup I_2 \cup I_3 = \bigcup_{j=0}^{3} I_j$$

The complete intersections or the minterms are the blocks of a partition of E generated by A and B (see Fig. 2-3.11*b*).

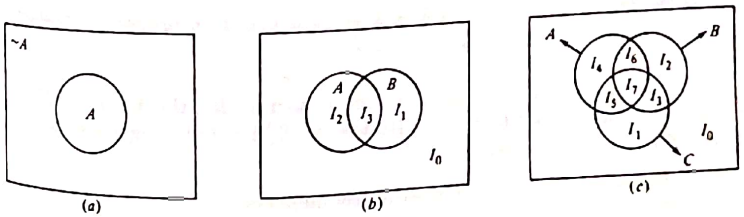


FIGURE 2-3.11 Complete intersections.

Scanned with OKEN Scanner

Let A, B, and C be three subsets of E and let the 2^3 minterms, denoted by $I_0, I_1, \ldots, I_7$ (see Fig. 2-3.11*c*), be as follows:

$$I_0 = \sim A \cap \sim B \cap \sim C \qquad I_1 = \sim A \cap \sim B \cap C$$
$$I_2 = \sim A \cap B \cap \sim C \qquad I_3 = \sim A \cap B \cap C$$
$$I_4 = A \cap \sim B \cap \sim C \qquad I_5 = A \cap \sim B \cap C$$
$$I_6 = A \cap B \cap \sim C \qquad I_7 = A \cap B \cap C$$

The subscript of I shows indirectly the minterm under consideration. In order to obtain the minterm, first we write the subscript as a binary integer containing three digits (since there are three subsets under consideration). The appearance of 1 or 0 in the first position on the left indicates the presence of A or $\sim A$, respectively. This relation also holds for the second and third positions. The notation is similar to the one used in Secs. 1-3.5 and 2-1.3. For example, $I_4 = A \cap \sim B \cap C$ since 5 written as a binary integer is 101.

In general, if $A_1, A_2, \ldots, A_n$ are any n subsets of the universal set E, then the complete intersections or minterms generated by these n subsets are denoted by $I_0, I_1, \ldots, I_{2^n-1}$ (see Sec. 2-1.3). These are mutually disjoint and are such that

$$E = \bigcup_{j=0}^{2^n-1} I_j$$

One can recognize a similarity between the minterms defined here and those given in the statement calculus. We shall return to a general discussion of this in Chap. 4.

EXERCISES 2-3.4

1 Define a well-formed formula of set theory in the same manner as in the definition given in Sec. 1-2.7, using the operators $\cap$, $\cup$, and $\sim$ only.
2 Show that for any formula in set theory involving set variables A and B and the operations $\cap$, $\cup$, and $\sim$, one can obtain another formula which is equal to the given formula and which contains the union of minterms only.
3 Show that the set of operations $\{\cup, \sim\}$ is functionally complete for formulas in set theory (*Hint*: Follow the same procedure used in Sec. 1-2.13).
4 Write the duals of minterms and discuss some of their important properties.

2-3.5 Equivalence Relations

Definition 2-3.9 A relation R in a set X is called an *equivalence relation* if it is reflexive, symmetric, and transitive.

If R is an equivalence relation in a set X, then $D(R)$, the domain of R, is X itself. Therefore R will be called a relation on X. The following are some examples of equivalence relations.

1 Equality of numbers on a set of real numbers
2 Equality of subsets of a universal set
3 Similarity of triangles on the set of triangles
4 Relation of lines being parallel on a set of lines in a plane
5 Relation of living in the same town on the set of persons living in Canada
6 Relation of statements being equivalent in the set of statements

EXAMPLE 1 Let $X = \{1, 2, 3, 4\}$ and

$$R = \{\langle 1, 1\rangle, \langle 1, 4\rangle, \langle 4, 1\rangle, \langle 4, 4\rangle, \langle 2, 2\rangle, \langle 2, 3\rangle, \langle 3, 2\rangle, \langle 3, 3\rangle\}$$

Write the matrix of R and sketch its graph.

SOLUTION The matrix and the graph of R are given in Fig. 2-3.12. It is clear that R is an equivalence relation. ////

EXAMPLE 2 Let $X = \{1, 2, \ldots, 7\}$ and

$$R = \{\langle x, y\rangle \mid x - y \text{ is divisible by } 3\}$$

Show that R is an equivalence relation. Draw the graph of R.

SOLUTION See Fig. 2-3.13. One can see from the figure that R is an equivalence relation. It is possible to prove this statement without using the graph of the relation in the following manner:

1 For any $a \in X$, $a - a$ is divisible by 3; hence $a\,R\,a$, or R is reflexive.
2 For any $a, b \in X$, if $a - b$ is divisible by 3, then $b - a$ is also divisible by 3; that is, $a\,R\,b \Rightarrow b\,R\,a$. Thus R is symmetric.

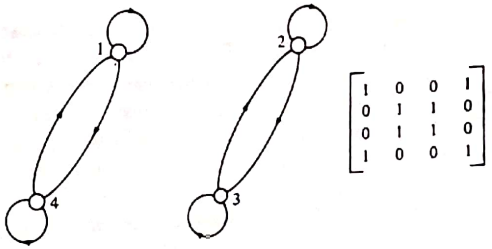


FIGURE 2-3.12 An equivalence relation

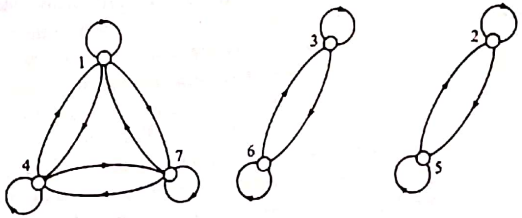


FIGURE 2-3.13

3 For $a, b, c \in X$, if $a \, R \, b$ and $b \, R \, c$, then both $a - b$ and $b - c$ are divisible by 3, so that $a - c = (a - b) + (b - c)$ is also divisible by 3, and hence $a \, R \, c$. Thus R is transitive. ////

Example 2 is a special case of a more general relation of equality in the modular number system. Let I denote the set of all positive integers, and let m be a positive integer. For $x \in I$ and $y \in I$, define R as

$$R = \{\langle x, y \rangle \mid x - y \text{ is divisible by } m\}$$

Note that "$x - y$ is divisible by m" is equivalent to the statement that both x and y have the same remainder when each is divided by m. It is customary to denote R by $\equiv$ and to write $x \, R \, y$ as $x \equiv y \ (m)$ or $x \equiv y \pmod{m}$, which is read as "x equals y modulo m." The relation $\equiv$ is also called a *congruence relation*. We define congruence relations in Sec. 3-1.2.

Definition 2-3.10 Let R be an equivalence relation on a set X. For any $x \in X$, the set $[x]_R \subseteq X$ given by

$$[x]_R = \{y \mid y \in X \wedge x \, R \, y\}$$

is called an *R-equivalence class* generated by $x \in X$.

Accordingly, the set $[x]_R$ consists of all the R-relatives of x in the set X. Sometimes $[x]_R$ is also written as x/R. We shall now study some properties of the equivalence classes generated by the elements of X.

1 For any element $x \in X$, we have $x \, R \, x$ because R is reflexive; therefore $x \in [x]_R$.

2 Let $y \in X$ be any other element such that $x \, R \, y$, so that $y \in [x]_R$. Because of the symmetry of R, $y \, R \, x$ and $x \in [y]_R$. Now, if there is an element $z \in [y]_R$, then z must be in $[x]_R$ because $y \, R \, z$, along with $x \, R \, y$, implies $x \, R \, z$. Thus $[y]_R \subseteq [x]_R$. By symmetry we must also have $[x]_R \subseteq [y]_R$. Finally, from $[y]_R \subseteq [x]_R$ and $[x]_R \subseteq [y]_R$, we have $[x]_R = [y]_R$.

3 In step 2 it is shown that if $x \, R \, y$, then $[x]_R = [y]_R$. We now show that if $x \not R \, y$, then $[x]_R$ and $[y]_R$ must be disjoint. This demonstration can be done by assuming that there is at least one element $z \in [x]_R$ and also $z \in [y]_R$; that is, $z \, R \, z$ and $y \, R \, z$, but this would imply $z \, R \, y$, and then from transitivity, $x \, R \, y$, which is a contradiction.

The above result shows that the R-equivalence class generated by any element $y \in X$ is equal to the R-equivalence class generated by $x \in X$ provided that $y \in [x]_R$. Otherwise the R-equivalence classes generated by x and y are disjoint. Further, each element of X generates an R-equivalence class which is nonempty. Therefore the R-equivalence classes generated by the elements of X cover X, that is, their union is the set X. Since the R-equivalence classes generated by any two elements are either equal or disjoint, we can say that the family of R-equivalence classes generated by the elements of X defines a partition of X. Such a partition is unique because an R-equivalence class of any element of X is unique. We now formulate this idea as a theorem.

Theorem 2-3.1 Every equivalence relation on a set generates a unique partition of the set. The blocks of this partition correspond to the R-equivalence classes.

As we have denoted the R-equivalence class generated by an element $x \in X$ by $[x]_R$, or x/R, we shall denote the family of equivalence classes by X/R, which is also written as X modulo R, or in short as X mod R. X/R is called the *quotient set* of X by R. Note that the elements of X/R are the equivalence classes which are themselves sets. They are, in fact, subsets of X or elements of the power set $\rho(X)$.

We consider now two special equivalence relations on a set X. The first such relation is $R_1 = X \times X$, and every element of X is in R_1-relation to all the elements of X. In this case the quotient set of X by R_1 is the set $\{X\}$. The other relation R_2 is such that every element of X is related to itself and to no other element. Such a relation is called an *identity relation*. An identity relation is an equivalence relation, and the quotient set of X by R_2 consists of sets which each contain a single element. Thus R_2 generates the largest partition of X.

EXAMPLE 3 Let $\mathbf{Z}$ be the set of integers and let R be the relation called "congruence modulo 3" defined by

$$R = \{\langle x, y \rangle \mid x \in \mathbf{Z} \wedge y \in \mathbf{Z} \wedge (x - y) \text{ is divisible by } 3\}$$

Determine the equivalence classes generated by the elements of $\mathbf{Z}$.

SOLUTION The equivalence classes are

$$[0]_R = \{\ldots, -6, -3, 0, 3, 6, \ldots\}$$
$$[1]_R = \{\ldots, -5, -2, 1, 4, 7, \ldots\}$$
$$[2]_R = \{\ldots, -4, -1, 2, 5, 8 \ldots\}$$
$$\mathbf{Z}/R = \{[0]_R, [1]_R, [2]_R\}$$ ////

In a similar manner one can find the equivalence classes generated by a relation "congruence modulo m" for any integer m.

EXAMPLE 4 Let S be the set of all statement functions in n variables and let R be the relation given by

$$R = \{\langle x \ y \rangle \mid x \in S \wedge y \in S \wedge x \Leftrightarrow y\}$$

Discuss the equivalence classes generated by the elements of S.

SOLUTION The number of possible distinct truth tables for statement functions which depend upon n statement variables is 2^{2^n} (see Sec. 1-2.12). Thus there are 2^{2^n} R-equivalence classes generated by the elements of S. ////

So far we have considered the partition of a set generated by an equivalence relation. Now we shall show that the converse of Theorem 2-3.1 also holds, i.e., if we start with a definite partition, say C, of a given set X, then we can define an equivalence relation which corresponds to this partition. For any $x \in X$,

Scanned with OKEN Scanner

there is a set $C_1 \in C$ such that $x \in C_1$; also x does not belong to any other ... ment of C. We now take all the elements of $C_1 \times C_1$ as members of a relation ... Thus every element of X that is in C_1 is an R-relative of every other ... of C_1. Furthermore, no other member of X which is not in C_1 is related to the ... ments of C_1. Similarly, for every other member of the partition C, we form ... bers of the relation R. If $C = \{C_1, C_2, C_3, \ldots, C_m\}$, then $R = (C_1 \times C_1) \cup (C_2 \times C_2) \cup \cdots \cup (C_m \times C_m)$. It is easy to see that R is an equivalence relation. Thus for every partition C we can define an equivalence relation.

EXAMPLE 5 Let $X = \{a, b, c, d, e\}$ and let $C = \{\{a, b\}, \{c\}, \{d, e\}\}$. Show that the partition C defines an equivalence relation on X.

SOLUTION

$$R = \{\langle a, a\rangle, \langle b, b\rangle, \langle a, b\rangle, \langle b, a\rangle, \langle c, c\rangle, \langle d, d\rangle, \langle e, e\rangle, \langle d, e\rangle, \langle e, d\rangle\}$$

////

It has been shown that an equivalence relation on a set generates a partition of the set, and conversely. It may happen that two relations, which may have been defined in different ways, generate the same partition. Since a relation is a set, any two relations consisting of equal sets are indistinguishable for our purpose. This statement will be true of every partition of the set as well. The following serves as an illustration.

Let $X = \{1, 2, \ldots, 9\}$ and $R_1 = \{\langle x, y\rangle \mid x \in X \wedge y \in X \wedge (x - y)$ is divisible by 3$\}$. Further, let

$R_2 = \{\langle x, y\rangle \mid x \in X \wedge y \in X$ and x, y are in same column of matrix A$\}$

where

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Although R_1 and R_2 have been defined differently, $R_1 = R_2$.

In Sec. 2-3.3 we have already given algorithms to determine whether a given relation R on a set is an equivalence relation. Once it is determined, our next task is to obtain the equivalence classes. Before giving an algorithm for this purpose, let us discuss the technique that will be used for the representation of the equivalence classes in the algorithm.

Given a set $\{1, 2, \ldots, n\}$ and an equivalence relation R on it, the equivalence classes can be represented by means of two vectors, each having n elements. These vectors are called *FIRST* and *MEMBER*. The ith component of *FIRST* for $1 \leq i \leq n$ contains the number which is the first element in the equivalence class to which i belongs. The ith component of *MEMBER* contains the number which follows i in the equivalence class, unless i is the last element, in which case *MEMBER*$[i]$ is equal to zero.

As an example, let the set be $\{1, 2, 3, 4, 5, 6\}$ and the equivalence classes be $\{1, 3, 6\}$, $\{2\}$, and $\{4, 5\}$. The vectors *FIRST* and *MEMBER* representing these equivalence classes are shown in Fig. 2-3.14.

	FIRST		MEMBER
1	1	1	3
2	2	2	0
3	1	3	6
4	4	4	5
5	4	5	0
6	1	6	0

FIGURE 2-3.14 Vector representation of equivalence classes.

Note that the vector structure employed here is useful in several ways. For example, if we wish to know whether 1 and 3 are in the same class, then the answer is yes because *FIRST*[1] is equal to *FIRST*[3]. Also, to obtain the number of equivalence classes, we simply count the zero elements in *MEMBER*.

The following algorithm will set up vectors *FIRST* and *MEMBER*.

Algorithm *CLASS* Given a relation matrix R representing an equivalence relation in a set of integers $\{1, 2, \ldots, n\}$, it is required to set up the *FIRST* and *MEMBER* vectors as described. All elements of vector *FIRST* are initially zero. The vector *OUTPUT* will be used to print the members of an equivalence class.

1 [Initialize row counter.] Set $i \leftarrow 0$ (i is an index to the rows of R and the vector *FIRST*).

2 [Scan next row.] Set $i \leftarrow i + 1$. If $i > n$ then Exit.

3 [Is i in a previous class?] If *FIRST*$[i] \neq 0$ then go to step 2; otherwise set $k \leftarrow i$ and $m \leftarrow 0$ (k is the previous element found which belongs to an equivalence class, and m is the index to the *OUTPUT* vector).

4 [Scan row starting at diagonal.] Repeat steps 5 and 6 for $j = i, i + 1, \ldots, n$.

5 [All members found?] If $j > n$ then *MEMBER*$[k] \leftarrow 0$, print class members from *OUTPUT*, and go to step 2.

6 [Is i related to j?] If $R[i, j] = T$ then *MEMBER*$[k] \leftarrow j$, *FIRST*$[j] \leftarrow i$, $k \leftarrow j$, $m \leftarrow m + 1$, and *OUTPUT*$[m] \leftarrow j$.

In this algorithm, the vector *FIRST* is scanned for zero elements. When one is found, the corresponding ith row of the relation matrix is scanned, starting with the ith column, for entries with truth value T. For each such entry in the jth column, *FIRST*$[j]$ is set to i, j is put in the *OUTPUT* vector, and *MEMBER*$[k]$ is set to j. Here k indicates the column subscript of the preceding true entry. In the case when $j = i$, k is also equal to i, and therefore a discrepancy arises. That is, *MEMBER*$[k] \leftarrow j$ means *MEMBER*$[i] \leftarrow i$, but this problem is resolved when the next j is found since k remains equal to i in the assignment $k \leftarrow j$. When all columns have been scanned, the members of an equivalence class are obtained as output, and *MEMBER*$[k]$ is set to zero, indicating k is the last element in the equivalence class. ////

Scanned with OKEN Scanner

```
      SUBROUTINE CLASS(A,N,FIRST,MEMBER)
C  THIS SUBROUTINE RECEIVES A RELATION MATRIX A WHICH REPRESENTS AN
C  EQUIVALENCE RELATION.  THE ARRAYS FIRST AND MEMBER ARE SET UP TO
C  REPRESENT THE EQUIVALENCE CLASSES.  ARRAY FIRST INITIALLY CONTAINS
C  ZEROS.  THE EQUIVALENCE CLASSES ARE PRINTED WITH THE AID OF ARRAY
C  OUTPUT.
C
      LOGICAL*1 A(N,N)
      INTEGER*2 FIRST(N),MEMBER(N),OUTPUT(10)
C
C  PRINT HEADING.
      WRITE(6,98)
   98 FORMAT('-',20X,'EQUIVALENCE CLASSES')
C
C  I IS THE ROW COUNTER.
      DO 20 I = 1,N
C
C  IF I IS ALREADY IN AN EQUIVALENCE CLASS, SKIP THE ITH ROW.
      IF(FIRST(I) .NE. 0) GO TO 20
C
C  K IS THE LAST NUMBER FOUND IN THE EQUIVALENCE CLASS.
C  J IS THE COLUMN COUNTER.
C  M COUNTS THE ELEMENTS OF THE EQUIVALENCE CLASS.
      K = I
      M = 0
      DO 10 J = I,N
C
C  FIND A J WHICH IS RELATED TO I.
      IF(.NOT.A(I,J)) GO TO 10
C
C  I IS THE FIRST NUMBER IN THE EQUIVALENCE CLASS TO WHICH J BELONGS.
C  J IS THE NEXT MEMBER OF THE CLASS TO WHICH K BELONGS.
      FIRST(J) = I
      MEMBER(K) = J
      K = J
      M = M + 1
      OUTPUT(M) = J
   10 CONTINUE
C
C  AT THIS POINT, ALL MEMBERS OF AN EQUIVALENCE CLASS HAVE BEEN FOUND.
C  PRINT THESE MEMBERS.
      MEMBER(K) = 0
      WRITE(6,99)(OUTPUT(L),L = 1,M)
   99 FORMAT('0',20X,10(I4))
   20 CONTINUE
      RETURN
      END
```

FIGURE 2-3.15 FORTRAN subroutine for algorithm *CLASS*.

A FORTRAN subroutine for *CLASS* is given in Fig. 2-3.15.

We now give a mainline program in Fig. 2-3.16*a* which first reads each ordered pair of a relation from one data card. The end of the data is signaled by the first element of an ordered pair having a value zero. The remainder of the mainline program uses the subprograms REFSYM, TRANS, and CLASS given here and in Sec. 2-3.3.

Given a relation R on the set $\{1, 2, \ldots, 10\}$ defined by

$$1\,R\,1, 2\,R\,2, 3\,R\,3, 4\,R\,4, 5\,R\,5, 6\,R\,6, 7\,R\,7, 8\,R\,8, 9\,R\,9, 10\,R\,10,$$

$$1\,R\,2, 1\,R\,8, 2\,R\,1, 2\,R\,8, 4\,R\,6, 6\,R\,4, 6\,R\,7, 7\,R\,6, 7\,R\,4, 4\,R\,7,$$

$$8\,R\,1, 8\,R\,2$$

the output obtained is shown in Fig. 2-3.16*b*.

2-3.6 Compatibility Relations

Definition 2-3.11 A relation R in X is said to be a *compatibility relation* if it is reflexive and symmetric.

Obviously all equivalence relations are compatibility relations. We shall, however, be concerned with those compatibility relations which are not equivalence relations. The following is an example of a compatibility relation.

Let $X = \{\text{ball, bed, dog, let, egg}\}$, and let the relation R be given by

$$R = \{\langle x, y\rangle \mid x, y \in X \wedge x\,R\,y \text{ if } x \text{ and } y \text{ contain some common letter}\}$$

```
C  MAINLINE
C  THIS PROGRAM READS ORDERED PAIRS OF NUMBERS AND PUTS THEM INTO A
C  RELATION MATRIX REPRESENTED BY THE LOGICAL ARRAY,A.  IT IS THEN
C  DETERMINED IF THE RELATION REPRESENTED BY A IS AN EQUIVALENCE
C  RELATION USING SUBROUTINES REFSYM AND TRANS WHICH TEST FOR THE
C  REFLEXIVE, SYMMETRIC, AND TRANSITIVE PROPERTIES.  IF IT IS AN
C  EQUIVALENCE RELATION, THE EQUIVALENCE CLASSES ARE PRINTED AND PUT
C  INTO A STRUCTURE CONSISTING OF ARRAYS FIRST AND MEMBER.
C  THE RELATION IS ON THE SET OF INTEGERS BETWEEN 1 AND 10 SO ALL ARRAY
C  DIMENSIONS WILL BE 10.
C
C     LOGICAL*1 A(10,10)/100*.FALSE./,FLAG/.FALSE./
C     INTEGER*2 FIRST(10)/10*0/,MEMBER(10)
C
C  READ THE ORDERED PAIRS AND REPRESENT THEM IN THE MATRIX.
C
   10 READ(5,98,END=20)I,J
   98 FORMAT(2I2)
      A(I,J) = .TRUE.
      GO TO 10
   20 N = 10
C
C  PRINT RELATION MATRIX.
      WRITE(6,97)(K,K = 1,N),(I,(A(I,J),J = 1,N),I = 1,N)
   97 FORMAT('-',31X,'RELATION MATRIX',/,'0',17X,10(I4),10(/,'0',15X,I2,
     1 10(L4)))
C
C  REFLEXIVE AND SYMMETRIC TEST.
      CALL REFSYM(A,N,FLAG)
C  IF FLAG IS STILL FALSE PRINT MESSAGE AND TERMINATE, OTHERWISE
C  TEST TRANSITIVITY.
      IF(.NOT.FLAG) GO TO 30
      FLAG = .FALSE.
      CALL TRANS(A,N,FLAG)
C
C  IF FLAG IS STILL FALSE TERMINATE.
      IF(.NOT.FLAG) GO TO 30
      WRITE(6,96)
   96 FORMAT('-',20X,'THE RELATION IS AN EQUIVALENCE RELATION.')
C
C  FIND THE EQUIVALENCE CLASSES.
      CALL CLASS(A,N,FIRST,MEMBER)
      WRITE(6,95)(FIRST(I),I = 1,10)
      WRITE(6,94)(MEMBER(I),I = 1,10)
   95 FORMAT('0',20X,'FIRST   ',10(I4))
   94 FORMAT('0',20X,'MEMBER  ',10(I4))
      STOP
C
C  THE RELATION IS NOT AN EQUIVALENCE RELATION.
   30 WRITE(6,93)
   93 FORMAT('-',20X,'THE RELATION IS NOT AN EQUIVALENCE RELATION.')
      STOP
      END
```

FIGURE 2-3.16*a* Main program and output for equivalence class program.

Scanned with OKEN Scanner

RELATION MATRIX

	1	2	3	4	5	6	7	8	9	10
1	T	T	F	F	F	F	F	T	F	F
2	T	T	F	F	F	F	F	T	F	F
3	F	F	T	F	F	F	F	F	F	F
4	F	F	F	T	F	T	T	F	F	F
5	F	F	F	F	T	F	F	F	F	F
6	F	F	F	T	F	T	T	F	F	F
7	F	F	F	T	F	T	T	F	F	F
8	T	T	F	F	F	F	F	T	F	F
9	F	F	F	F	F	F	F	F	T	F
10	F	F	F	F	F	F	F	F	F	T

THE RELATION IS AN EQUIVALENCE RELATION.

EQUIVALENCE CLASSES

1	2	8							
3									
4	6	7							
5									
9									
10									
FIRST	1	1	3	4	5	4	4	1	9	10
MEMBER	2	8	0	6	0	7	0	0	0	0

FIGURE 2-3.16*b*.

Then R is a compatibility relation, and x, y are called compatible if $x\,R\,y$. A compatibility relation is sometimes denoted by $\approx$. Note that ball $\approx$ bed, bed $\approx$ egg, but ball $\not\approx$ egg. Thus $\approx$ is not transitive. Denoting "ball" by x_1, "bed" by x_2, "dog" by x_3, "let" by x_4, and "egg" by x_5, the graph of $\approx$ is given in Fig. 2-3.17*a*.

Since $\approx$ is a compatibility relation, it is not necessary to draw the loops at each element nor is it necessary to draw both $x\,R\,y$ and $y\,R\,x$. Thus we can simplify the graph of $\approx$, as shown in Fig. 2-3.17*b*. Note that the elements in each of the sets $\{x_1, x_2, x_4\}$ and $\{x_2, x_3, x_5\}$ are related to each other, i.e., the elements are mutually compatible. Further, these two sets define a covering of X. The set $\{x_2, x_4, x_5\}$ also has elements compatible to each other.

The relation matrix of a compatibility relation is symmetric and has its diagonal elements unity. It is, therefore, sufficient to give only the elements of the lower triangular part of the relation matrix in such a case. For the compatibility relation we have been discussing, the relation matrix can be obtained from Table 2-3.2.

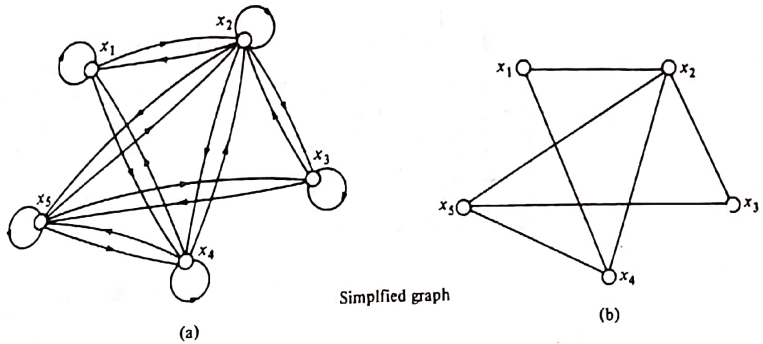


FIGURE 2-3.17 Graphs of compatibility relation $\approx$.

Although an equivalence relation on a set defines a partition of the set into equivalence classes, a compatibility relation does not necessarily define a partition. However, a compatibility relation does define a covering of the set.

Definition 2-3.12 Let X be a set and $\approx$ a compatibility relation on X. A subset $A \subseteq X$ is called a *maximal compatibility block* if any element of A is compatible to every other element of A and no element of $X - A$ is compatible to all the elements of A.

It is clear from Fig. 2-3.17*b* that the subset $\{x_1, x_2, x_4\}$ is a maximal compatibility block; so, too, are the subsets $\{x_2, x_3, x_5\}$ and $\{x_2, x_4, x_5\}$. These sets are not mutually disjoint, and therefore they only define a covering of X.

To find the maximal compatibility blocks corresponding to a compatibility relation on a set X, first we draw a simplified graph of the compatibility relation and pick from this graph the largest complete polygons. By a "largest complete polygon" we mean a polygon in which any vertex is connected to every other vertex. For example, a triangle is always a complete polygon, but for a quadrilateral to be a complete polygon we must have the two diagonals present. In addition to these examples, any element of the set which is related only to itself forms a maximal compatibility block. Similarly, any two elements which are compatible to one another but to no other elements also form a maximal compatibility block. We now give some graphs of compatibility relations, the corresponding relation matrices, and the maximal compatibility blocks.

The maximal compatibility blocks of the relations shown in Figs. 2-3.18

Table 2-3.2

x_2	1			
x_3	0	1		
x_4	1	1	0	
x_5	0	1	1	1
	x_1	x_2	x_3	x_4

Scanned with OKEN Scanner

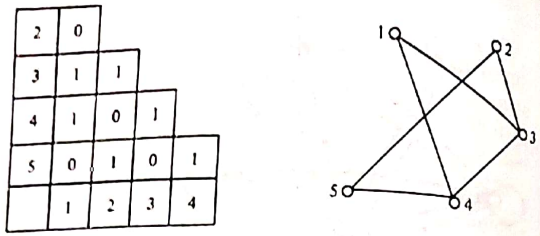


FIGURE 2-3.18

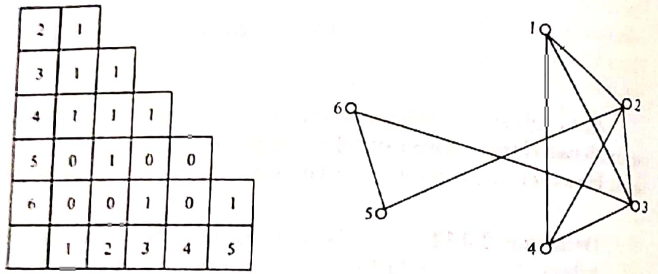


FIGURE 2-3.19

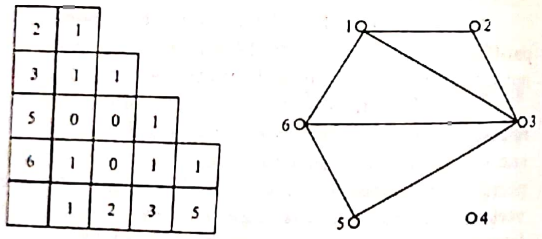


FIGURE 2-3.20

to 2-3.20 are given by

$$\{1,3,4\} \quad \{2,3\} \quad \{4,5\} \quad \{2,5\}$$

and

$$\{1,2,3,4\} \quad \{2,5\} \quad \{3,6\} \quad \{5,6\}$$

$$\{1,2,3\} \quad \{1,3,6\} \quad \{3,5,6\} \quad \{4\}$$

respectively. For the compatibility relation of the example discussed earlier and given in Fig. 2-3.17, the maximal compatibility blocks are $\{x_1, x_2, x_4\}$, $\{x_2, x_3, x_5\}$, and $\{x_3, x_4, x_5\}$.

Another procedure for finding the maximal compatibility blocks from the table of the relation matrix can be described in the following manner. It is assumed that first a simplified table is obtained in which those elements which are only compatible to themselves are deleted, because they are in a maximal compatibility block by themselves and are in no other compatibility block. Such blocks are included in the list at the end (see Fig. 2-3.20)

1 Start in the rightmost column of the table and proceed to the left until a column containing at least one nonzero entry is encountered. List all the compatible pairs represented by the entries in that column.

2 Proceed left to the next column that contains at least one nonzero entry. If any element is compatible to all the members of some previously defined compatibility class, then add this element to that class. If a member is compatible to only some members of a previously defined class, then form a new class which includes all the members that are compatible. Next, list all the compatible pairs not included in any previously defined class.

3 Repeat step 2 until all the columns are considered.

The final sets of compatibility classes including those which are isolated elements constitute the maximal compatibility classes.

Compatibility relations are useful in solving certain minimization problems of switching theory, particularly for incompletely specified minimization problems.

EXERCISES 2-3.6

1 Let R denote a relation on the set of ordered pairs of positive integers such that $\langle x, y\rangle \, R \, \langle u, v\rangle$ iff $xv = yu$. Show that R is an equivalence relation.

2 Given a set $S = \{1, 2, 3, 4, 5\}$, find the equivalence relation on S which generates the partition $\{\overline{1, 2}, \overline{3}, \overline{4, 5}\}$. Draw the graph of the relation.

3 Prove that the relation "congruence modulo m" given by

$$\equiv \; = \{\langle x, y\rangle \mid x - y \text{ is divisible by } m\}$$

over the set of positive integers is an equivalence relation. Show also that if $x_1 \equiv y_1$ and $x_2 \equiv y_2$, then $(x_1 + x_2) \equiv (y_1 + y_2)$.

4 Given a covering of the set $S = \{A_1, A_2, \ldots, A_n\}$, show how we can write a compatibility relation which defines this covering.

5 Let the compatibility relation on a set $\{x_1, x_2, \ldots, x_6\}$ be given by the matrix

x_2	1				
x_3	1	1			
x_4	0	0	1		
x_5	0	0	1	1	
x_6	1	0	1	0	1
	x_1	x_2	x_3	x_4	x_5

Draw the graph and find the maximal compatibility blocks of the relation.

Scanned with OKEN Scanner

2-3.7 Composition of Binary Relations

Since a binary relation is a set of ordered pairs, the usual operations such as union, intersection, etc., on these sets produce other relations. This topic was discussed in Sec. 2-3.1. We shall now consider another operation on relations—relations which are formed in two or more stages. Familiar examples of such relations are the relation of being a nephew or a brother's or sister's son, the relation of an uncle or a father's or mother's brother, and the relation of being a grandfather which is a father's or mother's father. These relations can be produced in the following manner.

Definition 2-3.13 Let R be a relation from X to Y and S be a relation from Y to Z. Then a relation written as $R \circ S$ is called a *composite relation* of R and S where

$$R \circ S = \{\langle x, z\rangle \mid x \in X \wedge z \in Z \wedge (\exists y)(y \in Y \wedge \langle x, y\rangle \in R \wedge \langle y, z\rangle \in S\}$$

The operation of obtaining $R \circ S$ from R and S is called *composition* of relations.

Note that $R \circ S$ is empty if the intersection of the range of R and the domain of S is empty. $R \circ S$ is nonempty if there is at least one ordered pair $\langle x, y\rangle \in R$ such that the second member $y \in Y$ of the ordered pair is a first member in an ordered pair in S. For the relation $R \circ S$, the domain is a subset of X and the range is a subset of Z. In fact, the domain is a subset of the domain of R, and its range is a subset of the range of S. From the graphs of R and S one can easily construct the graph of $R \circ S$. As an example, see Fig. 2-3.21.

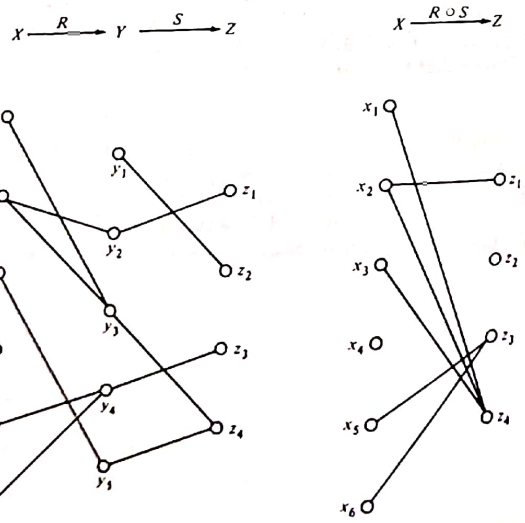


FIGURE 2-3.21 Relations R, S, and $R \circ S$.

The operation of composition is a binary operation on relations, and it produces a relation from two relations. The same operations can be applied again to produce other relations. For example, let R be a relation from X to Y, S a relation from Y to Z, and P a relation from Z to W. Then $R \circ S$ is a relation from X to Z. We can form $(R \circ S) \circ P$, which is a relation from X to W. Similarly, we can also form $R \circ (S \circ P)$, which again is a relation from X to W.

Let us assume that $(R \circ S) \circ P$ is nonempty, and let $\langle x, y\rangle \in R$, $\langle y, z\rangle \in S$, and $\langle z, w\rangle \in P$. This assumption means $\langle x, z\rangle \in R \circ S$ and $\langle x, w\rangle \in (R \circ S) \circ P$. Of course, $\langle y, w\rangle \in S \circ P$ and $\langle x, w\rangle \in R \circ (S \circ P)$, which shows that

$$(R \circ S) \circ P = R \circ (S \circ P)$$

This result can be stated by saying that the operation of composition on relations is associative. We may delete the parentheses in writing $(R \circ S) \circ P$, so that

$$(R \circ S) \circ P = R \circ (S \circ P) = R \circ S \circ P$$

The same result follows from the partial graph given in Fig. 2-3.22.

EXAMPLE 1 Let $R = \{\langle 1, 2\rangle, \langle 3, 4\rangle, \langle 2, 2\rangle\}$ and $S = \{\langle 4, 2\rangle, \langle 2, 5\rangle, \langle 3, 1\rangle, \langle 1, 3\rangle\}$. Find $R \circ S$, $S \circ R$, $R \circ (S \circ R)$, $(R \circ S) \circ R$, $R \circ R$, $S \circ S$, and $R \circ R \circ R$.

SOLUTION

$$R \circ S = \{\langle 1, 5\rangle, \langle 3, 2\rangle, \langle 2, 5\rangle\}$$

$$S \circ R = \{\langle 4, 2\rangle, \langle 3, 2\rangle, \langle 1, 4\rangle\} \neq R \circ S$$

$$(R \circ S) \circ R = \{\langle 3, 2\rangle\}$$

$$R \circ (S \circ R) = \{\langle 3, 2\rangle\} = (R \circ S) \circ R$$

$$R \circ R = \{\langle 1, 2\rangle, \langle 2, 2\rangle\}$$

$$S \circ S = \{\langle 4, 5\rangle, \langle 3, 3\rangle, \langle 1, 1\rangle\}$$

$$R \circ R \circ R = \{\langle 1, 2\rangle, \langle 2, 2\rangle\}$$

////

EXAMPLE 2 Let R and S be two relations on a set of positive integers $\mathbf{I}$:

$$R = \{\langle x, 2x\rangle \mid x \in \mathbf{I}\} \qquad S = \{\langle x, 7x\rangle \mid x \in \mathbf{I}\}$$

Find $R \circ S$, $R \circ R$, $R \circ R \circ R$, and $R \circ S \circ R$.

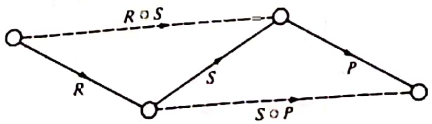


FIGURE 2-3.22 Associativity of composition.

SOLUTION

$$R \circ S = \{\langle x, 14x\rangle \mid x \in \mathbf{I}\} = S \circ R$$
$$R \circ R = \{\langle x, 4x\rangle \mid x \in \mathbf{I}\}$$
$$R \circ R \circ R = \{\langle x, 8x\rangle \mid x \in \mathbf{I}\}$$
$$R \circ S \circ R = \{\langle x, 28x\rangle \mid x \in \mathbf{I}\}$$

////

We know that the relation matrix of a relation R from a set $X = \{x_1, x_2, \ldots, x_m\}$ to a set $Y = \{y_1, y_2, \ldots, y_n\}$ is given by a matrix having m rows and n columns. We shall denote the relation matrix of R by M_R. M_R has entries which are 1s and 0s. Similarly the relation matrix M_S of a relation S from the set Y to a set $Z = \{z_1, z_2, \ldots, z_p\}$ is an $n \times p$ matrix. The relation matrix of $R \circ S$ can be obtained from the matrices M_R and M_S in the following manner.

From the definition it is clear that $\langle x_i, z_k\rangle \in R \circ S$ if there is at least one element of Y, say y_j, such that $\langle x_i, y_j\rangle \in R$ and $\langle y_j, z_k\rangle \in S$. There may be more than one element of Y which has properties similar to those of y_j; for example, $\langle x_i, y_r\rangle \in R$ and $\langle y_r, z_k\rangle \in S$. In all such cases, $\langle x_i, z_k\rangle \in R \circ S$. Thus when we scan the ith row of M_R and kth column of M_S and we come across at least one j, such that the entries in the jth location of the row as well as the column under consideration are 1s, then the entry in the ith row and kth column of $M_{R \circ S}$ is also 1; otherwise it is 0. Scanning a row of M_R along with every column of M_S gives one row of $M_{R \circ S}$. Similarly, we can obtain all the other rows.

EXAMPLE 3 For the relations R and S given in Example 1 over the set $\{1, 2, \ldots, 5\}$, obtain the relation matrices for $R \circ S$ and $S \circ R$.

SOLUTION

$$\begin{bmatrix} 0&1&0&0&0\\ 0&1&0&0&0\\ 0&0&0&1&0\\ 0&0&0&0&0\\ 0&0&0&0&0 \end{bmatrix} \cdot \begin{bmatrix} 0&0&1&0&0\\ 0&0&0&0&1\\ 1&0&0&0&0\\ 0&1&0&0&0\\ 0&0&0&0&0 \end{bmatrix} = \begin{bmatrix} 0&0&0&0&1\\ 0&0&0&0&1\\ 0&1&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0 \end{bmatrix}$$

M_R $\qquad$ M_S $\qquad$ $M_{R \circ S}$

$$\begin{bmatrix} 0&0&1&0&0\\ 0&0&0&0&1\\ 1&0&0&0&0\\ 0&1&0&0&0\\ 0&0&0&0&0 \end{bmatrix} \cdot \begin{bmatrix} 0&1&0&0&0\\ 0&1&0&0&0\\ 0&0&0&1&0\\ 0&0&0&0&0\\ 0&0&0&0&0 \end{bmatrix} = \begin{bmatrix} 0&0&0&1&0\\ 0&0&0&0&0\\ 0&1&0&0&0\\ 0&1&0&0&0\\ 0&0&0&0&0 \end{bmatrix}$$

M_S $\qquad$ M_R $\qquad$ $M_{S \circ R}$

In general, let the relations A and B be represented by $n \times m$ and $m \times r$ matrices respectively. Then the composition $A \cdot B$ which we denote by the relation matrix C is expressed as

$$c_{ij} = \bigvee_{k=1}^{m} a_{ik} \wedge b_{kj} \qquad i = 1, 2, \ldots, n; j = 1, 2, \ldots, r$$

where $a_{ik} \wedge b_{kj}$ and $\bigvee_{k=1}^{m}$ indicate bit-ANDing and bit-ORing respectively (see Sec. 2-2.5).

Definition 2-3.14 Given a relation R from X to Y, a relation $\tilde{R}$ from Y to X is called the *converse* of R, where the ordered pairs of $\tilde{R}$ are obtained by interchanging the members in each of the ordered pairs of R. This means, for $x \in X$ and $y \in Y$, that $x\,R\,y \Leftrightarrow y\,\tilde{R}\,x$.

From the definition of $\tilde{R}$ it follows that $\tilde{\tilde{R}} = R$. The relation matrix $M_{\tilde{R}}$ of $\tilde{R}$ can be obtained by simply interchanging the rows and columns of M_R. Such a matrix is called the *transpose* of M_R. Therefore

$$M_{\tilde{R}} = \text{transpose of } M_R$$

The graph of $\tilde{R}$ is also obtained from that of R by simply reversing the arrows on each arc.

We shall now consider the converse of a composite relation. For this purpose, let R be a relation from X to Y and S be a relation from Y to Z. Obviously, $\tilde{R}$ is a relation from Y to X, $\tilde{S}$ from Z to Y; $R \circ S$ is a relation from X to Z, and $R\,\bar{\circ}\,S$ is a relation from Z to X. Also the relation $\tilde{S} \circ \tilde{R}$ is from Z to X. We now show that

$$R\,\bar{\circ}\,S = \tilde{S} \circ \tilde{R}$$

If $x\,R\,y$ and $y\,S\,z$, then $x\,(R \circ S)\,z$ and $z\,(R\,\bar{\circ}\,S)\,x$. But $z\,\tilde{S}\,y$ and $y\,\tilde{R}\,x$, so that $z\,(\tilde{S} \circ \tilde{R})\,x$. This is true for any $x \in X$ and $z \in Z$; hence the required result.

The same rule can be expressed in terms of the relation matrices by saying that the transpose of $M_{R \circ S}$ is the same as the matrix $M_{\tilde{S} \circ \tilde{R}}$. The matrix $M_{\tilde{S} \circ \tilde{R}}$ can be obtained from the matrices $M_{\tilde{S}}$ and $M_{\tilde{R}}$, which in turn can be obtained from the matrices M_S and M_R.

EXAMPLE 4 Given the relation matrices M_R and M_S, find $M_{R \circ S}$, $M_{\tilde{R}}$, $M_{\tilde{S}}$, $M_{R\bar{\circ}S}$, and show that $M_{R\bar{\circ}S} = M_{\tilde{S} \circ \tilde{R}}$.

$$M_R = \begin{bmatrix} 1&0&1\\ 1&1&0\\ 1&1&1 \end{bmatrix} \qquad M_S = \begin{bmatrix} 1&0&0&1&0\\ 1&0&1&0&1\\ 0&1&0&1&0 \end{bmatrix}$$

Scanned with OKEN Scanner

SOLUTION

$$M_{\tilde{R}} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = \text{transpose of } M_R$$

$$M_{\tilde{S}} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \text{transpose of } M_S$$

$$M_{R \circ S} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad M_{R \circ S} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

$$M_{\tilde{S} \circ \tilde{R}} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} = M_{R \circ S}$$

////

The following hold for any relations R and S.

1 $\tilde{\tilde{R}} = R$
2 $R = S \Leftrightarrow \tilde{R} = \tilde{S}$
3 $R \subseteq S \Leftrightarrow \tilde{R} \subseteq \tilde{S}$
4 $R \cup S = \tilde{R} \cup \tilde{S}$
5 $R \cap S = \tilde{R} \cap \tilde{S}$

We shall leave the proofs as exercises.

Let us now consider some distinct relations R_1, R_2, R_3, R_4 in a set $X = \{a, b, c\}$ given by

$$R_1 = \{\langle a, b\rangle, \langle a, c\rangle, \langle c, b\rangle\}$$
$$R_2 = \{\langle a, b\rangle, \langle b, c\rangle, \langle c, a\rangle\}$$
$$R_3 = \{\langle a, b\rangle, \langle b, c\rangle, \langle c, c\rangle\}$$
$$R_4 = \{\langle a, b\rangle, \langle b, a\rangle, \langle c, c\rangle\}$$

Denoting the composition of a relation by itself as

$R \circ R = R^2 \quad R \circ R \circ R = R \circ R^2 = R^3 \quad \cdots \quad R \circ R^{m-1} = R^m \quad \cdots$

let us write the powers of the given relations. Clearly

$R_1^2 = \{\langle a, b\rangle\} \quad R_1^3 = \varnothing \quad R_1^4 = \varnothing \quad \cdots$

$R_2^2 = \{\langle a, c\rangle, \langle b, a\rangle, \langle c, b\rangle\} \quad R_2^3 = \{\langle a, a\rangle, \langle b, b\rangle, \langle c, c\rangle\}$

$R_2^4 = R_2 \quad R_2^5 = R_2^2 \quad R_2^6 = R_2^3 \quad \cdots$

$R_3^2 = \{\langle a, c\rangle, \langle b, c\rangle, \langle c, c\rangle\} = R_3^3 = R_3^4 = R_3^5 \cdots$

$R_4^2 = \{\langle a, a\rangle, \langle b, b\rangle, \langle c, c\rangle\} \quad R_4^3 = R_4 \quad R_4^5 = R_4^2 \quad \cdots$

Given a finite set X, containing n elements, and a relation R in X, we can interpret R^m ($m = 1, 2, \ldots$) in terms of its graph. This interpretation is done for a number of applications in Chap. 5. With the help of such an interpretation or from the examples given here, it is possible to say that there are at most n distinct powers of R, for R^m, $m > n$, can be expressed in terms of $R, R^2, \ldots, R^n$. Our next step is to construct the relation in X given by

$$R^+ = R \cup R^2 \cup R^3 \cup \cdots$$

Naturally, this construction will require only a finite number of powers of R to be calculated, and these calculations can easily be performed by using the matrix representation of the relation R and the Boolean multiplication of these matrices. Let us now see what the corresponding relations R_1^+, R_2^+, R_3^+, and R_4^+ are

$R_1^+ = R_1 \cup R_1^2 \cup R_1^3 \cdots = R_1$

$R_2^+ = R_2 \cup R_2^2 \cup R_2^3 \cdots = R_2 \cup R_2^2 \cup R_2^3$

$= \{\langle a, b\rangle, \langle b, c\rangle, \langle c, a\rangle, \langle a, c\rangle, \langle b, a\rangle, \langle c, b\rangle, \langle a, a\rangle, \langle b, b\rangle, \langle c, c\rangle\}$

$R_3^+ = \{\langle a, b\rangle, \langle b, c\rangle, \langle c, c\rangle, \langle a, c\rangle\}$

$R_4^+ = \{\langle a, b\rangle, \langle b, a\rangle, \langle c, c\rangle, \langle a, a\rangle, \langle b, b\rangle\}$

Observe that the relations $R_1^+, R_2^+, \ldots, R_4^+$ are all transitive and that $R_1 \subseteq R_1^+$, $R_2 \subseteq R_2^+, \ldots, R_4 \subseteq R_4^+$. From the graphs of these relations one can easily see that R_i^+ is obtained from R_i ($i = 1, 2, 3, 4$) by adding only those ordered pairs to R_i such that R_i^+ is transitive. We now define R^+ in general.

Definition 2-3.15 Let X be any finite set and R be a relation in X. The relation $R^+ = R \cup R^2 \cup R^3 \cup \cdots$ in X is called the *transitive closure* of R in X.

Theorem 2-3.2 The transitive closure R^+ of a relation R in a finite set X is transitive. Also for any other transitive relation P in X such that $R \subseteq P$, we have $R^+ \subseteq P$. In this sense, R^+ is the smallest transitive relation containing R.

Scanned with OKEN Scanner

PROOF First, to show that R^+ is transitive, let us assume that $a\ R^+\ b$ and $b\ R^+\ c$ for some $a, b, c \in X$. Since $a\ R^+\ b$, we must have a sequence of elements $d_1, d_2, \ldots, d_k \in X$ such that $d_1 = a$, $d_k = b$, and $d_1\ R\ d_2$, $d_2\ R\ d_3$, $\ldots$, $d_{k-1}\ R\ d_k$. Here we have assumed that $a\ R^k\ b$ for some k. Similarly, since $b\ R^+\ c$, we must have a sequence of elements, say $e_1, e_2, \ldots, e_j$, such that $e_1 = b$, $e_j = c$, and $e_1\ R\ e_2$, $e_2\ R\ e_3$, $\ldots$, $e_{j-1}\ R\ e_j$. Here we have assumed that $b\ R^j\ c$ for some j. It follows from these assumptions that $a\ R^{k+j}\ c$, implying that $a\ R^+\ c$. Hence R^+ must be transitive.

Let us now assume that $a\ R^+\ b$ for some $a, b \in X$, so that there exists a sequence of elements $c_1, c_2, \ldots, c_m \in X$ such that $a = c_1$, $b = c_m$, and $c_i\ R\ c_{i+1}$ for $i = 1, 2, \ldots, m-1$. If there is a transitive relation P in X such that $R \subseteq P$, then $c_i\ P\ c_{i+1}$ for $i = 1, 2, \ldots, m-1$, so that $c_1\ P\ c_m$, that is, $a\ P\ b$. Since (a, b) is an arbitrary element of R^+, we see by the same argument that $R^+ \subseteq P$. Hence R^+ is the smallest transitive relation which includes R. ///

Transitive closures of relations have important applications in certain areas such as networks, syntactic analysis, fault detection and diagnosis in switching circuits, etc. A number of these applications are discussed in Chap. 5.

EXERCISES 2-3.7

1 Prove the equivalences and equalities (1) to (5) given at the end of the section (following Example 4).

2 Show that if a relation R is reflexive, then $\tilde{R}$ is also reflexive. Show also that similar remarks hold if R is transitive, irreflexive, symmetric, or antisymmetric.

3 What nonzero entries are there in the relation matrix of $R \cap \tilde{R}$ if R is an antisymmetric relation?

4 Let E be the identity relation on a set X and R be any relation in X; show that $E \cup R \cup \tilde{R}$ is a compatibility relation.

5 Given the relation matrix M_R of a relation R on the set $\{a, b, c\}$, find the relation matrices of $\tilde{R}$, $R^2 = R \circ R$, $R^3 = R \circ R \circ R$, and $R \circ \tilde{R}$.

$$M_R = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

6 Two equivalence relations R and S are given by their relation matrices M_R and M_S. Show that $R \circ S$ is not an equivalence relation.

$$M_R = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad M_S = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Obtain equivalence relations R_1 and R_2 on $\{1, 2, 3\}$ such that $R_1 \circ R_2$ is also an equivalence relation.

2-3.8 Partial Ordering

Definition 2-3.16 A binary relation R in a set P is called a *partial order relation* or a *partial ordering* in P iff R is reflexive, antisymmetric, and transitive.

It is conventional to denote a partial ordering by the symbol $\leq$. This symbol does not necessarily mean "less than or equal to" as is used for real numbers. Since the relation of partial ordering is reflexive, we shall henceforth call it a relation on a set, say P. If $\leq$ is a partial ordering on P, then the ordered pair $\langle P, \leq \rangle$ is called a *partially ordered set* or a *poset*.

Definition 2-3.17 Let $\langle P, \leq \rangle$ be a partially ordered set. If for every $x, y \in P$ we have either $x \leq y \vee y \leq x$, then $\leq$ is called a *simple ordering* or *linear ordering* on P, and $\langle P, \leq \rangle$ is called a *totally ordered* or *simply ordered set* or a *chain*.

Note that it is not necessary to have $x \leq y$ or $y \leq x$ for every x and y in a partially ordered set P. In fact, x may not be related to y, in which case we say that x and y are *incomparable*.

If R is a partial ordering on P, then it is easy to see that the converse of R, namely $\tilde{R}$, is also a partial ordering on P. If R is denoted by $\leq$, then $\tilde{R}$ is denoted by $\geq$. This means that if $\langle P, \leq \rangle$ is a partially ordered set, then $\langle P, \geq \rangle$ is also a partially ordered set. $\langle P, \geq \rangle$ is called the dual of $\langle P, \leq \rangle$.

We now define another relationship which is associated with every partial ordering $\leq$ on P and which is denoted by $<$. This relation $<$ is defined, for every $x, y \in P$, as

$$x < y \Leftrightarrow x \leq y \wedge x \neq y$$

Similarly, corresponding to the converse partial ordering $\geq$, there is a relation $>$ such that

$$x > y \Leftrightarrow x \geq y \wedge x \neq y$$

Note that the relations $<$ and $>$ are antisymmetric and transitive. In addition, these relations are irreflexive. We now give some partial order relations which are frequently used.

1 *Less Than or Equal to, Greater Than or Equal to*: Let R be the set of real numbers. The relation "*less than or equal to*," or $\leq$, is a partial ordering on R. The converse of this relation, "*greater than or equal to*," or $\geq$, is also a partial ordering on R. Associated relations are "*less than*," or $<$, and "*greater than*," or $>$, respectively

2 *Inclusion*: Let $\rho(A) = 2^A = X$ be the power set of A, that is, X is the set of subsets of A. The relation of inclusion ($\subseteq$) on X is a partial ordering. Associated with the relation $\subseteq$ is a relation called proper inclusion ($\subset$) which is irreflexive, antisymmetric, and transitive.

As a special case, we let $A = \{a, b, c\}$. Then

$$X = \rho(A) = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, A\}$$

Scanned with OKEN Scanner

It is easy to write the elements of the relation $\subseteq$. Note that $\{a\}$ and $\{b, c\}$, $\{a, b\}$ and $\{a, c\}$, etc., are incomparable.

3 *Divides and Integral Multiple*: If a and b are positive integers, then we say "a divides b," written $a \mid b$, iff there is an integer c such that $ac = b$. Alternatively, we say that "b is an *integral multiple of* a." The relation "divides" is a partial order relation. Let X be the set of positive integers. The relations "divides" and "integral multiple of" are partial orderings on X, and each is the converse of the other.

As a special case, let $X = \{2, 3, 6, 8\}$ and let $\leq$ be the relation "divides" on X. Then

$$\leq = \{\langle 2, 2\rangle, \langle 3, 3\rangle, \langle 6, 6\rangle, \langle 8, 8\rangle, \langle 2, 8\rangle, \langle 2, 6\rangle, \langle 3, 6\rangle\}$$

The relation "integral multiple of," written as $\geq$, is given by

$$\geq = \{\langle 2, 2\rangle, \langle 3, 3\rangle, \langle 6, 6\rangle, \langle 8, 8\rangle, \langle 8, 2\rangle, \langle 6, 2\rangle, \langle 6, 3\rangle\}$$

4 *Lexicographic Ordering*: A useful example of simple or total ordering is the lexicographic ordering. We shall define it for certain ordered pairs first and then generalize it.

Let R be the set of real numbers and let $P = R \times R$. The relation $\geq$ on R is assumed to be the usual relation of "greater than or equal to." For any two ordered pairs $\langle x_1, y_1\rangle$ and $\langle x_2, y_2\rangle$ in P, we define the total ordering relation S as follows:

$$\langle x_1, y_1\rangle \, S \, \langle x_2, y_2\rangle \Leftrightarrow (x_1 > x_2) \vee ((x_1 = x_2) \wedge (y_1 \geq y_2))$$

It is clear that if $\langle x_1, y_1\rangle \not S \langle x_2, y_2\rangle$, then we must have $\langle x_2, y_2\rangle \, S \, \langle x_1, y_1\rangle$, so that S is a total ordering on P. The partial ordering S is called the *lexicographic ordering*. The significance of the terminology will become clear after we generalize the above ordering relation. The following are some of the ordered pairs of P which are S-related:

$$\langle 2, 2\rangle \, S \, \langle 2, 1\rangle$$
$$\langle 3, 1\rangle \, S \, \langle 1, 5\rangle$$
$$\langle 2, 2\rangle \, S \, \langle 2, 2\rangle$$
$$\langle 3, 2\rangle \, S \, \langle 1, 1\rangle$$

We now generalize this concept. For this purpose, let R be a total ordering relation on a set X and let

$$P = X \cup X^2 \cup X^3 \cup \cdots \cup X^n = \bigcup_i X^i \qquad (n = 1, 2, 3, \ldots)$$

This equation means that the set P consists of strings of elements of X of length less than or equal to n. We may assume some fixed value of n. A string of length p may be considered as an ordered p-tuple. We now define a total ordering S on P called lexicographic ordering. For this purpose, let $\langle u_1, u_2, \ldots, u_p\rangle$ and $\langle v_1, v_2, \ldots, v_q\rangle$, with $p \leq q$, be any two elements of P. Note that before starting, to compare two strings to determine the ordering in P, the strings are interchanged if necessary so that $p \leq q$. Now

$$\langle u_1, u_2, \ldots, u_p\rangle \, S \, \langle v_1, v_2, \ldots, v_q\rangle$$

if any one of the following holds:

1 $\langle u_1, u_2, \ldots, u_p\rangle = \langle v_1, v_2, \ldots, v_p\rangle$
2 $u_1 \neq v_1$ and $u_1 \, R \, v_1$ in X
3 $u_i = v_i$, $i = 1, 2, \ldots, k$ $(k < p)$, and $u_{k+1} \neq v_{k+1}$ and $u_{k+1} \, R \, v_{k+1}$ in X

If none of these conditions is satisfied, then

$$\langle v_1, v_2, \ldots, v_q\rangle \, S \, \langle u_1, u_2, \ldots, u_p\rangle$$

As a special case of lexicographic ordering, let $X = \{a, b, c, \ldots, z\}$ and let R be a simple ordering on X denoted by $\leq$ where $a \leq b \leq c \leq \cdots \leq z$ and $P = X \cup X^2 \cup X^3$. Thus, P consists of all "words" or strings of 3 or fewer than 3 letters from X. Let S denote the lexicographic ordering on P described earlier. We will have

me S met	by condition 1
bet S met	by condition 2
beg S bet	by condition 3
get S go	by the last rule

since "go" and "get" are compared and the conditions 1, 2, and 3 are not satisfied.

The order in which the words in an English dictionary appear is a familiar example of lexicographic ordering. Instead of using S to denote the lexicographic ordering, it is customary to use names such as "lexically less than or equal to" or "lexically greater than."

We shall now describe how the lexicographic ordering is used in sorting character data on a computer. For this purpose, let X denote the set of characters available on a particular computer. It is necessary first to define a simple ordering on the elements of X (frequently called the collating sequence). One method is to compare the numeric values of the coded representation of each character in the computer by using the relation "less than or equal to." This ordering may vary from one computer to another. An example of a code which has such an ordering is the Extended Binary Coded Decimal Interchange Code (EBCDIC). In any case, we have a totally ordered set $\langle X, \leq\rangle$, and character strings are formed from the elements of X. Since blanks are also permitted to appear in such strings, a blank is treated as a character, i.e., an element of X. It is convenient to assume that a blank is less than all other elements of X. Not only do blanks appear inside a string, but sometimes it will be convenient to add blanks at the end of a string. It will be assumed that such additions do not alter the relative ordering of a string.

Now we consider how two given strings of equal length are compared for the purpose of ordering them lexicographically. If one string is shorter than the other, we simply assume that it is padded at the right end (because we shall assume the scanning is done from left to right) with the number of blanks sufficient that both strings to be compared are of equal length. In some cases, it may be necessary to distinguish between a given string and the one to which some blanks are added. This distinction can be made by comparing the strings for lexical equality and then comparing them according to their lengths.

Scanned with OKEN Scanner

The primitive algorithm *SORT* which follows is based on lexicographic comparisons.

Algorithm *SORT* Given a vector $NAME$ which contains m character strings, it is required to sort these strings into the proper lexicographic order. To assist in this process, the character string variable $TEMP$, the logical variable $FLAG$, and parameters i and j are used.

1 [Repeat for $m-1$ passes] Repeat steps 2 to 4 inclusive for $i = 1, 2, \ldots, m-1$ and then Exit.

2 [Scan to element $m-i$] Set $FLAG \leftarrow T$. Repeat step 3 for $j = 1, 2, \ldots, m-i$.

3 [Exchange required?] If $NAME[j] > NAME[j+1]$ then set $TEMP \leftarrow NAME[j]$, $NAME[j] \leftarrow NAME[j+1]$, $NAME[j+1] \leftarrow TEMP$ and $FLAG \leftarrow F$.

4 [Pass without exchange?] If $FLAG$ then Exit.

The purpose of this algorithm is to sort the m character sequences in $NAME$ into an ascending lexicographic order; that is, $NAME[j]$ must not be lexically greater than $NAME[j+1]$ for $j = 1, 2, \ldots, m-1$. When i is equal to 1, all adjacent elements are compared and those which are out of order are exchanged. When this comparison is done for elements j and $j+1$, with j ranging from 1 to $m-i$, in this specific case we are assured that element m will contain the lexically greatest string. When i is equal to 2, the pass will be completed with the second greatest string in position $m-1$ of $NAME$. A maximum of $m-1$ such passes are performed, and on the ith pass, $m-i$ comparisons must be made. Of course, on completion of a pass without any exchanges, all elements are in the proper order and the algorithm is complete, as indicated by $FLAG$. ////

2-3.9 Partially Ordered Set: Representation and Associated Terminology

In a partially ordered set $\langle P, \leq \rangle$, an element $y \in P$ is said to *cover* an element $x \in P$ if $x < y$ and if there does not exist any element $z \in P$ such that $x \leq z$ and $z \leq y$; that is,

$$y \text{ covers } x \Leftrightarrow (x < y \wedge (x \leq z \leq y \Rightarrow x = z \vee z = y))$$

Sometimes the term "immediate predecessor" is also used. Note that "cover" as used here should not be confused with the "cover" of a set defined in Sec. 2-3.4.

A partial ordering $\leq$ on a set P can be represented by means of a diagram known as a Hasse diagram or a partially ordered set diagram of $\langle P, \leq \rangle$. In such a diagram, each element is represented by a small circle or a dot. The circle for $x \in P$ is drawn below the circle for $y \in P$ if $x < y$, and a line is drawn between x and y if y covers x. If $x < y$ but y does not cover x, then x and y are not connected directly by a single line. However, they are connected through one or more elements of P. It is possible to obtain the set of ordered pairs in $\leq$ from

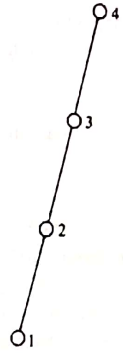


FIGURE 2-3.23 Hasse diagram.

such a diagram. Several examples of partially ordered sets and their Hasse diagrams follow.

For a totally ordered set $\langle P, \leq \rangle$, the Hasse diagram consists of circles, one below the other, as in Fig. 2-3.23. Thus a totally ordered set is called a chain. If we let $P = \{1, 2, 3, 4\}$ and $\leq$ be the relation "less than or equal to," then the Hasse diagram is as shown in Fig. 2-3.23.

Consider the set $P = \{\varnothing, \{a\}, \{a, b\}, \{a, b, c\}\}$ and the relation of inclusion $\subseteq$ on P. The Hasse diagram of $\langle P, \subseteq \rangle$ is similar to that given in Fig. 2-3.23 except that the nodes are relabeled.

The two relations defined above are not equal, but they have the same Hasse diagram. Such situations will be shown to occur frequently, and the reason for these occurrences is explained in Chap. 4 in the discussion of the order isomorphism of two partially ordered sets.

EXAMPLE 1 Let $X = \{2, 3, 6, 12, 24, 36\}$ and the relation $\leq$ be such that $x \leq y$ if x divides y. Draw the Hasse diagram of $\langle X, \leq \rangle$. ////

SOLUTION The Hasse diagram is given in Fig. 2-3.24.

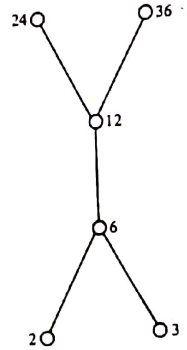


FIGURE 2-3.24 Hasse diagram of divides relation.

Scanned with OKEN Scanner

EXAMPLE 2 Let A be a given finite set and $\rho(A)$ its power set. Let $\subseteq$ be the inclusion relation on the elements of $\rho(A)$. Draw Hasse diagrams of $\langle \rho(A), \subseteq \rangle$ for (a) $A = \{a\}$; (b) $A = \{a, b\}$; (c) $A = \{a, b, c\}$; (d) $A = \{a, b, c, d\}$.

SOLUTION The required Hasse diagrams are given in Fig. 2-3.25a to d. ////

The following points may be noted about Hasse diagrams in general. For a given partially ordered set, a Hasse diagram is not unique, as can be seen from Fig. 2-3.25b. From a Hasse diagram of $\langle P, \leq \rangle$, the Hasse diagram of $\langle P, \geq \rangle$, which is the dual of $\langle P, \leq \rangle$, can be obtained by rotating the diagram through 180° so that the points at the top become the points at the bottom. Some Hasse diagrams have a unique point which is above all the other points, and similarly some Hasse diagrams have a unique point which is below all other points. Such was the case for all the Hasse diagrams given in Example 2, while the Hasse diagram given in Example 1 does not possess this property. The Hasse diagrams become more complicated when the number of elements in the partially ordered set is large.

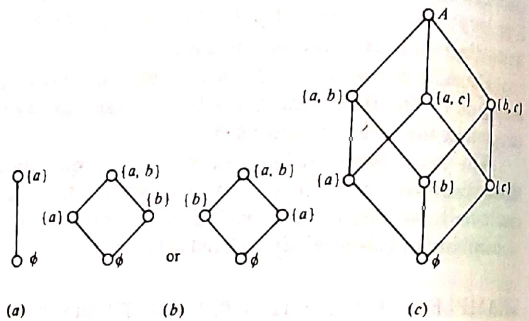


FIGURE 2-3.25 Hasse diagrams of $\langle \rho(A), \subseteq \rangle$.

EXAMPLE 3 Let A be the set of factors of a particular positive integer m and let $\leq$ be the relation divides, i.e.,

$$\leq = \{\langle x, y \rangle \mid x \in A \wedge y \in A \wedge (x \text{ divides } y)\}$$

Draw Hasse diagrams for (a) $m = 2$; (b) $m = 6$; (c) $m = 30$; (d) $m = 210$; (e) $m = 12$; and (f) $m = 45$.

SOLUTION The required Hasse diagrams for (a) to (d) are the same as given in Fig. 2-3.25a to d. Hasse diagrams of (e) and (f) are given in Fig. 2-3.26. ////

In Examples 2 and 3 we saw that the Hasse diagrams (a) to (d) are identical. However, Hasse diagrams (e) and (f) of Example 3 cannot be given by the Hasse diagram of any power set of a set, because a power set has 2^n elements, while in (e) and (f) we only have 6 elements in each of the partially ordered sets. Of course, in all the cases given in Example 3 we again have a single element at the top and a single element at the bottom because if p is any divisor of m, we have $1 \leq p \leq m$.

Hasse diagrams can also be drawn for any relation which is antisymmetric and transitive but not necessarily reflexive. Examples of such relations are proper inclusion and any relation $<$ associated with the partial ordering relation $\leq$. Any family tree or organization chart of the military or of any establishment is a Hasse diagram in this sense. We shall, however, assume that a Hasse diagram represents a partial ordering unless otherwise stated. Some Hasse diagrams are given in Fig. 2-3.27.

We shall now introduce terminology for partially ordered sets which will be found useful in Chap. 4. To this end, let $\langle P, \leq \rangle$ denote a partially ordered set.

If there exists an element $y \in P$ such that $y \leq x$ for all $x \in P$, then y is called the *least member* in P relative to the partial ordering $\leq$. Similarly, if there exists an element $y \in P$ such that $x \leq y$ for all $x \in P$, then y is called the *greatest*

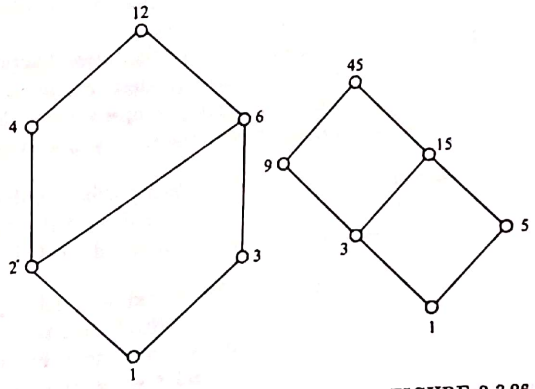


FIGURE 2-3.26

Scanned with OKEN Scanner

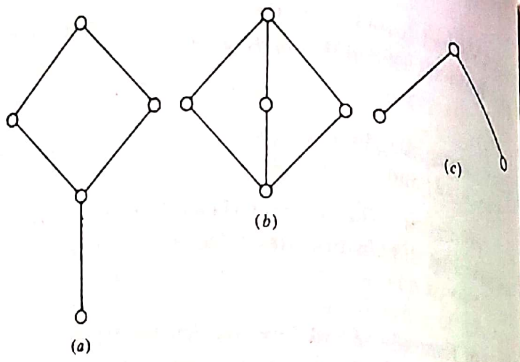


FIGURE 2-3.27

member in P relative to $\leq$. From the definition it is clear that the least member, if it exists, is unique; so also is the greatest member. It may happen that the least or the greatest member does not exist. The least member is usually denoted by 0 and the greatest by 1.

If the Hasse diagram of a partially ordered set is available, then it is easy to see whether the least or the greatest member exists. From Fig. 2-3.23 it is clear that the least member is 1 and the greatest is 4. In Example 1 there is no least or greatest member, while in Example 2 the least member is $\varnothing$ and the greatest member is A in all cases. In every simple ordering or chain, the least and the greatest members always exist. The Hasse diagram of Fig. 2-3.27*c* shows that the greatest member exists but there is no least member.

An element $y \in P$ is called a *minimal member* of P relative to a partial ordering $\leq$ if for no $x \in P$ is $x < y$. A minimal member need not be unique. All those members which appear at the lowest level of a Hasse diagram of a partially ordered set are minimal members. Similarly, an element $y \in P$ is called a *maximal member* of P relative to a partial ordering $\leq$ if for no $x \in P$ is $y < x$. In the Hasse diagram of Fig. 2-3.27*c*, there are two minimal members and one maximal member. Distinct minimal members are incomparable, and distinct maximal members are also incomparable.

It is not always necessary to draw the Hasse diagram of a partially ordered set in order to determine the least, greatest, maximal, and minimal members. However, their determination becomes simple when such a diagram is available.

We now extend these ideas to the subsets of a partially ordered set.

Definition 2-3.18 Let $\langle P, \leq \rangle$ be a partially ordered set and let $A \subseteq P$. Any element $x \in P$ is an *upper bound* for A if for all $a \in A$, $a \leq x$. Similarly, any element $x \in P$ is a *lower bound* for A if for all $a \in A$, $x \leq a$.

Let us consider the partially ordered set $\langle \rho(A), \subseteq \rangle$ in Example 2*c*. We choose a subset B of $\rho(A)$ given by $\{\{b, c\}, \{b\}, \{c\}\}$. Then $\{b, c\}$ and A are upper bounds for B, while $\varnothing$ is its lower bound. For the subset $C = \{\{a, c\}, \{c\}\}$, the upper bounds are $\{a, c\}$ and A while the lower bounds are $\{c\}$ and $\varnothing$. In Example 1, if $A = \{2, 3, 6\}$, then 6, 12, 24, and 36 are upper bounds for A, and there is no lower bound.

Note that upper and lower bounds of a subset are not necessarily unique. We therefore define the following terms.

Definition 2-3.19 Let $\langle P, \leq \rangle$ be a partially ordered set and let $A \subseteq P$. An element $x \in P$ is a *least upper bound*, or *supremum*, for A if x is an upper bound for A and $x \leq y$ where y is any upper bound for A. Similarly, the *greatest lower bound*, or *infimum*, for A is an element $x \in P$ such that x is a lower bound and $y \leq x$ for all lower bounds y.

A least upper bound, if it exists, is unique, and the same is true for a greatest lower bound. The least upper bound is abbreviated as "LUB" or "sup," and the greatest lower bound is abbreviated as "GLB" or "inf."

For a simply ordered set or a chain, every subset has a supremum and an infimum. Similarly, the partially ordered sets given in Examples 2 and 3 are such that every subset has a supremum and an infimum. This, however, is not generally the case, as can be seen from Example 1 in which the set $A = \{2, 3, 6\}$ has the LUB $A = 6$, while the GLB A does not exist. Similarly, for the subset $\{2, 3\}$, the supremum is again 6, but there is no infimum. For the subset $\{12, 6\}$, the supremum is 12 and the infimum is 6. The partially ordered sets which are such that every subset has a supremum and an infimum form an important subclass of partially ordered sets. Such sets are discussed in Chap. 4.

For a partially ordered set $\langle P, \leq \rangle$, we know that its dual $\langle P, \geq \rangle$ is also a partially ordered set. The least member of P relative to the ordering $\leq$ is the greatest member in P relative to the ordering $\geq$, and vice versa. Similarly, the maximal and minimal elements are interchanged. For any subset $A \subseteq P$, the GLB A in $\langle P, \leq \rangle$ is the same as the LUB A in $\langle P, \geq \rangle$.

We shall end this section by defining a property which has important applications in the use of the principle of transfinite induction.

Definition 2-3.20 A partially ordered set is called *well-ordered* if every nonempty subset of it has a least member.

As a consequence of this definition, it follows that every well-ordered set is totally ordered, because for any subset, say $\{x, y\}$, we must have either x or y as its least member. Of course, every totally ordered set need not be well-ordered. A finite totally ordered set is also well-ordered.

A simple example of a well-ordered set is the set $\mathbf{I}_n = \{1, 2, \ldots, n\}$ or the set $\mathbf{I} = \{1, 2, \ldots\}$. Similarly the sets $\mathbf{I}_n \times \mathbf{I}_n$ or $\mathbf{I} \times \mathbf{I}$ are well-ordered under the natural ordering of "less than or equal." It is possible, however, to define a certain partial ordering on $\mathbf{I} \times \mathbf{I}$ such that it is no longer a well-ordered set.

EXERCISES 2-3.9

1 Draw the Hasse diagrams of the following sets under the partial ordering relation "divides," and indicate those which are totally ordered.

$\{2, 6, 24\}$ $\{3, 5, 15\}$ $\{1, 2, 3, 6, 12\}$ $\{2, 4, 8, 16\}$ $\{3, 9, 27, 54\}$

2 If R is a partial ordering relation on a set X and $A \subseteq X$, show that $R \cap (A \times A)$ is a partial ordering relation on A.

Scanned with OKEN Scanner

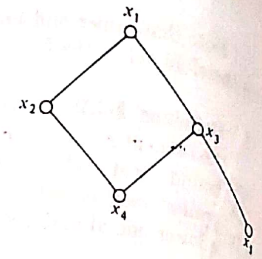


FIGURE 2-3.28

8 Give an example of a set X such that $\langle \rho(X), \subseteq \rangle$ is a totally ordered set.

4 Give a relation which is both a partial ordering relation and an equivalence relation on a set.

5 Let S denote the set of all the partial ordering relations on a set P. Define a partial ordering relation on S and interpret this relation in terms of the elements of P.

6 Figure 2-3.28 gives the Hasse diagram of a partially ordered set $\langle P, R \rangle$, where $P = \{x_1, x_2, \ldots, x_5\}$. Find which of the following are true: $x_1 R x_2$, $x_4 R x_1$, $x_3 R x_5$, $x_2 R x_5$, $x_1 R x_1$, $x_2 R x_3$, and $x_4 R x_5$. Find the least and greatest members in P if they exist. Also find the maximal and minimal elements of P. Find the upper and lower bounds of $\{x_2, x_3, x_4\}$, $\{x_3, x_4, x_5\}$, and $\{x_1, x_2, x_3\}$. Also indicate the LUB and GLB of these subsets if they exist.

7 Show that there are only five distinct Hasse diagrams for partially ordered sets that contain three elements.

2-4 FUNCTIONS

In this section we study a particular class of relations called functions. We are primarily concerned with discrete functions which transform a finite set into another finite set. There are several such transformations involved in the computer implementation of any program. Computer output can be considered as a function of the input. A compiler transforms a program into a set of machine language instructions (the object program). After introducing the concept of function in general, we discuss unary and binary operations which form a class of functions. Such operations have important applications in the study of algebraic structures in Chaps. 3 and 4. Also discussed is a special class of functions known as hashing functions that are used in organizing files on a computer, along with other techniques associated with such organizations. A PL/I program for the construction of a symbol table is also given.

2-4.1 Definition and Introduction

Definition 2-4.1 Let X and Y be any two sets. A relation f from X to Y is called a *function* if for every $x \in X$ there is a unique $y \in Y$ such that $\langle x, y \rangle \in f$.

Note that the definition of function requires that a relation must satisfy two additional conditions in order to qualify as a function. The first condition is that every $x \in X$ must be related to some $y \in Y$, that is, the domain of f must

be X and not merely a subset of X. The second requirement of uniqueness can be expressed as

$$\langle x, y \rangle \in f \wedge \langle x, z \rangle \in f \Rightarrow y = z$$

Terms such as "transformation," "map" (or "mapping"), "correspondence," and "operation" are used as synonyms for "function." The notations $f: X \to Y$ or $X \xrightarrow{f} Y$ are used to express f as a function from X to Y. Pictorially, a function is generally shown as in Fig. 2-4.1.

For a function $f: X \to Y$, if $\langle x, y \rangle \in f$, then x is called an *argument* and the corresponding y is called the *image* of x under f. Instead of writing $\langle x, y \rangle \in f$, it is customary to write $y = f(x)$ and to call y the *value of the function* f at x. Other ways of expressing $y = f(x)$ are $f: x \to y$, $x \xrightarrow{f} y$, and, of course, $\langle x, y \rangle \in f$. As an extension of this notation to the whole set X, we sometimes denote the range of f, viz., R_f, by $f(X)$. The range of f is defined as

$$\{y \mid \exists x \in X \wedge y = f(x)\}$$

It was mentioned that the domain of f is X, that is, $D_f = X$. The range of f is denoted by R_f and $R_f \subseteq Y$. The set Y is called the *codomain* of f. Some authors permit $D_f \subseteq X$, but according to our definition $D_f = X$.

Since a function is a relation, we can use a relation matrix or a graph to represent it in some cases. Note that from the definition of a function it follows that every row of its relation matrix must have only one entry which is 1 while all other entries in this row are 0s. Therefore, one can replace the relation matrix by a single column, i.e., a vector consisting of entries which are images of the arguments. Thus the column consists of entries which show a correspondence between the argument and the image of the function under the argument. In certain other cases, this correspondence can be expressed more easily by a rule. For example, $f(x) = x^2$ for $x \in \mathbf{R}$ represents the function $\{\langle x, x^2 \rangle \mid x \in \mathbf{R}\}$ where $\mathbf{R}$ is the set of real numbers and $f: \mathbf{R} \to \mathbf{R}$. Graphs of some functions are shown in Fig. 2-4.2.

Note that more than one element $x \in X$ may have the same function value; for example, $g(x_1) = g(x_2) = y_3$ for the function g whose graph is given in Fig. 2-4.2.

The following are some illustrations of functions.

1 Let $X = \{1, 5, P, \text{Jack}\}$, $Y = \{2, 5, 7, q, \text{Jill}\}$, and $f = \{\langle 1, 2 \rangle, \langle 5, 7 \rangle, \langle P, q \rangle, \langle \text{Jack}, q \rangle\}$. Obviously $D_f = X$, $R_f = \{2, 7, q\}$, and $f(1) = 2$, $f(5) = 7$, $f(P) = q$, $f(\text{Jack}) = q$.

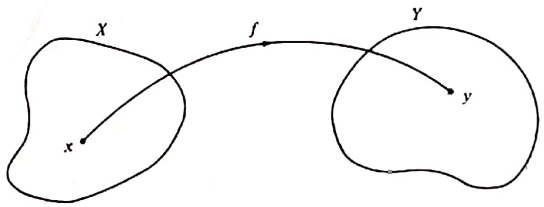


FIGURE 2-4.1 Representation of a function.

Scanned with OKEN Scanner

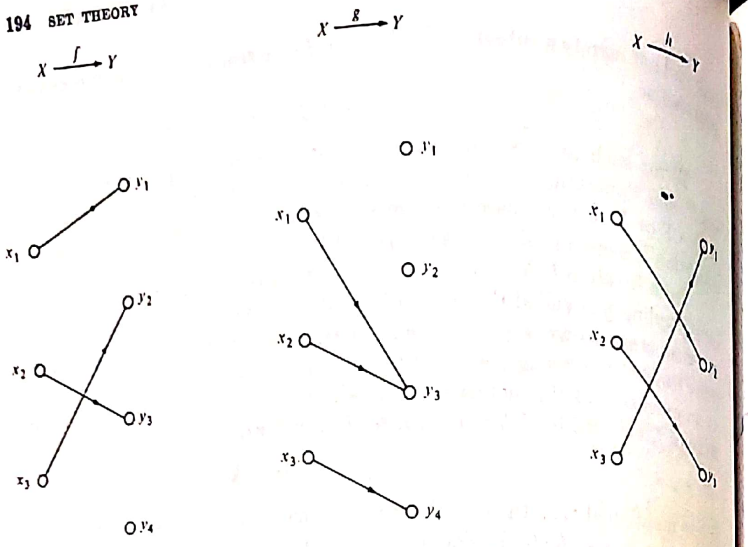


FIGURE 2-4.2 Graphs of functions.

2 Let $X = Y = \mathbf{R}$ and $f(x) = x^2 + 2$. $D_f = \mathbf{R}$ and $R_f \subseteq \mathbf{R}$. The values of f for different values of $x \in \mathbf{R}$ all lie on a parabola, as shown in Fig. 2-4.3.

3 Let $X = Y = \mathbf{R}$ and let

$$f = \{\langle x, x^2\rangle \mid x \in \mathbf{R}\}$$

$$g = \{\langle x^2, x\rangle \mid x \in \mathbf{R}\}$$

Clearly f is a function from X to Y. However, g is not a function because the uniqueness condition is violated, as can be seen by noting that for any real number a, $\langle a^2, a\rangle$ and $\langle a^2, -a\rangle$ are both in g.

4 Let E be the universal set and $\rho(E)$ be its power set. For any two sets $A, B \in \rho(E)$, the operations of union and intersection are mappings from

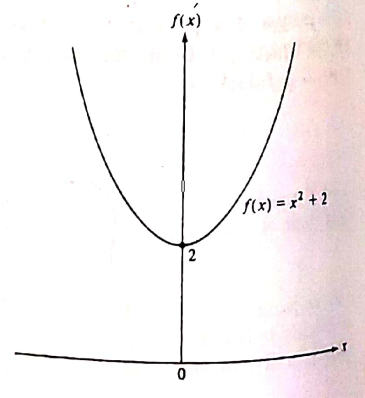


FIGURE 2-4.3

$\rho(E) \times \rho(E)$ to $\rho(E)$. Similarly, complementation is a mapping from $\rho(E)$ to $\rho(E)$.

5 Let P be the set of all positive integers and $\sigma: P \to P$ be such that $\sigma(n) = n + 1$ where $n \in P$. Obviously $\sigma(1) = 2, \sigma(2) = 3, \ldots$. The function σ is called *Peano's successor function* and is used in the description of integers in Sec. 2-5.1.

6 Let X be the set of all statements in logic and let Y denote the set $\{T, F\}$, where T and F denote the truth values. The assignment of truth values to statements can be considered as a mapping from X to Y.

7 Let functions f and g be defined by

$$f = \{\langle x, \lfloor x \rfloor\rangle \mid x \in \mathbf{R} \wedge \lfloor x \rfloor = \text{the greatest integer less than or equal to } x\}$$

$$g = \{\langle x, \lceil x \rceil\rangle \mid x \in \mathbf{R} \wedge \lceil x \rceil = \text{the least integer greater than or equal to } x\}$$

The function $f(x) = \lfloor x \rfloor$ is frequently called the *floor* of x, and similarly, the function $g(x) = \lceil x \rceil$ is called the *ceiling* of x. As examples, study the following:

$$f(3.75) = \lfloor 3.75 \rfloor = 3 \qquad \lfloor 4 \rfloor = 4 \qquad \lfloor -3.75 \rfloor = -4$$

$$g(3.33) = \lceil 3.33 \rceil = 4 \qquad \lceil 4 \rceil = 4 \qquad \lceil -3.33 \rceil = -3$$

8 A program written in a high-level language is transformed (or mapped) into a machine language by a compiler. Similarly, the output from a computer is a function of its input.

Definition 2-4.2 If $f: X \to Y$ and $A \subseteq X$, then $f \cap (A \times Y)$ is a function from $A \to Y$ called the *restriction* of f to A and is sometimes written as f/A. If g is a restriction of f, then f is called the *extension* of g.

Note that $(f/A): A \to Y$ is such that for any $a \in A$, $(f/A)(a) = f(a)$. The domain of f/A is A, while that of f is X. Obviously, if g is a restriction of f, then $D_g \subseteq D_f$ and $g(x) = f(x)$ for $x \in D_g$ and $g \subseteq f$. As illustrations of these concepts, consider the following:

9 Let $f: \mathbf{R} \to \mathbf{R}$ be given by $f(x) = x^2$ as in (3). If $\mathbf{N}$ is the set of natural numbers, $\{0, 1, 2, \ldots\}$, then $\mathbf{N} \subseteq \mathbf{R}$ and

$$f/\mathbf{N} = \{\langle 0, 0\rangle, \langle 1, 1\rangle, \langle 2, 4\rangle, \langle 3, 9\rangle, \ldots\}$$

10 Let $f: \mathbf{R} \to \mathbf{R}$ be given by $f(x) = |x|$, where $|x|$ denotes the absolute value of x. Let $\mathbf{R}_+$ be the set of positive real numbers and $g: \mathbf{R}_+ \to \mathbf{R}$ be given by $g(x) = x$; then g is a restriction of f, that is, $g = f/\mathbf{R}_+$ and f is the extension of g in $\mathbf{R}$.

Equality of functions can be defined in terms of the equality of sets since functions are sets of ordered pairs. This definition also requires that equal functions have the same domain and the same range.

We know that not all possible subsets of $X \times Y$ are functions from X to Y. The collection of all those subsets of $X \times Y$ which define a function is denoted

Scanned with OKEN Scanner

by Y^X. The reason for using this notation will be clear from the following illustration.

11 Let $X = \{a, b, c\}$ and $Y = \{0, 1\}$. Then

$$X \times Y = \{\langle a, 0\rangle, \langle b, 0\rangle, \langle c, 0\rangle, \langle a, 1\rangle, \langle b, 1\rangle, \langle c, 1\rangle\}$$

and there are 2^6 possible subsets of $X \times Y$. Of these, only the following 2^3 subsets define functions from X to Y.

$$f_0 = \{\langle a, 0\rangle, \langle b, 0\rangle, \langle c, 0\rangle\} \qquad f_4 = \{\langle a, 1\rangle, \langle b, 0\rangle, \langle c, 0\rangle\}$$
$$f_1 = \{\langle a, 0\rangle, \langle b, 0\rangle, \langle c, 1\rangle\} \qquad f_5 = \{\langle a, 1\rangle, \langle b, 0\rangle, \langle c, 1\rangle\}$$
$$f_2 = \{\langle a, 0\rangle, \langle b, 1\rangle, \langle c, 0\rangle\} \qquad f_6 = \{\langle a, 1\rangle, \langle b, 1\rangle, \langle c, 0\rangle\}$$
$$f_3 = \{\langle a, 0\rangle, \langle b, 1\rangle, \langle c, 1\rangle\} \qquad f_7 = \{\langle a, 1\rangle, \langle b, 1\rangle, \langle c, 1\rangle\}$$

In order to determine the number of functions from X to Y when both X and Y are finite, let us assume that X and Y have m and n distinct elements respectively. Since the domain of any function from X to Y is X, there are exactly m ordered pairs in each of the functions. Further, any element $x \in X$ can have any one of the n elements of Y as its image; therefore there are n^m possible functions which are distinct. In illustration (11), $m = 3$ and $n = 2$ and there are $2^3 = 8$ functions. The number n^m also explains why the notation Y^X was used to represent the set of all functions from X to Y. The same notation is used even when X or Y are infinite sets. It must be emphasized that the number of functions depends upon the number of elements in the sets X and Y and not on the sets. Therefore, in illustration (11) if Y is any set having 2 elements, the number of functions from X to Y will still be 8.

In the construction of truth tables of a statement function of n variables, we had 2^n rows because each row defined a distinct function from the set of n variables to the set $\{T, F\}$ of truth values.

Definition 2-4.3 A mapping of $f: X \to Y$ is called *onto* (*surjective*, a *surjection*) if the range $R_f = Y$; otherwise it is called *into*.

In Fig. 2-4.2 the function h is an onto mapping, while the others are into mappings. Illustrations (1), (2), (3), and (5) are into mappings while (4) and (6) are onto.

Definition 2-4.4 A mapping $f: X \to Y$ is called *one-to-one* (*injective*, or *1-1*) if distinct elements of X are mapped into distinct elements of Y. In other words, f is one-to-one if

$$x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$$

or equivalently

$$f(x_1) = f(x_2) \Rightarrow x_1 = x_2$$

In Fig. 2-4.2, f and h are both one-to-one. Also the mapping in illustration (5) is one-to-one, and so is $f/\mathbf{N}$ in (9). Although f in (10) is not one-to-one, its restriction g is.

It is easy to see that when X and Y are finite sets, a mapping $f: X \to Y$ can be one-to-one only if the number of elements in X is less than or equal to the number of elements in Y.

Definition 2-4.5 A mapping $f: X \to Y$ is called *one-to-one onto* (*bijective*) if it is both one-to-one and onto. Such a mapping is also called a one-to-one correspondence between X and Y.

For $f: X \to Y$ to be bijective when X and Y are finite requires that the number of elements in X be the same as the number of elements in Y. The function h whose graph is shown in Fig. 2-4.2 is bijective. Similarly, $f(x) = 2x + 1$ and $g(x) = x$ for $x \in \mathbf{R}$ are also bijective mappings from $\mathbf{R}$ to $\mathbf{R}$. We shall be interested in bijective mappings in the next section where it is shown that such functions have inverses.

EXERCISES 2-4-1

1 Let $\mathbf{N}$ be the set of natural numbers including zero. Determine which of the following functions are one-to-one, which are onto, and which are one-to-one onto.

(*a*) $f: \mathbf{N} \to \mathbf{N}$ $\quad f(j) = j^2 + 2$

(*b*) $f: \mathbf{N} \to \mathbf{N}$ $\quad f(j) = j \pmod 3$

(*c*) $f: \mathbf{N} \to \mathbf{N}$ $\quad f(j) = \begin{cases} 1 & j \text{ is odd} \\ 0 & j \text{ is even} \end{cases}$

(*d*) $f: \mathbf{N} \to \{0, 1\}$ $\quad f(j) = \begin{cases} 0 & j \text{ is odd} \\ 1 & j \text{ is even} \end{cases}$

2 Let $\mathbf{I}$ be the set of integers, $\mathbf{I}_+$ the set of positive integers, and $\mathbf{I}_p = \{0, 1, 2, \ldots, p-1\}$. Determine which of the following functions are one-to-one, which are onto, and which are one-to-one onto.

(*a*) $f: \mathbf{I} \to \mathbf{I}$ $\quad f(j) = \begin{cases} j/2 & j \text{ is even} \\ (j-1)/2 & j \text{ is odd} \end{cases}$

(*b*) $f: \mathbf{I}_+ \to \mathbf{I}_+$ $\quad f(x) =$ greatest integer $\leq \sqrt{x}$

(*c*) $f: \mathbf{I}_7 \to \mathbf{I}_7$ $\quad f(x) = 3x \pmod 7$

(*d*) $f: \mathbf{I}_4 \to \mathbf{I}_4$ $\quad f(x) = 3x \pmod 4$

3 If X and Y are finite sets, find a necessary condition for the existence of one-to-one mappings from X to Y.

4 Do the following sets define functions? If so, give their domain and range in each case.

(*a*) $\{\langle 1, \langle 2, 3\rangle\rangle, \langle 2, \langle 3, 4\rangle\rangle, \langle 3, \langle 1, 4\rangle\rangle, \langle 4, \langle 1, 4\rangle\rangle\}$

(*b*) $\{\langle 1, \langle 2, 3\rangle\rangle, \langle 2, \langle 3, 4\rangle\rangle, \langle 3, \langle 3, 2\rangle\rangle\}$

(*c*) $\{\langle 1, \langle 2, 3\rangle\rangle, \langle 2, \langle 3, 4\rangle\rangle, \langle 1, \langle 2, 4\rangle\rangle\}$

(*d*) $\{\langle 1, \langle 2, 3\rangle\rangle, \langle 2, \langle 2, 3\rangle\rangle, \langle 3, \langle 2, 3\rangle\rangle\}$

5 List all possible functions from $X = \{a, b, c\}$ to $Y = \{0, 1\}$ and indicate in each case whether the function is one-to-one, is onto, and is one-to-one onto.

6 If $A = \{1, 2, \ldots, n\}$, show that any function from A to A which is one-to-one must also be onto, and conversely.

Scanned with OKEN Scanner

7 Show that the functions f and g which both are from $\mathbf{N} \times \mathbf{N}$ to $\mathbf{N}$ given by $f\langle x, y\rangle = x + y$ and $g\langle x, y\rangle = xy$ are onto but not one-to-one.

2-4.2 Composition of Functions

The operation of composition of relations can be extended to functions in the following manner.

Definition 2-4.6 Let $f: X \to Y$ and $g: Y \to Z$ be two functions. The *composite relation* $g \circ f$ such that

$$g \circ f = \{\langle x, z\rangle \mid (x \in X) \wedge (z \in Z) \wedge (\exists y)(y \in Y \wedge y = f(x) \wedge z = g(y))\}$$

is called the *composition* of functions or *relative product* of functions f and g. More precisely, $g \circ f$ is called the *left composition* of g with f.

Note that in the above definition it is assumed that the range R_f of f is a subset of the domain of g, which is Y, that is $R_f \subseteq D_g$; otherwise, $g \circ f$ is empty. Assuming that $g \circ f$ is not empty, we now show that $g \circ f$ is a function from X to Z. For this purpose, let us assume that $\langle x, z_1\rangle$ and $\langle x, z_2\rangle$ are both in $g \circ f$. This assumption requires that there is a $y \in Y$ such that $y = f(x)$ and $z_1 = g(y)$; also $z_2 = g(y)$. Since g is a function, we cannot have $z_1 = g(y)$ and $z_2 = g(y)$; hence $g \circ f$ is a function. Any function g for which $g \circ f$ can be formed is said to be *left-composable* with the function f. In such a case, $(g \circ f)(x) = g(f(x))$, where x is in the domain of $g \circ f$. The composition of functions is shown in Fig. 2-4.4.

Given $f: X \to Y$ and $g: Y \to Z$, we have the composite function $g \circ f$. However, the composite function $f \circ g$ may or may not exist. For the existence of $f \circ g$, it is necessary that $R_g \subseteq D_f$. For functions $f: X \to X$ and $g: X \to X$, the composite functions such as $f \circ g$, $g \circ f$, $f \circ f$, $g \circ g$, etc., can be formed. This point will be demonstrated by means of examples.

Consider three functions $f: X \to Y$, $g: Y \to Z$, and $h: Z \to W$. The composite functions $(g \circ f): X \to Z$ and $(h \circ g): Y \to W$ can be formed. Other composite functions such as $h \circ (g \circ f)$ and $(h \circ g) \circ f$ can also be formed. Both of these functions are from X to W. Assuming $y = f(x)$, $z = g(y)$, and $w = h(z)$, we have $\langle x, y\rangle \in f$, $\langle y, z\rangle \in g$, $\langle z, w\rangle \in h$ and $\langle x, z\rangle \in g \circ f$. $\langle y, w\rangle \in h \circ g$. Continuing the same argument, $\langle x, w\rangle \in h \circ (g \circ f)$. Similarly $\langle x, w\rangle \in (h \circ g) \circ f$. This fact being true for any x and corresponding w, we have (see also Fig. 2-4.5)

$$h \circ (g \circ f) = (h \circ g) \circ f \qquad (1)$$

Thus the composition of functions is associative, and we may drop the parentheses in writing the functions in (1), so that $h \circ g \circ f = h \circ (g \circ f) = (h \circ g) \circ f$.

The composition of functions is also shown in Fig. 2-4.6.

EXAMPLE 1 Let $X = \{1, 2, 3\}$, $Y = \{p, q\}$, and $Z = \{a, b\}$. Also let $f: X \to Y$ be $f = \{\langle 1, p\rangle, \langle 2, p\rangle, \langle 3, q\rangle\}$ and $g: Y \to Z$ be given by $g = \{\langle p, b\rangle, \langle q, b\rangle\}$. Find $g \circ f$.

SOLUTION $g \circ f = \{\langle 1, b\rangle, \langle 2, b\rangle, \langle 3, b\rangle\}$. ////

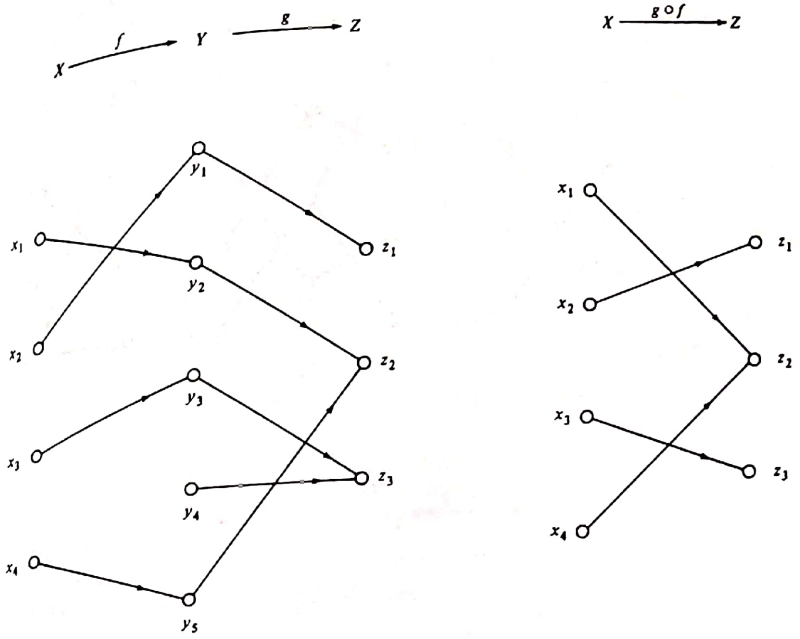


FIGURE 2-4.4 Composition of functions

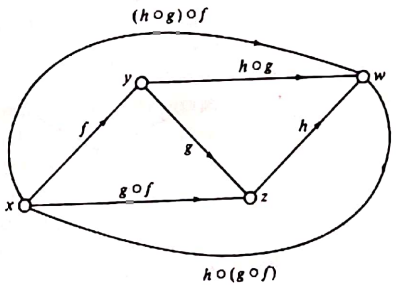


FIGURE 2-4.5

EXAMPLE 2 Let $X = \{1, 2, 3\}$ and f, g, h, and s be functions from X to X given by

$$f = \{\langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3, 1\rangle\} \qquad g = \{\langle 1, 2\rangle, \langle 2, 1\rangle, \langle 3, 3\rangle\}$$
$$h = \{\langle 1, 1\rangle, \langle 2, 2\rangle, \langle 3, 1\rangle\} \qquad s = \{\langle 1, 1\rangle, \langle 2, 2\rangle, \langle 3, 3\rangle\}$$

Find $f \circ g$; $g \circ f$; $f \circ h \circ g$; $s \circ g$; $g \circ s$; $s \circ s$; and $f \circ s$.

Scanned with OKEN Scanner

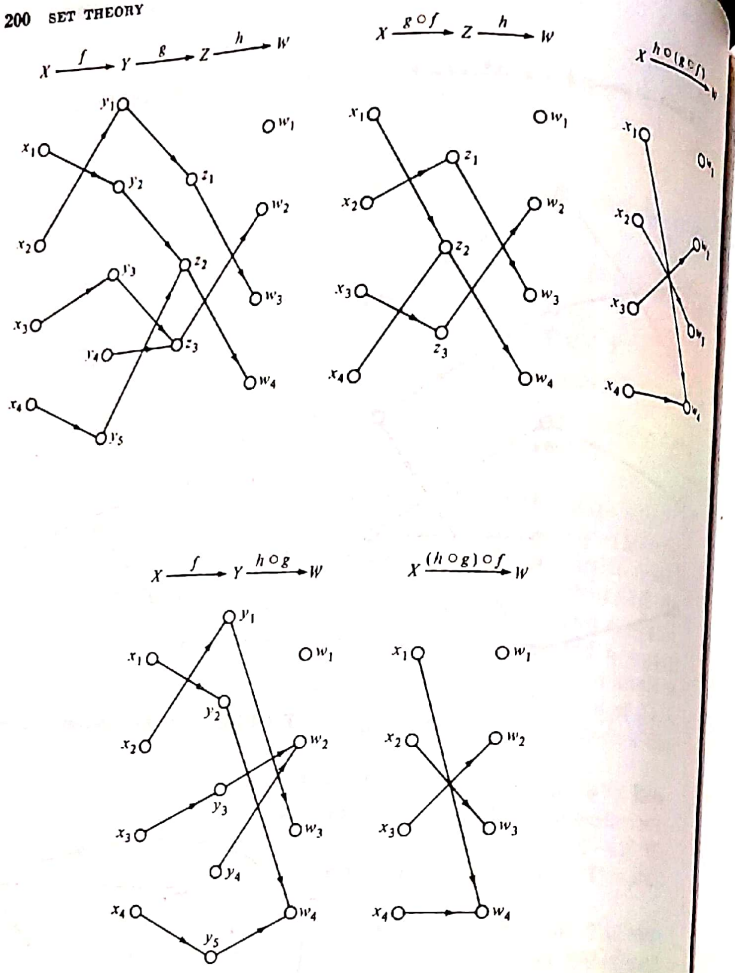


FIGURE 2-4.8

SOLUTION

$$f \circ g = \{\langle 1, 3\rangle, \langle 2, 2\rangle, \langle 3, 1\rangle\}$$
$$g \circ f = \{\langle 1, 1\rangle, \langle 2, 3\rangle, \langle 3, 2\rangle\} \neq f \circ g$$
$$f \circ h \circ g = \{\langle 1, 3\rangle, \langle 2, 2\rangle, \langle 3, 2\rangle\}$$
$$s \circ g = \{\langle 1, 2\rangle, \langle 2, 1\rangle, \langle 3, 3\rangle\} = g = g \circ s$$
$$s \circ s = \{\langle 1, 1\rangle, \langle 2, 2\rangle, \langle 3, 3\rangle\} = s$$
$$f \circ s = \{\langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3, 1\rangle\} = f^{-}$$

Observe that $s \circ s = s$; $f \circ s = s \circ f = f$; $g \circ s = s \circ g = g$; and $h \circ s = s \circ h = h$. ////

EXAMPLE 3 Let $f(x) = x + 2$, $g(x) = x - 2$, and $h(x) = 3x$ for $x \in \mathbf{R}$, where $\mathbf{R}$ is the set of real numbers. Find $g \circ f$; $f \circ g$; $f \circ f$; $g \circ g$; $f \circ h$; $h \circ g$; $h \circ f$; and $f \circ h \circ g$.

SOLUTION

$$g \circ f = \{\langle x, x\rangle \mid x \in \mathbf{R}\}$$
$$f \circ g = \{\langle x, x\rangle \mid x \in \mathbf{R}\} = g \circ f$$
$$f \circ f = \{\langle x, x + 4\rangle \mid x \in \mathbf{R}\}$$
$$g \circ g = \{\langle x, x - 4\rangle \mid x \in \mathbf{R}\}$$
$$f \circ h = \{\langle x, 3x + 2\rangle \mid x \in \mathbf{R}\}$$
$$h \circ g = \{\langle x, 3x - 6\rangle \mid x \in \mathbf{R}\}$$
$$h \circ f = \{\langle x, 3x + 6\rangle \mid x \in \mathbf{R}\}$$
$$(f \circ h) \circ g = \{\langle x, 3x - 4\rangle \mid x \in \mathbf{R}\} = f \circ (h \circ g) = f \circ h \circ g \qquad ////$$

EXAMPLE 4 Let $f: \mathbf{R} \to \mathbf{R}$ be given by $f(x) = -x^2$ and $g: \mathbf{R}_+ \to \mathbf{R}_+$ be given by $g(x) = \sqrt{x}$ where $\mathbf{R}_+$ is the set of nonnegative real numbers and $\mathbf{R}$ is the set of real numbers. Find $f \circ g$. Is $g \circ f$ defined?

SOLUTION $(f \circ g)(x) = -x$ for all $x \in \mathbf{R}_+$. The function $f \circ g$: $\mathbf{R}_+ \to \mathbf{R}$ is defined because the range of g is $\mathbf{R}_+ \subseteq \mathbf{R}$ and $\mathbf{R}$ is the domain of f. On the other hand, the range of f is not included in the domain of g; therefore $g \circ f$ is not defined. The only element common to R_f and D_g is 0. ////

2-4.3 Inverse Functions

The converse of a relation R from X to Y was defined in Sec. 2-3.7 to be a relation $\tilde{R}$ from Y to X such that $\langle y, x\rangle \in \tilde{R} \Leftrightarrow \langle x, y\rangle \in R$; that is, the ordered pairs of $\tilde{R}$ are obtained from those of R by simply interchanging the members. The situation is not quite the same for functions. Let $\tilde{f}$ denote the converse of f, where f is considered as a relation from $X \to Y$. Naturally $\tilde{f}$ may not be a function, first, because the domain of $\tilde{f}$ may not be Y but only a subset of Y, and second, $\tilde{f}$ may not be a function from D_f to X because it may not satisfy the uniqueness condition. For example, $\langle x_1, y\rangle$ and $\langle x_2, y\rangle$ may be in f, so that $\langle y, x_1\rangle$ and $\langle y, x_2\rangle$ will be in $\tilde{f}$. In certain special cases, $\tilde{f}$ may be a function from a subset of Y to X or even from Y to X. The following examples illustrate the situation.

1 Let $X = \{1, 2, 3\}$, $Y = \{p, q, r\}$, and $f: X \to Y$ be given by $f = \{\langle 1, p\rangle, \langle 2, q\rangle, \langle 3, q\rangle\}$. Then $\tilde{f} = \{\langle p, 1\rangle, \langle q, 2\rangle, \langle q, 3\rangle\}$ and $\tilde{f}$ is not a function.

2 Let $\mathbf{R}$ be the set of real numbers and let $f: \mathbf{R} \to \mathbf{R}$ be given by

$$f = \{\langle x, x^2\rangle \mid x \in \mathbf{R}\}$$

Then $\tilde{f} = \{\langle x^2, x\rangle \mid x \in \mathbf{R}\}$ is not a function.

3 Let $\mathbf{R}$ be the set of real numbers and let $f: \mathbf{R} \to \mathbf{R}$ be given by

$$f = \{\langle x, x + 2\rangle \mid x \in \mathbf{R}\}$$

Then $\tilde{f} = \{\langle x + 2, x\rangle \mid x \in \mathbf{R}\}$ is a function from $\mathbf{R}$ to $\mathbf{R}$.

Scanned with OKEN Scanner

4 Let $X = \{0, 1\}$, $Y = \{p, q, r, s\}$, and $f = \{\langle 0, p\rangle, \langle 1, r\rangle\}$. Then $\tilde{f} = \{\langle p, 0\rangle, \langle r, 1\rangle\}$ is a function from a subset of Y to X, that is, $\tilde{f}$: $\{p, r\} \to \{0, 1\}$

It is easy to see that for a given f: $X \to Y$, $\tilde{f}$ is a function only if f is one-to-one. But this condition does not guarantee that $\tilde{f}$ is a function from Y to X. However, if f is one-to-one and onto, i.e., if f is bijective, then $\tilde{f}$ is a function from Y to X. In such cases, $\tilde{f}$ is written as f^{-1} so that f^{-1}: $Y \to X$ and f^{-1} is called the *inverse* of the function f. If f^{-1} exists, then f is called *invertible*. Obviously f^{-1} is also one-to-one and onto.

Definition 2-4.7 A mapping I_x: $X \to X$ is called an *identity map* if $I_x = \{\langle x, x\rangle \mid x \in X\}$.

Observe that for any function g: $X \to X$ the functions $I_x \circ g$ and $g \circ I_x$ are both equal to g. Also for any function f: $X \to Y$, we have $f \circ I_x = f$. These properties of the identity function can be used in stating the following theorems about the inverse of a function.

Theorem 2-4.1 If f: $X \to Y$ is invertible, then

$$f^{-1} \circ f = I_x \quad \text{and} \quad f \circ f^{-1} = I_y \tag{1}$$

Theorem 2-4.2 Let f: $X \to Y$ and g: $Y \to X$. The function g is equal to f^{-1} only if

$$g \circ f = I_x \quad \text{and} \quad f \circ g = I_y \tag{2}$$

Both the conditions given in (2) are necessary, as can be seen from the graphs of f and g shown in Fig. 2-4.7 where $g \circ f = I_x$ but $f \circ g \neq I_y$ and $g \neq f^{-1}$.

PROOF For a proof of Theorem 2-4.2, first we show that if there is any other function h: $Y \to X$ such that

$$h \circ f = I_x \quad \text{and} \quad f \circ h = I_y \tag{3}$$

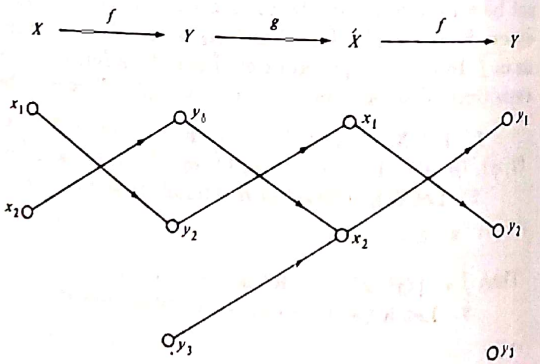


FIGURE 2-4.7

then $h = g$. Once this is proved, then $g = f^{-1}$ follows from Eq. (1). From Eqs. (2) and (3)

$$(h \circ f) \circ g = h \circ (f \circ g) = h \circ I_y = h = I_x \circ g = g$$

Let f: $X \to Y$ and g: $Y \to Z$ be such that $g \circ f$: $X \to Z$ can be constructed. If f and g are both one-to-one and onto, then $g \circ f$ will also be one-to-one and onto, and the inverses f^{-1}, g^{-1}, and $(g \circ f)^{-1}$ exist and are one-to-one and onto. Since f^{-1}: $Y \to X$ and g^{-1}: $Z \to Y$, we can form $f^{-1} \circ g^{-1}$. Both $(g \circ f)^{-1}$ and $f^{-1} \circ g^{-1}$ are functions from Z to X. Consider now any $x \in X$, and let $y = f(x)$ and $z = g(y)$. Thus, $\langle x, z\rangle \in g \circ f$ and $\langle z, x\rangle \in (g \circ f)^{-1}$. On the other hand, $x = f^{-1}(y)$ and $y = g^{-1}(z)$, so that $\langle z, x\rangle \in f^{-1} \circ g^{-1}$. This is true for any x, y, and z which satisfy $y = f(x)$ and $z = g(y)$; hence

$$(g \circ f)^{-1} = f^{-1} \circ g^{-1}$$

i.e., the inverse of a composite function can be expressed in terms of the composition of the inverses in the reverse order. ////

In the remaining part of this section, we consider mappings which are bijective and from a set X onto itself. For this purpose, let F_x denote the collection of all bijective functions from X onto X, so that the elements of F_x are all invertible functions. The following properties hold.

1 For any $f, g \in F_x$, $f \circ g$, and $g \circ f$ are also in F_x. This is called the closure property of the operation of composition, which is discussed in Sec. 2-4.4.

2 For any $f, g, h \in F_x$,

$$(f \circ g) \circ h = f \circ (g \circ h)$$

i.e., composition is associative.

3 There exists a function $I_x \in F_x$ called the *identity map* such that for any $f \in F_x$

$$I_x \circ f = f \circ I_x = f$$

4 For every $f \in F_x$, there exists an inverse function $f^{-1} \in F_x$ such that

$$f \circ f^{-1} = f^{-1} \circ f = I_x$$

In fact, (1) and (2) hold for all the elements of X^X, that is, for all the functions from X to X and not only for the elements of F_x.

EXAMPLE 1 Show that the functions $f(x) = x^3$ and $g(x) = x^{1/3}$ for $x \in \mathbf{R}$ are inverses of one another.

SOLUTION Since $(f \circ g)(x) = f(x^{1/3}) = x = I_x$ and $(g \circ f)(x) = g(x^3) = x = I_x$, then $f = g^{-1}$ or $g = f^{-1}$. ////

EXAMPLE 2 Let F_x be the set of all one-to-one onto mappings from X onto X, where $X = \{1, 2, 3\}$. Find all the elements of F_x and find the inverse of each element.

SOLUTION The graphs of functions shown in Fig. 2-4.8 represent the elements of $F_x = \{f_1, f_2, f_3, f_4, f_5, f_6\}$, where $f_1^{-1} = f_1$, $f_2^{-1} = f_2$, $f_3^{-1} = f_3$, $f_4^{-1} = f_4$,

Scanned with OKEN Scanner

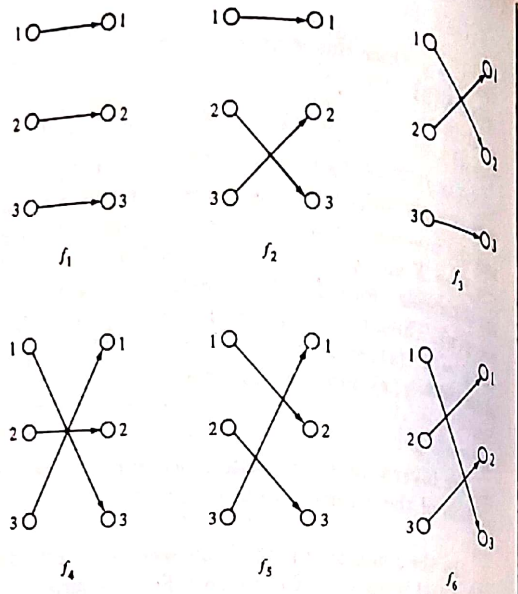


FIGURE 2-4.8

Table 2-4.1

$\circ$	f_1	f_2	f_3	f_4	f_5	f_6
f_1	f_1	f_2	f_3	f_4	f_5	f_6
f_2	f_2	f_1	f_6	f_5	f_4	f_3
f_3	f_3	f_5	f_1	f_6	f_2	f_4
f_4	f_4	f_6	f_5	f_1	f_3	f_2
f_5	f_5	f_3	f_4	f_2	f_6	f_1
f_6	f_6	f_4	f_2	f_3	f_1	f_5

and $f_5^{-1} = f_6$. Other compositions of the elements of F_x are given in Table 2-4.1, in which $f_i \circ f_j$ is entered at the intersection of the ith row and jth column. ////

The functions defined in Example 2 show the permutations of the elements of the set X. There are $3! = 6$ such permutations of 3 elements, and hence there are 6 functions from X to X which are bijective. If a set X has n elements, then there are $n!$ functions from X to X which are bijective.

EXERCISES 2-4.3

1 Let $f: \mathbf{R} \to \mathbf{R}$ and $g: \mathbf{R} \to \mathbf{R}$, where $\mathbf{R}$ is the set of real numbers. Find $f \circ g$ and $g \circ f$, where $f(x) = x^2 - 2$ and $g(x) = x + 4$. State whether these functions are injective, surjective, and bijective.

2 If $f: X \to Y$ and $g: Y \to Z$ and both f and g are onto, show that $g \circ f$ is also onto. Is $g \circ f$ one-to-one if both g and f are one-to-one?

3 Let $f: \mathbf{R} \to \mathbf{R}$ be given by $f(x) = x^3 - 2$. Find f^{-1}.

4 How many functions are there from X to Y for the sets given below? Find also the number of functions which are one-to-one, onto, and one-to-one onto.

(a) $X = \{1, 2, 3\}$ $Y = \{1, 2, 3\}$
(b) $X = \{1, 2, 3, 4\}$ $Y = \{1, 2, 3\}$
(c) $X = \{1, 2, 3\}$ $Y = \{1, 2, 3, 4\}$

5 Show that there exists a one-to-one mapping from $A \times B$ to $B \times A$. Is it also onto?

6 Let $X = \{1, 2, 3, 4\}$. Define a function $f: X \to X$ such that $f \neq I_x$ and is one-to-one. Find $f \circ f = f^2$, $f^3 = f \circ f^2$, f^{-1}, and $f \circ f^{-1}$. Can you find another one-to-one function $g: X \to X$ such that $g \neq I_x$ but $g \circ g = I_x$?

2-4.4 Binary and n-ary Operations

In Sec. 2-4.1 we discussed functions from a set X to a set Y. Now we restrict our discussion to functions from a set $X \times X$ to X, or more generally to functions from X^n to X, where $n = 1, 2, \ldots$. Such a mapping prescribes a unique value in X to every ordered pair or n-tuple whose members are also in X.

Definition 2-4.8 Let X be a set and f be a mapping $f: X \times X \to X$. Then f is called a *binary operation* on X. In general, a mapping $f: X^n \to X$ is called an *n-ary operation* and n is called the *order* of the operation. For $n = 1$, $f: X \to X$ is called a *unary* operation.

If an operation (or a mapping) on the members of a set produces images which are also members of the same set, then the set is said to be *closed* under that operation, and this property is called the *closure* property. The definition of binary or n-ary operations implies that the sets on which such operations are defined are closed under these operations. This property distinguishes the binary or n-ary operations from other functions.

The operations of addition, multiplication, and subtraction are binary operations on the set of integers and also on the set of real numbers. The operation of division is not a binary operation on these sets. Operations of set union and intersection are binary operations on the set of subsets of a universal set. They are also binary operations on the power set of any set. The operation of complementation (absolute or relative) is a unary operation on such sets. The operations of conjunction and disjunction are binary operations on the set of statements as well as on the set of statement formulas in statement logic. The operation of negation is a unary operation on such sets. Another example of a binary operation is the composition of bijective functions from a set X to X.

Sometimes a binary operation can be conveniently specified by a table called the composition table. Such a table was given in Example 2 of the previous subsection. The composition tables for the binary operations of union and intersection over the power set $\rho(A) = \{B_0, B_1, B_2, B_3\}$, where $A = \{a, b\}$, are given in Table 2-4.2 where $B_0 = \varnothing$, $B_1 = \{b\}$, $B_2 = \{a\}$, and $B_3 = \{a, b\} = A$.

It is customary to denote a binary operation by a symbol such as $+$, $-$, $\cdot$, $*$, Δ, $\cup$, $\cap$, $\vee$, $\wedge$, etc., and the value of the operation (or function) by placing the operator between the two operands. For example, $f(x, y)$ may be written as $x f y$ or $x * y$. A similar notation is used in arithmetic where we write the

Table 2-4.2

$\cup$	B_0	B_1	B_2	B_3
B_0	B_0	B_1	B_2	B_3
B_1	B_1	B_1	B_3	B_3
B_2	B_2	B_3	B_2	B_3
B_3	B_3	B_3	B_3	B_3

$\cap$	B_0	B_1	B_2	B_3
B_0	B_0	B_0	B_0	B_0
B_1	B_0	B_1	B_0	B_1
B_2	B_0	B_0	B_2	B_2
B_3	B_0	B_1	B_2	B_3

sum of two real numbers x and y as $x + y$. Also in set theory the union of two sets A and B is written as $A \cup B$.

Now we discuss some general properties of binary operations. For this purpose we shall consider X to be any set.

Definition 2-4.9 A binary operation $f: X \times X \to X$ is said to be *commutative* if for every $x, y \in X$,

$$f\langle x, y\rangle = f\langle y, x\rangle$$

Definition 2-4.10 A binary operation $f: X \times X \to X$ is said to be *associative* if for every $x, y, z \in X$,

$$f\langle f\langle x, y\rangle, z\rangle = f\langle x, f\langle y, z\rangle\rangle$$

Definitions 2-4.9 and 2-4.10 can be rewritten using $*$ to denote the binary relation on X. That is, $*$ is commutative if for any $x, y \in X$, $x * y = y * x$. Similarly $*$ is associative on X if for any $x, y, z \in X$,

$$(x * y) * z = x * (y * z)$$

Definition 2-4.11 A binary operation $f: X \times X \to X$ is said to be *distributive* over the operation $g: X \times X \to X$ if for every $x, y, z \in X$

$$f\langle x, g\langle y, z\rangle\rangle = g\langle f\langle x, y\rangle, f\langle x, z\rangle\rangle$$

If we denote f by $*$ and g by $\circ$ we say $*$ is distributive over $\circ$ if for any $x, y, z \in X$

$$x * (y \circ z) = (x * y) \circ (x * z)$$

The operations of addition and multiplication over the set of real numbers are commutative and associative. Union and intersection over the power set of any set are other examples of commutative and associative operations. The operation of subtraction over the set of real numbers is not commutative. It was also shown that the composition of bijective functions from a set X to X is not commutative. The operation of multiplication is distributive over that of addition. Both union and intersection of sets distribute over each other.

Given a binary operation $*$ on a set X, we now define certain distinguished elements of X which are associated with the operation. Such elements may or may not exist.

Definition 2-4.12 Let $*$ be a binary operation on X. If there exists an element $e_l \in X$ such that $e_l * x = x$ for every $x \in X$, then e_l is called a *left identity* with respect to $*$. Similarly, if there exists an element $e_r \in X$ such that $x * e_r = x$ for every $x \in X$, then e_r is called a *right identity* with respect to $*$.

We now give a theorem which relates left and right identities if both of them exist.

Theorem 2-4.3 Let $*$ be a binary operation, and let e_l and e_r be left and right identities with respect to $*$. Then $e_l = e_r = e$ (say), such that $e * x = x * e = x$ for every $x \in X$, and in such a case $e \in X$ is unique and is called the *identity* with respect to $*$.

PROOF Since e_l and e_r are left and right identities,

$$e_l * e_r = e_l = e_r$$

Next, let us assume e_1 and e_2 are two distinct identities. Then

$$e_1 * e_2 = e_1 = e_2$$

which is a contradiction; hence an identity, if it exists, is unique. ////

For a commutative binary operation, a left identity is also a right identity, and hence any left or right identity is the identity.

The element 0 is the identity for addition, and 1 is the identity for multiplication over a set of real numbers. Similarly the empty set $\varnothing$ is the identity for the operation of union, and the universal set E is the identity for the operation of intersection over the subsets of a universal set. The identity mapping defined in Sec. 2-4.3 is the identity with respect to composition of bijective functions from a set X to X. A contradiction, i.e., an identically false statement, is an identity for disjunction, while a tautology is an identity for conjunction of statements.

Definition 2-4.13 Let $*$ be a binary operation on X. If there exists an element $0_l \in X$ such that $0_l * x = 0_l$ for every $x \in X$, then 0_l is called a *left zero* with respect to $*$. Similarly, if there exists an element $0_r \in X$ such that $x * 0_r = 0_r$ for every $x \in X$, then 0_r is called a *right zero* with respect to $*$.

A theorem similar to Theorem 2-4.3 can now be given.

Theorem 2-4.4 Let $*$ be a binary operation, and 0_l and 0_r be left and right zeros with respect to $*$. Then $0_l = 0_r = 0$ such that

$$0 * x = x * 0 = 0 \qquad \text{for all } x \in X$$

$0 \in X$ is unique and is called the *zero* with respect to $*$.

The element 0 is the zero for multiplication on a set of real numbers. The

Scanned with OKEN Scanner

empty set $\varnothing$ is the zero for intersection, and the universal set E is the zero for the union of subsets of a universal set.

Definition 2-4.14 Let $*$ be a binary operation on X. An element $a \in X$ is called *idempotent* with respect to $*$ if $a * a = a$.

The identity and zero elements with respect to a binary operation are idempotent. There may be other idempotent elements besides the identity and zero elements. For example, every set is idempotent with respect to the operations of union and intersection.

Definition 2-4.15 Let $*$ be a binary operation on X with the identity e. An element $a \in X$ is said to be *left-invertible* if there exists an element $x_l \in X$ such that $x_l * a = e$. x_l is called a *left inverse* of a. Similarly, $a \in X$ is said to be *right-invertible* if there exists an element $x_r \in X$ such that $a * x_r = e$. x_r is called a *right inverse* of a. If an element $a \in X$ is both left-invertible and right-invertible, then a is called *invertible*.

Obviously, if a binary operation $*$ on X with the identity e is commutative, then any element that is left- or right-invertible is invertible. For operations which are associative, we can prove the following theorem.

Theorem 2-4.5 Let $*$ be a binary operation on X which is associative and which has the identity $e \in X$. If an element $a \in X$ is invertible, then both its left and right inverses are equal. Such an element is called the inverse of a because it is unique.

PROOF Let x_l and x_r be any left and right inverses of a respectively. We show that $x_l = x_r$ as follows.

$$x_l * a = a * x_r = e$$

hence

$$x_l * a * x_r = (x_l * a) * x_r = x_l * (a * x_r) = e * x_r = x_r = x_l * e = x_l$$

Here we have used the associativity of the operation $*$.

To show uniqueness, let us assume that x and y are two distinct inverses of a. Thus

$$y = y * e = y * (a * x) = (y * a) * x = e * x = x$$

which is a contradiction. ////

The unique inverse of an element $a \in X$, if it exists, is denoted by a^{-1}, so that

$$a^{-1} * a = a * a^{-1} = e$$

From symmetry it follows that $(a^{-1})^{-1} = a$.

In any binary operation the identity element, if it exists, is invertible. Since it is also idempotent, the identity is its own inverse. Other invertible elements may or may not exist. For example, every real number $a \in \mathbf{R}$ has an inverse $-a \in \mathbf{R}$ for the operation of addition. Similarly, for the operation of multiplication, the inverse of every nonzero real number $a \in \mathbf{R}$ is $1/a \in \mathbf{R}$. For a set A which is a subset of a universal set, the set A is idempotent for the operations of union as well as intersection. In Example 2 of Sec. 2-4.3, all the functions are invertible for the operation of composition. It may be noted that a zero element with respect to an operation cannot be invertible.

Definition 2-4.16 An element $a \in X$ is called *cancellable* with respect to a binary operation $*$ on X, if for every $x, y \in X$,

$$(a * x = a * y) \vee (x * a = y * a) \Rightarrow (x = y)$$

If the operation $*$ is associative and the element $a \in X$ is invertible, then a is cancellable. However, there are cases where an element is cancellable but not necessarily invertible. For example, in the set of integers any nonzero integer is cancellable with respect to the operation of multiplication, although the only integer which is invertible is the identity, that is, 1.

The properties of binary operations given here are used in Chaps. 3 and 4. We have not discussed n-ary operations in general because we will be most concerned with unary and binary operations.

Given a set X and a binary operation $*$ on X, we have represented the value of the binary operation $*$ on any two elements $x, y \in X$ by writing $x * y$. Since $x * y \in X$, we may again apply the same or any other binary operation, say $+$, on $x * y$ and an element of X, say z. Obviously the following possibilities exist:

$$(x * y) * z \qquad z * (x * y) \qquad (x * y) + z \qquad z + (x * y)$$

The parentheses have been used in the usual sense to indicate that in all these cases $x * y$ is obtained first. If the operation $*$ is associative, then $(x * y) * z = x * (y * z)$, and we may drop the parentheses. In other words, the order in which the two binary operations are carried out is not important. Sometimes the parentheses are dropped even though the operation is not associative. In such cases, a convention is adopted regarding the order in which the operations are performed. For example, in a computer program using FORTRAN, A + B + C is understood to be (A + B) + C. Note that (A + B) + C may not be equal to A + (B + C) due to rounding-off operations on a computer. In any case, we say that the operation $+$ is assumed to be left-associative. In the case of the assignment A ← B, we understand A ← (B ← C), that is, the operation is right-associative. Similarly for a logical expression $\neg\neg P$, it is assumed that we have $\neg(\neg P)$, where $\neg P$ is formed first. Since subtraction is not associative, we write (A − B) − C as distinct from A − (B − C). However, in FORTRAN A − B − C is understood to mean (A − B) − C, and so subtraction is left-associative. It is very important to know whether an operation is understood to be left- or right-associative. This need to know is greater in the case of programming languages than in mathematics where parentheses are always used except when an operation is associative.

Scanned with OKEN Scanner

EXERCISES 2-4.4

1 Let $g: I \times I \to I$ where I is the set of integers and

$$g\langle x, y\rangle = x * y = x + y - xy$$

Show that the binary operation $*$ is commutative and associative. Find the identity element and indicate the inverse of each element.

2 Let $*$ denote a binary operation on the set of natural numbers given by $x * y = x$. Show that $*$ is not commutative, but is associative. Which elements are idempotent? Are there any left or right identities?

3 Let $x * y =$ lowest common multiple of x and y, where $*$ is a binary operation on the set of positive integers. Show that $*$ is commutative and associative. Find the identity element and also state which elements are idempotent.

4 Let $I_p = \{0, 1, 2, \ldots, p-1\}$. The operations $+_p$ and $*_p$ are given by $x +_p y = (x + y) \pmod p$ and $x *_p y = xy \pmod p$. Give composition tables for these operations for $p = 3$ and 4. Indicate the identity and zero elements. Does the operation $*_p$ distribute over $+_p$?

5 Show that $x * y = x - y$ is not a binary operation over the set of natural numbers, but that it is a binary operation on the set of integers. Is it commutative or associative?

6 Show that $x * y = x^y$ is a binary operation on the set of positive integers. Determine whether $*$ is commutative or associative.

7 How many distinct binary operations are there on the set $\{0, 1\}$? Give their composition tables and indicate which ones are commutative or associative. Can you determine the number of distinct binary operations on any finite set?

2-4.5 Characteristic Function of a Set

In this section we shall discuss functions from the universal set E to the set $\{0, 1\}$. These functions are associated with sets in the same way as the principle of specification given in Sec. 2-1.7. A one-to-one correspondence is established between these functions and the sets. With the use of these functions, statements about sets and their operations can be represented on a computer in terms of binary numbers; hence their manipulation becomes easier.

Definition 2-4.17 Let E be a universal set and A be a subset of E. The function $\psi_A: E \to \{0, 1\}$ defined by

$$\psi_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

is called the *characteristic function* of the set A.

As an example, let E be the set of all persons living in Toronto and let F be the set of all females in Toronto. Then ψ_F associates the number 1 with each female and 0 with each male in Toronto.

The following properties suggest how one can use the characteristic functions of sets to determine set relations.

Let A and B be any two subsets of a universal set E. Then the following hold for all $x \in E$.

$$\psi_A(x) = 0 \Leftrightarrow A = \varnothing \tag{1}$$
$$\psi_A(x) = 1 \Leftrightarrow A = E \tag{2}$$
$$\psi_A(x) \leq \psi_B(x) \Leftrightarrow A \subseteq B \tag{3}$$
$$\psi_A(x) = \psi_B(x) \Leftrightarrow A = B \tag{4}$$
$$\psi_{A \cap B}(x) = \psi_A(x) * \psi_B(x) \tag{5}$$
$$\psi_{A \cup B}(x) = \psi_A(x) + \psi_B(x) - \psi_{A \cap B}(x) \tag{6}$$
$$\psi_{\sim A}(x) = 1 - \psi_A(x) \tag{7}$$
$$\psi_{A-B}(x) = \psi_{A \cap \sim B}(x) = \psi_A(x) - \psi_{A \cap B}(x) \tag{8}$$

Note that the operations $\leq$, $=$, $+$, $*$, and $-$ used with the characteristic functions are the usual arithmetic operations because the values of the characteristic functions are always either 1 or 0. On the other hand, the equality used for sets is the usual set equality. Other set operations used above are $\cup$, $\cap$, $\sim$, and $-$. The above properties can easily be proved using the definition of characteristic functions. For example, (5) can be proved as follows:

$x \in A \cap B \Leftrightarrow x \in A \wedge x \in B$, so that $\psi_A(x) = 1$ and $\psi_B(x) = 1$ and $\psi_{A \cap B}(x) = 1 * 1 = 1$. If $x \notin A \cap B$, then $\psi_{A \cap B}(x) = 0$ and $\psi_A(x) = 0$ or $\psi_B(x) = 0$. Consequently $\psi_A(x) * \psi_B(x) = 0$.

Many set identities and other relations can be proved by using characteristic functions and the usual arithmetic operations and relations.

EXAMPLE 1 Show that $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.

SOLUTION

$$\begin{aligned}
\psi_{A \cap (B \cup C)}(x) &= \psi_A(x) * \psi_{B \cup C}(x) && \text{(using 5)} \\
&= \psi_A(x)(\psi_B(x) + \psi_C(x) - \psi_{B \cap C}(x)) && \text{(using 6)} \\
&= \psi_A(x) * \psi_B(x) + \psi_A(x) * \psi_C(x) \\
&\quad - \psi_A(x) * \psi_{B \cap C}(x) \\
&= \psi_{A \cap B}(x) + \psi_{A \cap C}(x) - \psi_{A \cap (B \cap C)}(x) && \text{(using 5)} \\
&= \psi_{A \cap B}(x) + \psi_{A \cap C}(x) - \psi_{A \cap B \cap C}(x) \\
&= \psi_{A \cap B}(x) + \psi_{A \cap C}(x) - \psi_{(A \cap B) \cap (A \cap C)}(x) \\
&= \psi_{(A \cap B) \cup (A \cap C)}(x) && \text{(using 6)} \qquad ////
\end{aligned}$$

EXAMPLE 2 Show that $\sim\sim A = A$.

SOLUTION

$$\begin{aligned}
\psi_{\sim\sim A}(x) &= 1 - \psi_{\sim A}(x) \\
&= 1 - (1 - \psi_A(x)) && \text{(using 7)} \\
&= \psi_A(x) && ////
\end{aligned}$$

Scanned with OKEN Scanner

The notation used for naming the subsets of a finite set introduced earlier in Sec. 2-1.3 can now be explained by using the characteristic function. Consider $E = \{a, b, c\}$. The subsets of E are $\varnothing$, $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{a, c\}$, $\{b, c\}$, and $\{a, b, c\}$. The values of the characteristic functions of these subsets are given in Table 2-4.3. The values of the characteristic function of any of the subsets of E consist of three binary digits or binary triples. If we let

$$B = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

then Table 2-4.3 can be considered as a mapping from the power set of E to B. This mapping is one-to-one and onto and hence describes a one-to-one correspondence between the sets $\rho(E)$ and B. The elements of B were used to denote the corresponding subsets.

Consider a universal set E and a mapping f from the set E to a finite set $\{a_1, a_2, \ldots, a_n\}$ where $a_1, a_2, \ldots, a_n$ are all distinct. Let A_1 be the set of elements of E such that $f(x) = a_1$ for $x \in A_1$. Similarly define the subsets $A_2, A_3, \ldots, A_n$ of E. Obviously, $A_1, A_2, \ldots, A_n$ are all disjoint; in addition, $A_1 \cup A_2 \cup \cdots \cup A_n = E$, that is, $A_1, A_2, \ldots, A_n$ are the blocks of a partition of E. It is possible to write

$$f(x) = \sum_{i=1}^{n} a_i * \psi_{A_i}(x)$$

A function $f(x)$ which has a finite set of possible values is called a *simple function*. Obviously the range of a simple function is a finite set. It is possible to extend the above description to functions which have countably infinite distinct values.

Finally, we consider the characteristic functions of certain sets which were called minterms, or complete intersections in Sec. 2-3.4. It was noted that the minterms generated by a finite number of sets define a partition of the universal set. Let $X_1, X_2, \ldots, X_n$ be any n subsets of a universal set E and let $I_0, I_1, \ldots, I_{2^n-1}$ be the minterms or complete intersections of $X_1, X_2, \ldots, X_n$. Any element $x \in E$ is a member of only one of the minterms. If $x \in I_j$, then $\psi_{I_j}(x) = 1$ and $\psi_{I_m}(x) = 0$ for $m \neq j$. This statement holds for any $x \in I_j$. Now let us form a new set, say F, from the sets $X_1, \ldots, X_n$ by using the operations of union, intersection, and complementation. Then the characteristic function of F will remain constant over the sets of minterms.

2-4.6 Hashing Functions

In Sec. 2-2.5 we introduced terms such as "file," "record," "field," etc., which are used frequently in connection with the storage of information on a computer. An example of a file is the symbol table of a compiler or an assembler which contains variable names, labels, and literals, together with any associated values (addresses and values of variables or literals). Each entry in a symbol table can be considered as a record. In this section we discuss some aspects of file organization that require the insertion, deletion, and searching of records in a file. Hashing functions are introduced during the course of this discussion. A familiarity with the characteristics of magnetic tape and also with direct-access storage devices such as magnetic drum, disk, etc., is assumed.

Table 2-4.3

x	$\varnothing$	$\{a\}$	$\{b\}$	$\{c\}$	$\{a, b\}$	$\{a, c\}$	$\{b, c\}$	$\{a, b, c\}$
a	0	1	0	0	1	1	0	1
b	0	0	1	0	1	0	1	1
c	0	0	0	1	0	1	1	1

In the case of a magnetic tape, we know that binary-encoded information can be read from or written on a tape as the tape moves past a stationary read/write head. Records are stored consecutively on a tape and are accessed by skip-forward, skip-backward, or rewind operations. Although computer tape units read and write at high speeds, there are certain disadvantages associated with their use. For example, the process of locating a particular record is time-consuming. On the other hand, for direct- (or random-) access storage devices, a particular location where data are stored has a unique address and hence can be accessed rapidly.

In the files that we consider here and more generally, each record contains a field which is designated as a *key* to that record. The key has a value that identifies a record and thus is an indication of where a record is stored or found in computer storage. For example, in a student history file, a field which represents a student number in each student record could be treated as a key. Such a key could also be used to specify an ordering on the student records.

Let us first consider sequential files in which the records are stored in successive physical locations of memory or a storage device. In fact, this restriction is necessary for files stored on magnetic tape or punched cards. Such a file may be unsorted or sorted. Here "sorting" means ordering the records in some way by using the key values. If a file is unsorted, then a record can be inserted very easily at one end, but deleting a record is not as simple. If it is desired to access a particular record according to its key, then the file must be searched sequentially from its beginning until that record is found. The average time to complete such a search is proportional to $n/2$ where n is the number of records in the file. This process is very time-consuming, and an alternative procedure known as *binary search* could be used to reduce the search time.

In order to use the binary search procedure, it is necessary that the records in a sequential file be stored according to an alphabetically or numerically increasing order of the keys. We shall assume that such is the case. Now, given a key value of a particular record to be searched, first the middle record of the file is located, and then the value of its key is compared with that of the record under consideration. If the value of the key is the same as that of the middle record, then the search is complete. If the value of the key is less than that of the middle record, then the first half of the file is examined by repeatedly using the same procedure until the record is located. On the other hand, if the value of the key is greater than that of the middle record, the second half of the file is searched in the same way. The average time for a binary search is proportional to $\log_2 n$, which is a considerable improvement over the time required for a linear search on an unordered file. This improvement is significant when n is large. For small n, the work involved in ordering and the complexity of programming may not justify a binary search. Further, a binary search is not suitable when a

Scanned with OKEN Scanner

file is stored on magnetic tape, because the time taken to skip forward and backward to the midpoint several times during the search would be considerable.

Although a record insertion in an unordered file is an easy task, it becomes complicated in the case of an ordered sequential file. Deletions from a sequential file are time-consuming both in an ordered and in an unordered file.

From this discussion it appears that sequential files are not suitable from the point of view of organization because, of three operations—deletion, insertion, and search—only one can be performed efficiently on either an ordered or an unordered file. However, there are many data processing applications which require sequential processing of records in a file. As an example, consider the case in which a master file of records is updated from a transaction file, which has been ordered in the same way as the master file. In such cases, magnetic tape and punched card storage are quite efficient.

An alternative to sequential processing is random processing, in which the records in a file are processed in any order. Direct organization of files can be used in this type of processing. In direct organization, the address of a record is obtained by performing some reproducible arithmetic or logical operation on the internal bit representation of its key. Any transformation which maps the internal bit representation of the set of keys to a set of addresses is called a *hashing function*. Various hashing functions are available. Two commonly used hashing functions are obtained by what are known as the division method and the mid-square method.

Before we describe the hashing function obtained by the division method, note that every key has a binary representation, which may be treated as a binary number. Let this numerical value of a key be denoted by k, and let n be a fixed integer (preferably prime) which is suitably chosen. Then the hashing function h defined by the division method is

$$h(k) = k \pmod{n}$$

that is, $h(k)$ is the remainder of dividing k by n and is therefore an element of $\{0, 1, \ldots, n-1\}$. Thus the hashing function maps the set of keys to the set of n addresses, viz., the set $\{0, 1, \ldots, n-1\}$, which may be called the address set. The choice of n depends upon the fact that a good hashing function should uniformly distribute the records over the elements of the address set.

A hashing function quite often maps different keys to the same element of the address set. Thus the set of records is partitioned into n equivalence classes. Those records which are mapped to the same address are in the same equivalence class. It is therefore necessary to provide storage space for and also a method of finding the "colliding" or "overflow" records when more than one record has the same address. There are many techniques, called *collision resolution techniques*, for this purpose. One method, called *open addressing*, probes possible addresses in some reproducible order and inserts the colliding record at the first empty location found. The data structure used in this case would be a vector, each element of which is capable of holding a record. There are many variations of this basic concept. Another method known as *chaining* uses a linear linked list to represent each equivalence class in memory. In other words, a one-to-one correspondence is established between the equivalence classes and linked lists.

An algorithm for inserting a record in a directly organized file which uses chaining would proceed as follows. First the key of the record is mapped into an address using a hashing function. In this sense, the record belongs to an equivalence class. Corresponding to the equivalence class there is a linked list of that class. If the record is not already in the linked list of that class, then it is added to the end of the list. The search for or the deletion of a record can be performed in a similar manner.

We can achieve greater efficiency (in terms of speed, not required storage) with direct organization of files than with sequentially organized files if we make proper choices of hashing functions and collision resolution techniques. The importance of a suitable hashing function cannot be overemphasized. Ideally, it would be desirable to have a function which results in all equivalence classes having an equal number of records. The worst possible case is the one where all key values would be mapped into the same number (one equivalence class), thereby causing the insertion and fetching algorithms to be no more efficient than those in a linear search method. It is a nontrivial matter to obtain the "proper" hashing function, since the size of the equivalence classes which it induces depends on the key values being used (i.e., on the domain of the function). The functions used should not be too complex since the time taken to evaluate the function for a particular argument must be added to the insertion and fetch times. It is usually possible to live within this constraint because of the computer's great speed in doing arithmetic computations.

Another important factor in attaining reasonable efficiency is keeping the size of each equivalence class relatively small, but at the same time keeping the memory requirements to a minimum. For example, if it is known that, on an average, a file will have 125 entries at any given time, then one would ideally want approximately 125 equivalence classes. In such a case, the average number of comparisons necessary for accessing an entry would be no more than 2.

As mentioned earlier in this subsection, a symbol table is an example of a file. We shall now discuss by means of an example how variable names and labels can be inserted into a symbol table. Due to the random nature of insertions, the direct organization of files may best be applied. The division hashing function will be used together with the chaining method previously described. To simplify the discussion, the values associated with the entries in the symbol table will be omitted.

As an example, assume that the hashing function gives the following results:

The names AB12, LL, F, and LLLL are all mapped into the number 0.

The names A1 and DDCC are all mapped into the number 2.

The names B2 and AA are all mapped into the number 4.

The names VV and B are all mapped into the number 6.

The name C is mapped into the number 7.

Let us use a one-dimensional array called *EQUIV* consisting of 100 elements. Each element of *EQUIV* contains the address of the first node of the linked list representing a particular equivalence class. If there are no names in a certain

Scanned with OKEN Scanner

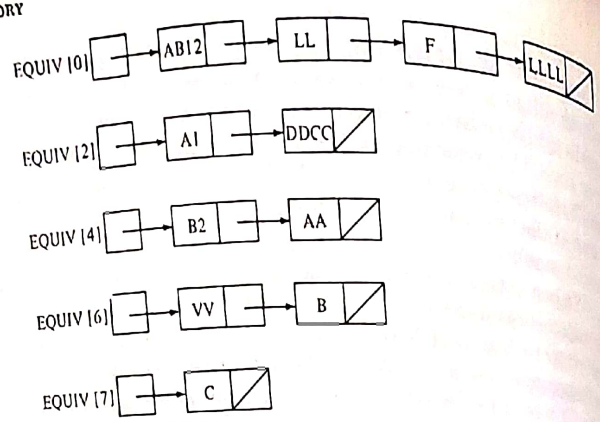


FIGURE 2-4.9

```
HASHING:
   PROCEDURE OPTIONS(MAIN);
/*THIS PROGRAM BUILDS A DICTIONARY OF NAMES CONSISTING OF 1 TO 12
  ALPHANUMERIC CHARACTERS.  EACH NAME IS HASHED INTO AN INTEGER
  WHICH IS BETWEEN 0 AND N-1 INCLUSIVE AND DENOTES TO WHICH OF N
  POSSIBLE EQUIVALENCE CLASSES NAME BELONGS.  PROCEDURE ENTER PUTS
  NAME IN THE SYMBOL FIELD OF A NODE WHICH IS PLACED AT THE END OF
  ONE OF THE LISTS POINTED TO BY AN ELEMENT OF THE POINTER ARRAY
  EQUIV.                                                        */
    DECLARE
        1 RECORD BASED(NEW),
            2 SYMBOL CHARACTER(12),
            2 LINK POINTER,
        NAME CHARACTER(12) VARYING,
        (I,N) BINARY FIXED,
        PTR POINTER;
      GET LIST(N); /* READ NUMBER OF EQUIVALENCE CLASSES */
    BEGIN; /* AUTOMATIC STORAGE ALLOCATION */
      DECLARE
          HASH RETURNS(BINARY FIXED),
          EQUIV(0:N - 1) POINTER;
       ON ENDFILE(SYSIN) GO TO OUTPUT;
       DO I = 0 TO N - 1;
          EQUIV(I) = NULL;
       END;
       PUT EDIT('NAME','HASHED INTO')(X(30),A(20),A(11));
    READ:  /* GET NEXT NAME FOR DICTIONARY*/
       GET LIST(NAME);
       CALL ENTER; /* INSERT NAME IN DICTIONARY*/
       GO TO READ;
    OUTPUT: /* OUTPUT CONTENTS OF EACH EQUIVALENCE CLASS */
       DO I = 0 TO N - 1;
          PUT SKIP(2) EDIT('EQUIVALENCE CLASS NUMBER ',I)(X(30),A(25),
                                                          F(2));
          PTR = EQUIV(I); /* SCAN LIST */
          DO WHILE (PTR ¬= NULL);
             PUT SKIP EDIT(PTR->SYMBOL)(X(41),A(12));
             PTR = PTR->LINK;
          END;
       END;
```

FIGURE 2-4.10 PL/I program demonstrating a hashing function.

equivalence class, then the corresponding element in the array *EQUIV* has a value of *NULL*. For the equivalence class consisting of the names AB12, LL, F, and LLLL which are mapped into zero, the diagram in Fig. 2-4.9 gives the structures. The other classes are also given.

We now turn to the formulation of an insertion algorithm for the symbole system.


```
ENTER:   PROCEDURE;
/*EACH ELEMENT OF ARRAY EQUIV IS A POINTER TO A LIST OF NAMES WHICH
  HAVE BEEN MAPPED INTO THE EQUIVALENCE CLASS REFERENCED BY THE
  SUBSCRIPT OF THE ELEMENT.  THIS PROCEDURE PUTS NAME IN A NODE AT THE
  END OF THE LIST WHICH BELONGS TO THE EQUIVALENCE CLASS DENOTED
  BY THE INTEGER RANDOM.*/
       DECLARE
            POINTER POINTER,
          RANDOM BINARY FIXED;
          RANDOM = HASH(NAME); /*COMPUTE THE HASH NUMBER*/
          PUT SKIP EDIT(NAME,RANDOM)(X(30),A(25),F(2));
          IF EQUIV(RANDOM) = NULL
          THEN /* EQUIVALENCE CLASS IS EMPTY */
              DO;
                   ALLOCATE RECORD;
                   EQUIV(RANDOM) = NEW;
                   NEW->SYMBOL = NAME;
                   NEW->LINK = NULL;
                   RETURN;
              END;
          POINTER = EQUIV(RANDOM); /* INITIATE SEARCH FOR NAME */
SEARCH:   /* OF LIST FOR AN OCCURRENCE OF NAME */
          IF POINTER->SYMBOL = NAME THEN RETURN;
          IF POINTER->LINK = NULL
          THEN /* INSERT A NEW NODE */
              DO;
                   ALLOCATE RECORD;
                   NEW->SYMBOL = NAME;
                   POINTER->LINK = NEW;
                   NEW->LINK = NULL;
                   RETURN;
              END;
          POINTER = POINTER->LINK; /* CONTINUE SEARCH */
          GO TO SEARCH;
   END ENTER;

   HASH: /* MAP NAME TO A NUMBER BETWEEN 0 AND N-1 */
          PROCEDURE(NAME) RETURNS(BINARY FIXED);
          DECLARE
              NAME CHARACTER(12) VARYING,
              BITNAME BIT(96) VARYING,
              NUMBER BINARY FIXED(31);
          BITNAME=UNSPEC(NAME); /* OBTAIN BIT REPRESENTATION OF NAME */
          IF LENGTH(BITNAME) > 24
          THEN NUMBER = SUBSTR(BITNAME,2,32);
          ELSE NUMBER = BITNAME;
          RETURN(MOD(NUMBER,N));
   END HASH;

   END; /* OF BEGIN BLOCK */

   END HASHING;
```

FIGURE 2-4.10 (Continued)

Algorithm *ENTER* (Enters a new name in a symbol table) Given a one-dimensional reference array *EQUIV*, each element of which contains a pointer to an equivalence class, and a hashing function *HASH*, which maps a name into an integer, it is required to append the entry denoted by *NAME* to the end of the appropriate equivalence class (if it is not already there). The typical node in the list representing an equivalence class consists of an information field and a link field denoted by *SYMBOL* and *LINK* respectively. This node structure is referred to as *RECORD*.

1 [Compute the hash number] Set $RANDOM \leftarrow HASH(NAME)$.

2 [Is the equivalence class empty?] If $EQUIV[RANDOM] \neq NULL$ then go to step 4.

3 [Enter *NAME* in empty class] Set $NEW \leftarrow RECORD$, *EQUIV*

Scanned with OKEN Scanner

$[RANDOM] \leftarrow NEW$, $SYMBOL(NEW) \leftarrow NAME$, $LINK(NEW) \leftarrow NULL$, and Exit.

4 [Initiate search for $NAME$] Set $POINTER \leftarrow EQUIV[RANDOM]$.

5 [Begin search] If $SYMBOL(POINTER) = NAME$ then Exit.

6 [End of equivalence class?] If $LINK(POINTER) = NULL$ then set $NEW \leftarrow RECORD$, $SYMBOL(NEW) \leftarrow NAME$, $LINK(POINTER) \leftarrow NEW$, $LINK(NEW) \leftarrow NULL$, and Exit; otherwise set $POINTER \leftarrow LINK(POINTER)$ and go to step 5.

```
NAME                    ////
ABSOLUTE              HASHED INTO
C                          5
DEFINED                    8
ENTER                      5
POINTER                    7
NEW                        4
SYMBOL                    10
LINK                       4
HASHING                    5
RANDOM                     7
MOD                        3
NAME                       7
END                        3
ENTER                      1
EQUIVALENCE                7
POINTER                    9
X                          4
Y                          0
Z                          1
                           2

EQUIVALENCE CLASS NUMBER 0
        X

EQUIVALENCE CLASS NUMBER 1
        END
        Y

EQUIVALENCE CLASS NUMBER 2
        Z

EQUIVALENCE CLASS NUMBER 3
        RANDOM
        NAME

EQUIVALENCE CLASS NUMBER 4
        POINTER
        SYMBOL

EQUIVALENCE CLASS NUMBER 5
        ABSOLUTE
        DEFINED
        LINK

EQUIVALENCE CLASS NUMBER 6

EQUIVALENCE CLASS NUMBER 7
        ENTER
        HASHING
        MOD

EQUIVALENCE CLASS NUMBER 8
        C

EQUIVALENCE CLASS NUMBER 9
        EQUIVALENCE

EQUIVALENCE CLASS NUMBER 10
        NEW
```

FIGURE 2-4.10 (Continued)

The previous algorithm is reasonably simple and requires no further comment.

The PL/I program in Fig. 2-4.10 constructs a symbol table. It consists of a hashing function, a procedure based on the preceding algorithm, and a mainline program. The hashing function finds the remainder on dividing the bit representation (obtained by using UNSPEC) of NAME by N. If NAME contains more than three characters, then only the first three are used in the process. The MOD function is used to find the remainder when NUMBER is divided by N.

The main program is written so that up to 25 equivalence classes can be handled in the symbol table. Each name to be inserted is on one input card. On an end-of-file, control transfers to the statement labeled OUTPUT. At this point, each simple linked list representing an equivalence class is scanned, and each name in it is printed.

EXERCISES 2-4.6

1 The midsquare hashing method follows:
   (a) Square part of the key, or the whole key if possible.
   (b) Either (1) extract n digits from the middle of the result to give $h(\text{key}) \in \{0, 1, \ldots, 10^n - 1\}$, or (2) extract n bits from the middle of the result to give $h(\text{key}) \in \{0, 1, \ldots, 2^n - 1\}$.

   Write a program which uses this procedure to hash a set of variable names. The midsquare method frequently gives satisfactory results, but in many cases the keys are unevenly distributed over the required range.

2 A hashing method often implemented, called *folding*, is performed by dividing the key into several parts and adding the parts to form a number in the required range. For example, if we have 8-digit keys and wish to obtain a 3-digit address, we may do the following:

$$h(97434658) = 974 + 346 + 58 = 378$$

$$h(31269857) = 312 + 698 + 57 = 67$$

   Note that the final carry is ignored.

   Implement this method in a computer program.

3 Compare the results of applying the division, midsquare, and folding hashing functions to a fixed set of keys. Make sure the range is the same or almost the same in each case. Which method distributes the keys most evenly over the elements of the range?

EXERCISES 2-4

1 Show that

$$f(A \cup B) = f(A) \cup f(B)$$

$$f(A \cap B) \subseteq f(A) \cap f(B)$$

Construct an example to show that in general it is not possible to replace $\subseteq$ by $=$ in the second relation. Under what condition will $f(A \cap B) = f(A) \cap f(B)$?

Scanned with OKEN Scanner

2 Show that $f: X \to Y$ is one-to-one iff any proper subsets of X are mapped into proper subsets of Y; that is, if $A \subset B \subseteq X$, then $f(A) \subset f(B) \subseteq Y$.

3 Let $\mathbf{I}_p = \{0, 1, \ldots, p-1\}$ and $f_r: \mathbf{I}_p \to \mathbf{I}_p$ be given by

$$f_r(x) = rx \pmod{p}$$

where $r = 0, 1, \ldots, p-1$. Show that f is not bijective for any r unless p is prime.

4 Determine the set of all bijective mappings on the set $\{1, 2\}$. Determine the identity element and inverses of each element under the composition of functions on this set.

5 Let $f: \mathbf{R} \to \mathbf{R}$ be given by $f(x) = x^2$ and $g: \mathbf{R} - \{2\} \to \mathbf{R}$ be given by $g(x) = x/(x-2)$. Find $f \circ g$. Is $g \circ f$ defined?

6 Let X be a set and $\rho(X)$ its power set. Show that the operations of union and intersection on $\rho(X)$ are both associative and commutative. Also every element of $\rho(X)$ is idempotent under these operations. Determine the zeroes, and show that the operations distribute over each other.

7 Show that the operation of symmetric difference $\triangle$ defined by

$$A \triangle B = (A \cup B) - (A \cap B)$$

is commutative and associative and has an identity element. Show that the inverse of A is A itself. Show that the operation of intersection, but not that of union, distributes over $\triangle$.

8 Consider the set $\mathbf{I}_+$ of positive integers and let $\mathbf{N}$ be the set of natural numbers (including zero). Observe that for any $x \in \mathbf{I}_+$ we have

$$x = 2^r(2s + 1)$$

for some $r, s \in \mathbf{N}$. This means that we can define a mapping $f: \mathbf{I}_+ \to \mathbf{N} \times \mathbf{N}$ such $f(x) = \langle r, s \rangle$, as indicated above. Show that f is one-to-one onto.

2-5 NATURAL NUMBERS

We are all familiar with the set of natural numbers and many of their properties. In this section we examine the set of natural numbers and introduce a method of generating such a set by starting with the null set and a function called the successor function. We then study some important properties (axioms) of the set of natural numbers. One of the axioms leads us to formulate the principle of mathematical induction. The use of natural numbers for counting enables us to define the notion of similarity, or equipotence, of two sets and the cardinal number of a set. The concepts of finite and infinite sets are introduced. These concepts have important applications in the theory of automata. Later in Chap. 3 we list the operations of addition and multiplication as well as the ordering relation on the set of natural numbers.

2-5.1 Peano Axioms and Mathematical Induction

The set $\mathbf{N} = \{0, 1, 2, 3, \ldots\}$ of natural numbers (including zero) can be generated by starting with a null set $\varnothing$ and the notion of a successor set. A *successor set* of a set A is denoted by A^+ and defined to be the set $A^+ = A \cup \{A\}$.

Let $\varnothing$ be the empty set, and obtain the successor sets $\varnothing^+$, $(\varnothing^+)^+$, $((\varnothing^+)^+)^+$, ... These sets are

$$\varnothing \quad \varnothing \cup \{\varnothing\} \quad \varnothing \cup \{\varnothing\} \cup \{\varnothing \cup \{\varnothing\}\} \quad \varnothing \cup \{\varnothing\} \cup \{\varnothing \cup \{\varnothing\}\} \cup \{\varnothing \cup \{\varnothing\} \cup \{\varnothing \cup \{\varnothing\}\}\}$$

These sets can be simplified to

$$\varnothing \quad \{\varnothing\} \quad \{\varnothing, \{\varnothing\}\} \quad \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\} \quad \cdots$$

If we rename the set $\varnothing$ as 0, then $\varnothing^+ = 0^+ = \{\varnothing\} = 1$, $1^+ = \{\varnothing, \{\varnothing\}\} = \{0, 1\} = 2$, and $2^+ = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\} = 3$, ..., and we get the set $\{0, 1, 2, 3, \ldots\}$ in which each element is a successor set of the previous element except for the element 0 which is assumed to be present. This discussion can be summarized by saying that the set of natural numbers can be obtained from the following axioms, known as Peano axioms.

1 $0 \in \mathbf{N}$ (where $0 = \varnothing$).
2 If $n \in \mathbf{N}$, then $n^+ \in \mathbf{N}$ where $n^+ = n \cup \{n\}$.
3 If a subset $S \subseteq \mathbf{N}$ possesses the properties
   (a) $0 \in S$, and
   (b) if $n \in S$, then $n^+ \in S$
   then $S = \mathbf{N}$.

Property 3 is known as the *minimality property* and asserts the fact that a minimal set satisfying (a) and (b) is the set of natural numbers. It is convenient to write n^+ as $n + 1$, although we need not necessarily consider $+$ as an operation on $\mathbf{N}$.

Property 3 is also the basis of the *principle of mathematical induction*, which is frequently employed in proofs. We shall put axiom 3 in an equivalent form.

3′ If $P(n)$ is any property (or predicate) defined over the set of natural numbers and (a) If $P(0)$ is *true*, (b) If $P(m) \Rightarrow P(m^+)$ for any $m \in \mathbf{N}$, then $P(n)$ holds for all $n \in \mathbf{N}$.

It is easy to see the equivalence of 3 and 3′ by choosing S in axiom 3 as $S = \{m \in \mathbf{N} \mid P(m)\}$.

The principle of mathematical induction is used in proving a collection of statements which can be put in one-to-one correspondence with the set of natural numbers. It was shown earlier in the theory of inference that a finite set of statements can be proved one after the other by showing $S_1 \Rightarrow S_2$, $S_2 \Rightarrow S_3$, $S_3 \Rightarrow S_4$, and so on. But such a method cannot be justified for an infinite number of statements.

In the formulation of the principle of mathematical induction as given in axiom 3′, we began with the element 0. However, this is not necessary as we can start with any natural number n_0. In this case, the conclusion is *true* for all $n \geq n_0$.

We shall give here several examples to illustrate the application of the principle of mathematical induction.

EXAMPLE 1 Show that $n < 2^n$.

SOLUTION Let $P(n): n < 2^n$.
(a) For $n = 0$, $P(0): 0 < 2^0 = 1$, so that $P(0)$ is *true*.
(b) For some arbitrary choice of $m \in \mathbf{N}$, assume that $P(m)$ holds, that is, $P(m)$:

Scanned with OKEN Scanner

$m < 2^m$. From this, by adding 1 to both sides, we get

$$m + 1 < 2^m + 1 < 2^m + 2^m = 2^m * 2 = 2^{m+1}$$

which is exactly $P(m+1)$. So $P(m) \Rightarrow P(m+1)$. Hence from the principle of mathematical induction, $P(n)$ is *true* for all $n \in \mathbf{N}$. ////

EXAMPLE 2 Show that $2^n < n!$ for $n \geq 4$.

SOLUTION Let $P(n): 2^n < n!$. Obviously $P(1)$, $P(2)$, $P(3)$ are not *true*. We also do not need them to be *true*. Now $P(4): 2^4 = 16 < 4! = 24$, so that $P(4)$ holds. Assume that $P(m)$ holds for any $m > 4$, and so

$$2^m < m!$$

Multiply both sides by 2 to get

$$2 * 2^m = 2^{m+1} < 2(m!) < (m+1) * (m!) = (m+1)!$$

This equation means that $P(m+1)$ holds. Hence $P(n)$ holds for all $n \in \mathbf{N}$ and $n \geq 4$. ////

EXAMPLE 3 Show that

$$B \cup (\bigcap_{i=1}^{n} A_i) = \bigcap_{i=1}^{n} (B \cup A_i)$$

SOLUTION Let $P(n): B \cup (\bigcap_{i=1}^{n} A_i) = \bigcap_{i=1}^{n} (B \cup A_i)$

For $n = 2$, $$B \cup (A_1 \cap A_2) = (B \cup A_1) \cap (B \cup A_2)$$

which follows from the distributive law of union and intersection. Note that here we are trying to prove that the distributive law holds over the intersection of any number of sets. Assume now that $P(m)$ holds for any m, so that

$$B \cup (\bigcap_{i=1}^{m} A_i) = \bigcap_{i=1}^{m} (B \cup A_i)$$

Now

$$B \cup (\bigcap_{i=1}^{m+1} A_i) = B \cup (\bigcap_{i=1}^{m} A_i \cap A_{m+1})$$

$$= (B \cup (\bigcap_{i=1}^{m} A_i)) \cap (B \cup A_{m+1})$$

using $P(2)$ for sets $\bigcap_{i=1}^{m} A_i$ and A_{m+1}

$$= (\bigcap_{i=1}^{m} (B \cup A_i)) \cap (B \cup A_{m+1})$$

$$= \bigcap_{i=1}^{m+1} (B \cup A_i)$$

Here we have used the fact that $P(2)$ and $P(m)$ hold, and we have shown that $P(m+1)$ holds. Hence $P(n)$ is *true* for all n. ////

EXAMPLE 4 Show that $n^3 + 2n$ is divisible by 3.

SOLUTION Let $P(n): n^3 + 2n$ is divisible by 3. Now $P(0): 0$ is divisible by 3, so that $P(0)$ is *true*. Let us assume for any m, $P(m)$ is *true*, that is, $m^3 + 2m$ is divisible by 3.

Now
$$\begin{aligned}(m+1)^3 + 2(m+1) &= m^3 + 3m^2 + 3m + 1 + 2m + 2 \\ &= m^3 + 3m^2 + 5m + 3 \\ &= m^3 + 2m + 3(m^2 + m + 1)\end{aligned}$$

Since $m^3 + 2m$ is assumed to be divisible by 3 and $3(m^2 + m + 1)$ is also divisible by 3, $(m+1)^3 + 2(m+1)$ is divisible by 3; that is, $P(m+1)$ is true. Hence $P(n)$ is *true* for all $n \in \mathbf{N}$. ////

The definition of well-formed formula given in Sec. 1-2.7 and the definition of cartesian product of n sets in Sec. 2-1.9 were inductive definitions. In general, an inductive definition of a property or set P is given in the following manner.

1 Given a finite set A whose elements have the property P.
2 The elements of a set B, all of which are constructed from A. satisfy the property P.
3 The elements constructed as in (1) and (2) are the only elements satisfying the property P.

In fact, the set of natural numbers is defined in a similar manner. We shall make use of the inductive definition in defining a class of functions known as recursive functions in Sec. 2-6.1.

In general, an inductive definition of a property, or more precisely a set P, consists of three main steps. The first step consists of naming the elements of a set, say A, to be the *basic* elements of P. The next step consists of a set of rules which define how other elements of P can be obtained from the elements of A by means of some n-ary operations. This step is called the *inductive step*. Finally, the last step, which is often omitted, states that P consists of only those elements which are obtained according to steps 1 and 2. It is assumed that all objects in the definition are the elements of a universal set.

The definition of the set $\mathbf{N}$ of natural numbers given here was of this form. Also the definition of well-formed formulas was of this form, in which the inductive step contained the unary operation $\neg$ and the binary operations $\wedge$, $\vee$, $\rightarrow$, and $\rightleftarrows$.

EXAMPLE 5 Find the set given by the following definition:

1 $3 \in P$.
2 For $x, y \in P$, $x + y \in P$.
3 Only those elements obtained from steps (1) and (2) are in P.

SOLUTION The set P consists of positive integers which are multiples of 3. ////

Scanned with OKEN Scanner

EXAMPLE 6 Give an inductive definition of the set

$$P = \{2, 3, 4, \ldots\} = \mathbf{N} - \{0, 1\}$$

SOLUTION

1 $2 \in P$ and $3 \in P$.
2 If $x, y \in P$, then $x + y \in P$.
3 Only those elements obtained from steps 1 and 2 are in P. ////

EXAMPLE 7 Show that all the functions of the form $f(x) = x + a$ for $a \in \mathbf{N}$ are defined by the following. It is assumed that the universal set is $\mathbf{N}$.

1 $I(x) = x$ and $S(x) = x + 1$ are in P.
2 For $f, g \in P$, $f \circ g$ is also in P.
3 Only those functions obtained from steps (1) and (2) are in P.

SOLUTION It is easy to see by induction that $f(x) = x + a$ is in P, because $I(x) = x + 0$ is in P and if $f_k(x) = x + k$, $k \in \mathbf{N}$, is in P, then $(S \circ f_k)(x) = S(x + k) = (x + k) + 1$ or $f_{k+1}(x)$ is also in P. Hence $f(x) = x + a$ is in P. Now if $f(x) = x + a$ and $g(x) = x + b$, then

$$(f \circ g)(x) = f(x + b) = x + a + b = x + (a + b)$$

which is also of the form $x + a$. This guarantees that step 2 generates only the functions which are required. ////

EXAMPLE 8 Give an inductive definition of the well-formed formulas of set theory involving the operations $\cap$, $\cup$, and $\sim$.

SOLUTION

1 The symbols $A, B, C, \ldots$ are wff's.
2 (a) If X is a wff, so is $\sim X$.
(b) If X and Y are wff's, so are $(X \cup Y)$ and $(X \cap Y)$.
3 The only wff's are those obtained by using steps 1 and 2. ////

2-5.2 Cardinality

In the previous section we were concerned with the generation of the set of natural numbers. We are all familiar with a useful application of the set of natural numbers in counting. This property of natural numbers is used in measuring the "size" of a set and in comparing the "sizes" of any two sets.

The first question that we should examine is how do we count, say the number of people in a room, the number of books on a shelf, or the number of elements in a set. What we do is establish a one-to-one correspondence between the objects to be counted and the set of integers $\{1, 2, 3, \ldots, n\}$. From this correspondence we say that the number of objects is n. We now generalize this concept.

Definition 2-5.1 Two sets A and B are said to be *equipotent* (or *equivalent* or to have the same *cardinality*, or to be *similar*) and written as $A \sim B$ if and only if there is one-to-one correspondence between the elements of A and those of B.

Note that one-to-one correspondence can be established by showing a mapping $f: A \to B$ which is one-to-one and onto.

EXAMPLE 1 Let $\mathbf{N} = \{0, 1, 2, \ldots\}$ and $\mathbf{N}_2 = \{0, 2, 4, \ldots\}$. Show that $\mathbf{N} \sim \mathbf{N}_2$.

SOLUTION Let $f: \mathbf{N} \to \mathbf{N}_2$ be defined by $f(n) = 2n$, $n \in \mathbf{N}$. The function f establishes a one-to-one correspondence between $\mathbf{N}$ and $\mathbf{N}_2$. Hence $\mathbf{N} \sim \mathbf{N}_2$. Note that $\mathbf{N}_2 \subset \mathbf{N}$. ////

EXAMPLE 2 Let P be the set of all positive real numbers and S be the subset of P given by $S = \{x \mid x \in P \wedge 0 < x < 1\}$. Show that $S \sim P$.

SOLUTION Let $f: P \to S$ be defined by $f(x) = x/(1 + x)$ for $x \in P$. Obviously the range of f is in S. Further, for any $y \in S$, we have $x = y/(1 - y)$, so that f is one-to-one and onto. ////

The one-to-one correspondence established in showing that any two sets are equipotent is not unique. However, any one-to-one correspondence is enough to show that the two sets are equipotent.

It is easy to see that equipotence is an equivalence relation on a family of sets and hence partitions the family of sets into equivalence classes.

Definition 2-5.2 Let F be a family of sets and let $\sim$ denote the relation of equipotence on F. The equivalence classes of F under the relation $\sim$ are called *cardinal numbers*. For any set $A \in F$, the equivalence class to which A belongs is denoted by $[A]$ or $\bar{A}$ or Card A and is called the *cardinal number of* A. For $A, B \in F$

$$\bar{A} = \bar{B} \Leftrightarrow A \sim B$$

We shall now show how the cardinal numbers of some sets can be represented by means of natural numbers. Recall that the natural numbers have been defined as sets, starting from the null set and then constructing the successor sets. The cardinal numbers are also defined as sets. We shall first start with the empty set and denote its cardinal number by 0. For our present discussion, we shall denote the cardinal number of a set A by $k(A)$, so that $k(\varnothing) = 0$. If an element $p \notin A$, where A is some set with the cardinal number $k(A)$, then the cardinal number of the set $A \cup \{p\}$, that is, $k(A \cup \{p\})$, can be written as $[k(A)]^+$ or $k(A) + 1$. This notation is justified by observing that if $A \sim B$, so that $k(A) = k(B)$, and if $p \notin A$ and $q \notin B$, then $k(A \cup \{p\}) = k(B \cup \{q\})$. The last equality follows from the fact that a one-to-one correspondence can be established between the sets $A \cup \{p\}$ and $B \cup \{q\}$. By following this convention, we can build sets starting from the null set and building successive unions such that the cardinalities of these sets can be represented by the natural numbers. For example, let $A_1 = \{a\}$, $A_2 = \{a, b\}$, $A_3 = \{a, b, c\}$, ... We then have

$A_1 = \{a\} \cup \varnothing$ hence $k(A_1) = 0^+ = 1$
$A_2 = \{a, b\} = A_1 \cup \{b\}$ hence $k(A_2) = 1^+ = 2$
$A_3 = \{a, b, c\} = A_2 \cup \{c\}$ hence $k(A_3) = 2^+ = 3$

Scanned with OKEN Scanner

and so on. Accordingly, the cardinal number of a set containing n elements can be denoted by the natural number n.

It is not possible to represent the cardinality of every set by means of natural numbers because there are sets which cannot be built up by successive unions, as was done above. In any case, we now have a definition of a finite set as follows.

Definition 2-5.3 Any set whose cardinal number is a natural number is a *finite set*. Also any set which is not finite is called an *infinite set*.

Definition 2-5.4 Any set which is equipotent to the set of natural numbers is called *denumerable*.

The cardinality of a denumerable set is denoted by the symbol $\aleph_0$ called aleph null. We shall restrict the use of $k(A)$ to denote only the cardinality of a finite set A.

Definition 2-5.5 A set is called *countable* if it is finite or denumerable, and a set is called *nondenumerable* if it is infinite and not denumerable.

An important difference between a finite and an infinite set is that no proper subset of a finite set can be equipotent to the set itself, because a one-to-one correspondence between the elements of such sets is impossible. However, for infinite sets this is not necessarily the case, as can be seen from some of the examples given here. Some authors take these as definitions of finite and infinite sets. We shall now prove a theorem about infinite sets which are denumerable.

Theorem 2-5.1 An infinite subset of a denumerable set is also denumerable.

PROOF Let S be an infinite subset of a given denumerable set A. Obviously $A \sim \mathbf{N}$, and there exists a one-to-one correspondence between the elements of A and the set $\mathbf{N}$ of natural numbers. Consequently, we have a function $f: \mathbf{N} \to A$ given by $f(n) = x$ for $x \in A$, which is one-to-one and onto. Thus the elements of A can be arranged as $f(1), f(2), \ldots$. Now, delete from this list those elements which are not present in S. The number of the remaining elements is still infinite because S is infinite. Let us denote these elements by $f(i_1), f(i_2), \ldots$. Define a function $g: \mathbf{N} \to S$ such that $g(n) = f(i_n)$; then g is one-to-one and onto S, so that S is also denumerable. ////

EXAMPLE 3 Show that the set of integers, positive, negative, and zero is denumerable.

SOLUTION Let $\mathbf{I} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. If we enumerate the elements of $\mathbf{I}$, we can establish the following one-to-one correspondence between $\mathbf{I}$ and $\mathbf{N}$.

$$\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & \ldots \\ \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \ldots \\ 0 & -1 & 1 & -2 & 2 & -3 & \ldots \end{array}$$

This correspondence shows that $\mathbf{I} \sim \mathbf{N}$. Alternatively, we can define a function $f: \mathbf{N} \to \mathbf{I}$ where

$$f(n) = \begin{cases} \dfrac{n}{2} & \text{if } n \text{ is even} \\ -\dfrac{n+1}{2} & \text{if } n \text{ is odd} \end{cases}$$

This mapping is one-to-one and onto, hence $\mathbf{I} \sim \mathbf{N}$. ////

A sequence which is used to establish a one-to-one correspondence with the elements of a set is called an *enumeration*. In Example 3 we used the sequence $0, -1, 1, -2, \ldots$ as an enumeration of the integers.

EXAMPLE 4 Show that the set $\mathbf{N} \times \mathbf{N}$ is denumerable. Hence the set of positive rational numbers, as well as the set of rational numbers, is also denumerable.

SOLUTION Write the elements of $\mathbf{N} \times \mathbf{N}$ as shown in Table 2-5.1. We now arrange the elements of $\mathbf{N} \times \mathbf{N}$ in the order shown by the arrows, viz,

$$\langle 0,0\rangle \quad \langle 0,1\rangle \quad \langle 1,0\rangle \quad \langle 0,2\rangle \quad \langle 1,1\rangle \quad \langle 2,0\rangle \quad \langle 0,3\rangle \quad \langle 1,2\rangle \quad \langle 2,1\rangle \quad \langle 3,0\rangle \quad \ldots$$

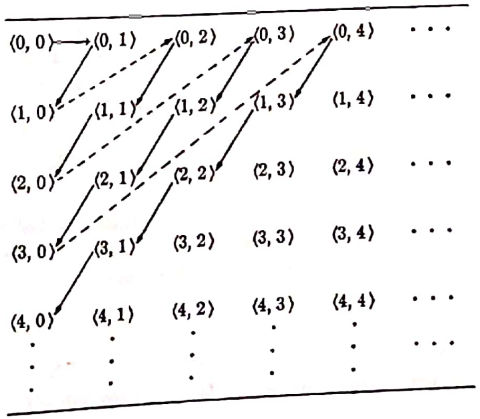
and establish a one-to-one correspondence between these elements and the elements of $\mathbf{N}$. The one-to-one correspondence can be expressed by $f: \mathbf{N} \times \mathbf{N} \to \mathbf{N}$

$$f(m, n) = \tfrac{1}{2}(m + n + 1)(m + n) + m$$

which shows that $\mathbf{N} \times \mathbf{N}$ is denumerable.

Let us now obtain a subset S of $\mathbf{N} \times \mathbf{N}$ by deleting all pairs (m, n) in which m and n are not relatively prime (that is, m and n have a common factor which is an integer greater than 1). Obviously S contains at least the elements $\langle 1, 1\rangle$,

Table 2-5.1



Scanned with OKEN Scanner

$\langle 2, 1\rangle, \langle 3, 1\rangle, \ldots$, so that S is infinite. Since $\mathbf{N} \times \mathbf{N}$ is infinite and denumerable and S is an infinite subset of $\mathbf{N} \times \mathbf{N}$, S is denumerable. The set S is equipotent to the set of positive rational numbers $\mathbf{Q}^+$ by observing the correspondence $\langle m, n\rangle \leftrightarrow m/n$. Therefore $\mathbf{Q}^+$ is denumerable. ////

From this example, it may appear that all the infinite sets are denumerable, but such is not the case. For example, we shall show that the set of real numbers is not denumerable. Our proof is based upon Cantor's diagonal argument and the indirect method. We first assume that the set $\mathbf{R}$ is denumerable and then show that a member of the set different from all those enumerated exists, showing that the enumeration is not exhaustive, and hence arriving at a contradiction. The argument is called "diagonal" because to obtain this particular member, we move along the diagonal of an array. The diagonal argument is employed frequently in the theory of automata and other logical investigations. This method of proof can be used for showing the nondenumerability of other sets as well.

Theorem 2-5.2 Show that the set $\mathbf{R}$ of real numbers is not denumerable.

PROOF

(a) It is sufficient to show that the set S given by

$$S = \{x \mid (x \in \mathbf{R}) \wedge (0 < x < 1)\}$$

is nondenumerable because $S \sim \mathbf{R}$ follows from the mapping $f\colon S \to \mathbf{R}$ given by (see also Example 2)

$$f(x) = \begin{cases} \dfrac{1}{2x} - 1 & 0 < x \leq \tfrac{1}{2} \\ \dfrac{1}{2(x-1)} + 1 & \tfrac{1}{2} \leq x < 1 \end{cases}$$

which is one-to-one and onto and

$$f^{-1}(x) = \begin{cases} \dfrac{1}{2(x+1)} & x \geq 0 \\ \dfrac{1}{2(x-1)} + 1 & x \leq 0 \end{cases}$$

(b) Let us assume that S is denumerable, so that we can arrange the elements of S in an infinite sequence $s_1, s_2, \ldots, s_n, \ldots$. Now we know that any positive number less than 1 can be expressed as

$$s = 0.y_1y_2y_3\ldots$$

where $y_i \in \{0, 1, 2, \ldots, 9\}$ and $\{y_1, y_2, \ldots\}$ has an infinite number of nonzero elements. This statement is true because numbers such as 0.2 and 0.123 can be written as $0.1999\ldots$ and $0.122999\ldots$ respectively. In view of this fact, we express the elements $s_1, s_2, \ldots$ as

$$\begin{aligned} s_1 &= 0.a_{11}a_{12}a_{13}\ldots a_{1n}\ldots \\ s_2 &= 0.a_{21}a_{22}a_{23}\ldots a_{2n}\ldots \\ s_3 &= 0.a_{31}a_{32}a_{33}\ldots a_{3n}\ldots \\ &\ldots\ldots\ldots\ldots \end{aligned}$$

Next, let us construct a real number

$$r = 0.b_1b_2b_3\ldots b_n\ldots$$

by choosing $b_j = 1$ if $a_{jj} \neq 1$ and $b_j = 2$ if $a_{jj} = 1$, for $j = 1, 2, 3, \ldots$. Obviously r is different from all the numbers $s_1, s_2, s_3, \ldots$ because it differs from s_1 in position 1, from s_2 in position 2, and so on. This shows that $r \notin S$, which is a contradiction. Hence S is nondenumerable. ////

This result is true for the set of real numbers lying in any interval (a, b) where $b > a$. The cardinality of all these sets which are equipotent is denoted by $\mathfrak{c}$ and is called the *power of continuum*. The sets $\mathbf{R}^2, \mathbf{R}^3, \ldots$ also have the cardinality $\mathfrak{c}$.

So far, we have seen examples of infinite sets which are denumerable and have cardinality $\aleph_0$. On the other hand, we have also seen examples of infinite sets which are nondenumerable and have cardinality $\mathfrak{c}$. There are several questions which can be asked at this stage. Are there other infinite sets whose cardinal numbers are different from $\aleph_0$ and $\mathfrak{c}$? Can we compare the cardinal numbers and arrange them in some order? Before we answer some of these questions, we introduce an ordering relation on the family of subsets of the universal set and a corresponding ordering on the set of cardinal numbers.

Definition 2-5.6 If A and B are sets such that A is equipotent to a subset of B, then we say that A is *dominated* by B or A *precedes* B and write $A \preceq B$. If α and β denote the cardinal numbers of the sets A and B, respectively, and if $A \preceq B$, then we say that α is *less than or equal to* β. Symbolically,

$$A \preceq B \Leftrightarrow \alpha \leq \beta$$

The choice of the term "less than or equal to" to express the relation on the set of cardinal numbers is based on the fact that for finite sets, $A \preceq B$ implies that the natural number representing $k(A)$ is less than or equal to the natural number representing $k(B)$.

From the definition, it is clear that the relations $\preceq$ and $\leq$ are both reflexive and transitive. These relations are also antisymmetric. This fact follows from a theorem known as the Schröder-Bernstein theorem, which we do not prove here. In any case, these two relations are partial ordering relations. It can also be shown that they are total orderings on the respective sets. Thus for any two sets A and B, we have either $A \preceq B$ or $B \preceq A$. Associated with the ordering relations $\preceq$ and $\leq$ we also have the relations $\prec$ and $<$ given by

$$\begin{aligned} A \prec B &\Leftrightarrow A \preceq B \quad \text{and} \quad A \not\sim B \\ \alpha < \beta &\Leftrightarrow \alpha \leq \beta \quad \text{and} \quad \alpha \neq \beta \end{aligned}$$

Scanned with OKEN Scanner

Since the set of natural numbers is a proper subset of the real numbers, we have $\mathbf{N} < \mathbf{R}$ and $\aleph_0 < \mathfrak{c}$. Now we return to the question of whether for a given set we can find another set whose cardinality is greater than that of the given set. For finite sets, such a construction is easy and was given earlier. For infinite sets, a theorem due to Cantor shows the existence of such sets.

Theorem 2-5.3 For any set A, $A < 2^A$ where 2^A is the power set of A. If α is the cardinality of A and 2^α denotes the cardinality of 2^A, then $\alpha < 2^\alpha$.

PROOF Let $f: A \to 2^A$ be defined by $f(a) = \{a\}$ for every $a \in A$. Obviously f is one-to-one, and hence $A \preceq 2^A$. To show that $A \not\sim 2^A$, we shall use the indirect method of proof. Assume that a function $g: A \to 2^A$ exists which is one-to-one and onto. If $a \in g(a)$, we call a an "interior member" of A. Similarly, if $a \notin g(a)$, then we call a an "exterior member" of A. Now let a set B be the set of exterior members of A, that is,

$$B = \{x \mid (x \in A) \wedge (x \notin g(x))\}$$

Obviously $B \subseteq A$ and hence $B \in 2^A$. Since g is onto, there must be an element $b \in A$ such that $g(b) = B$. Now two cases exist: either $b \in B$, so that $g(b) = B$ and b is an interior member, which is a contradiction; or $b \notin B$ and hence $b \in g(b) = B$, which is again a contradiction. Hence $g: A \to 2^A$ is not one-to-one and onto. Therefore $A \not\sim 2^A$ and $A < 2^A$. ////

Although we do not prove it here, note that for the set $\mathbf{N}$ of natural numbers the cardinality of the power set 2^N is $\mathfrak{c}$, and hence 2^N is also nondenumerable. There are many interesting theorems regarding infinite sets, but we shall not discuss them here.

The concepts discussed in this section are used extensively in computer science. In particular, the proof technique of diagonalization is used extensively in computability theory. In the areas of formal languages and automata we will be concerned with domains other than the natural numbers. A one-to-one correspondence between elements in a nonarithmetic system and the natural numbers can be made. This type of numbering (called Gödel numbering) permits statements about elements in the nonnumeric system to be transformed into corresponding statements about natural numbers. Since much is known about the natural numbers, we can indirectly prove assertions made in a nonnumeric system by proving the corresponding assertions in the natural number system.

Gödel numbering is one technique used for establishing a one-to-one correspondence between nonnumeric elements and a subset of the natural numbers. This method is based on the existence of a unique representation of any natural number as a product of successive prime numbers raised to certain powers. There are other methods available that provide a one-to-one mapping of nonnumeric items onto the natural numbers, methods which may well be simpler than Gödel numbering.

One such method makes use of n-adic number systems. An n-adic number system, for $n > 0$, is a positional number system in which a number is written as a sequence of digits, each selected from the set $\{1,2,\cdots,n\}$, and the weight value associated with each digit position is n raised to some nonnegative power.

The n-adic number $d_m d_{m-1} \cdots d_2 d_1$ represents the value $\sum_{i=1}^{m} d_i n^{i-1}$. For $n = 2$, these numbers are called *dyadic numbers*, and an m-digit dyadic number has the value $d_1 + 2d_2 + 4d_3 + \cdots + 2^{m-1}d_m$. Thus, the dyadic number 12221 represents the value 45 (because $45 = 1 + 2 \cdot 2 + 4 \cdot 2 + 8 \cdot 2 + 16 \cdot 1$). Note that for $n = 1$, we have a monadic number system in which the value of the monadic number is given by the number of digits used to represent it. Such a system is commonly called a *tally system*.

Clearly, the set of n-adic numbers, for any $n > 0$, can be put in one-to-one correspondence with the positive integers. The number zero, however, cannot be represented in an n-adic system as we have defined it. But we can extend an n-adic number system to include "0" as an n-adic number symbol representing the value zero, though we do not permit "0" to be a digit. The n-adic numbers, thus augmented by "0," can now be mapped onto the natural numbers.

Any nonnumeric item can be considered a finite sequence of symbols, each selected from a finite set of symbols $\{s_1, s_2, \ldots, s_n\}$. To each symbol s_i we assign the number i. The set $\{1, 2, \ldots, n\}$ can then be considered the set of digits of an n-adic number system, and each nonnumeric item can be translated into an n-adic number by replacing each symbol s_i in the sequence by its assigned digit i. (A sequence of length zero is translated into the n-adic number 0.) Because the n-adic numbers are one-to-one with the natural numbers, we now have a one-to-one mapping of the nonnumeric items onto the natural numbers. We shall make use of this translation technique in Sec. 6-2.

EXERCISES 2-5

1 Show that $S(n) = 1 + 2 + \cdots + n = n(n+1)/2$.

2 Prove that

$$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \cdots + \frac{1}{n(n+1)} = \frac{n}{n+1}$$

3 Show that

$$2 + 2^2 + 2^3 + \cdots + 2^n = 2^{n+1} - 2$$

4 Show that the set of ordered pairs $\mathbf{N} \times \mathbf{N}$ is countable. (*Hint*: Use the result of Problem 8 in Exercises 2-4.)

5 Show that the following intervals have the power of the continuum: $[a, b]$, $[a, b)$, (a, b), $(a, b]$.

6 If two sets A_1 and A_2 are both denumerable and disjoint, show that $A_1 \cup A_2$ is also denumerable. Using mathematical induction, show also that the union of a finite number of sets which are denumerable is also denumerable. (*Hint*: Map A_1 on $\mathbf{N} \times \{0\}$ and A_2 on $\mathbf{N} \times \{1\}$.)

7 If two sets A_1 and A_2 are both denumerable, then show that $A_1 \times A_2$ is also denumerable.

8 Show that $\mathbf{R} \times \mathbf{R}$ has the power of the continuum.

9 It is known that every real number x, $0 \leq x \leq 1$, has a *p-adic* expansion $\sum_{i=1}^{\infty} n_i p^{-i}$, where $p > 1$, $p \in \mathbf{N}$, and $0 \leq n_i \leq p - 1$. In particular, if $p = 2$, the expansion is called a *dyadic expansion*. Such an expansion is unique, unless x is a nonzero number of the form $\sum_{i=1}^{M} n_i p^{-i}$. In such case, there are two possible expansions: one of which is finite as shown, and the other is infinite. If we agree to use the infinite expansion

Scanned with OKEN Scanner

in the latter case, show that the set $C = \{x \in \mathbf{R} \mid 0 \leq x \leq 1\}$ is such that $C \subseteq 2^{\mathbf{N}}$. Also show that the cardinality of $2^{\mathbf{N}}$ is C.

10 Given the set of symbols $\{a, b, c\}$ and the correspondence $f(a) = 1, f(b) = 2, f(c) = 3$, write the triadic numbers and natural numbers associated with the items cab, $bbca$, abc.

11 Given the symbols $\{P, L, E, K, H\}$ and the correspondence $f(E) = 1, f(H) = 2, f(K) = 3, f(L) = 4, f(P) = 5$, along with the pentadic (5-adic) number system, what is the nonnumeric string associated with the natural number 300?

12 Prove that the set of natural numbers is in one-to-one correspondence with the set of symbols of the form $'c_1 c_2 \ldots c_n'$, $n = 1, 2, \ldots, \in \{a, b\}$.

2-6 RECURSION

We have already seen a procedure in Sec. 1-4 by which one can prove the validity of a conclusion in a mechanical way, i.e., by following a set of rules. Such mechanical procedures are needed if an end result is to be obtained by using a machine. In fact, automata theory is concerned with the study of mathematical models of computing devices (or machines) and the types of problems that can be solved by such machines. Given a particular problem, the standard procedure for determining whether it is mechanically "solvable" is to reduce the problem (by Gödel numbering, say) to an equivalent one consisting of a function on the natural numbers and then to decide whether such a function can be evaluated by mechanical means. Functions that can be evaluated by mechanical means are said to be "effectively computable" or "simply computable." We discuss computable functions at some length in Chap. 6. In this section we first define a class of functions inductively and show that any such function can be evaluated in a purely mechanical manner. The notions of recursive sets and predicates are also introduced. The more general concept of the decision problem associated with a set is briefly discussed. Finally, a discussion of recursion in computer programming is given.

2-6.1 Recursive Functions, Sets, and Predicates

In this section we study an important class of functions called recursive functions. We shall restrict ourselves to only those functions whose arguments and values are natural numbers. Such functions are called *number-theoretic*. In general, we shall consider number-theoretic functions of n variables which will be denoted as $f\langle x_1, x_2, \ldots, x_n\rangle$. From now on, we shall not mention the fact—although it is assumed throughout—that all functions are number-theoretic. Any function $f: \mathbf{N}^n \to \mathbf{N}$ is called *total* because it is defined for every n-tuple in $\mathbf{N}^n$. On the other hand, if $f: D \to \mathbf{N}$ where $D \subseteq \mathbf{N}^n$, then f is called *partial*. Examples of such functions are

1 $f\langle x, y\rangle = x + y$, which is defined for all $x, y \in \mathbf{N}$ and hence is a total function.

2 $g\langle x, y\rangle = x - y$, which is defined for only those $x, y \in \mathbf{N}$ which satisfy $x \geq y$. Hence $g\langle x, y\rangle$ is partial.

Every total function of n variables is also an n-ary operation on $\mathbf{N}$ according to our definition.

We now give a set of three functions called the *initial functions*, which are used in defining other functions by induction.

$$Z: Z(x) = 0 \qquad \textit{zero function}$$

$$S: S(x) = x + 1 \qquad \textit{successor function}$$

$$U_i^n: U_i^n\langle x_1, x_2, \ldots, x_n\rangle = x_i \qquad \textit{projection function}$$

The projection function is also called the *generalized identity function*. As examples, we have $U_1^2\langle x, y\rangle = x$, $U_2^3\langle 2, 4, 6\rangle = 4$, etc.

The operation of composition will be used to generate other functions. Composition of functions was defined in Sec. 2-4.2 for functions of one variable. The same idea can be extended to functions of more than one variable. For example, let $f_1\langle x, y\rangle$, $f_2\langle x, y\rangle$, and $g\langle x, y\rangle$ be any three functions. The composition of g with f_1 and f_2 is a function h given by

$$h\langle x, y\rangle = g\langle f_1\langle x, y\rangle, f_2\langle x, y\rangle\rangle$$

Naturally, for h to be nonempty, it is necessary that the domain of g include $R_{f_1} \times R_{f_2}$ where R_{f_1} and R_{f_2} are the ranges of f_1 and f_2, respectively. Also the domain of h is $D_{f_1} \cap D_{f_2}$, where D_{f_1} and D_{f_2} are the domains of f_1 and f_2 respectively. If f_1, f_2, and g are total, then h is also total. In general, let $f_1, f_2, \ldots, f_n$ each be partial functions of m variables, and let g be a partial function of n variables. Then the composition of g with $f_1, f_2, \ldots, f_n$ produces a partial function h given by

$$h\langle x_1, \ldots, x_m\rangle = g\langle f_1\langle x_1, \ldots, x_m\rangle, \ldots, f_n\langle x_1, \ldots, x_m\rangle\rangle$$

It is assumed that the domain of g includes the n-tuples $\mathbf{X}_{i \in I_n} R_{f_i}$, $\mathbf{I}_n = \{1, 2, \ldots, n\}$ and R_{f_i} denotes the range of f_i. Also the domain of h is given by $\bigcap_{i \in I_n} D_{f_i}$ where D_{f_i} is the domain of f_i. The function h is total iff $f_1, f_2, \ldots, f_n$ and g are total.

As an example, let

$$f_1\langle x, y\rangle = x + y \qquad f_2\langle x, y\rangle = xy + y^2 \qquad g\langle x, y\rangle = xy$$

Then

$$\begin{aligned} h\langle x, y\rangle &= g\langle f_1\langle x, y\rangle, f_2\langle x, y\rangle\rangle \\ &= g\langle x + y, xy + y^2\rangle \\ &= (x + y)(xy + y^2) \end{aligned}$$

Here h is total, because f_1, f_2, and g are all total.

Given a function $f\langle x_1, x_2, \ldots, x_n\rangle$ of n variables, it may be convenient to consider $n - 1$ of the variables as fixed and vary only the remaining variable over the set of natural numbers or over a subset of it. For example, we may treat x as fixed and vary y in $f\langle x, y\rangle$ to obtain $f\langle x, 0\rangle$, $f\langle x, 1\rangle$, $f\langle x, 2\rangle$, ... In a mechanical process of computing the value of a function, this procedure can be used conveniently although it appears too cumbersome for hand computation. Thus, to determine $f\langle 2, 3\rangle$, where $f\langle x, y\rangle = x + y$, we assume that $f\langle 2, 0\rangle = 2$ is given and then proceed to evaluate $f\langle 2, 1\rangle$, $f\langle 2, 2\rangle$, and finally $f\langle 2, 3\rangle$. Each functional value (except $f\langle 2, 0\rangle$) is computed by adding 1 to the previous value

Scanned with OKEN Scanner

until the desired result is obtained. The computation for $f\langle 2, 3\rangle$ is

$$f\langle 2,3\rangle = [(f\langle 2,0\rangle + 1) + 1] + 1$$
$$= [(2+1)+1]+1 = (3+1)+1 = 4+1 = 5$$

It is assumed that we have a mechanism by which we can determine the value of the function when an argument is zero and also its value for the argument $n+1$ from the value of the function when the argument is n. Such a procedure will now be described. The arguments which are considered to be fixed are called *parameters*, while the one which is assumed to vary is considered a variable for that discussion.

The following operation which defines a function $f\langle x_1, x_2, \ldots, x_n, y\rangle$ of $n+1$ variables by using two other functions $g\langle x_1, x_2, \ldots, x_n\rangle$ and $h\langle x_1, x_2, \ldots, x_n, y, z\rangle$ of n and $n+2$ variables, respectively, is called *recursion*.

$$f\langle x_1, x_2, \ldots, x_n, 0\rangle = g\langle x_1, x_2, \ldots, x_n\rangle$$

$$f\langle x_1, x_2, \ldots, x_n, y+1\rangle = h\langle x_1, x_2, \ldots, x_n, y, f\langle x_1, x_2, \ldots x_n, y\rangle\rangle$$

In this definition, the variable y is assumed to be the inductive variable in the sense that the value of f at $y+1$ is expressed in terms of the value of f at y. The variables $x_1, x_2, \ldots, x_n$ are treated as parameters and are assumed to remain fixed throughout the definition. Also it is assumed that both the functions g and h are known. We shall now impose restrictions on g and h which will guarantee that the function f which is defined recursively, as above, can actually be computed and is total. The operation of recursion will be used frequently in this section.

Definition 2-6.1 A function f is called *primitive recursive* iff it can be obtained from the initial functions by a finite number of operations of composition and recursion.

From this definition it follows that it is not always necessary to use only the initial functions in the construction of a particular primitive recursive function. For example, if we already have a set of functions $f_1, f_2, \ldots, f_k$ which are primitive recursive, then we could use any of these functions along with the initial functions to obtain another primitive recursive function, provided we restrict ourselves to the operations of composition and recursion only. In the examples given here, first we construct some primitive recursive functions by using the initial functions alone, and then we gradually use these functions wherever required in order to construct other primitive recursive functions. Of course, all primitive recursive functions are total.

EXAMPLE 1 Show that the function $f\langle x, y\rangle = x + y$ is primitive recursive.

SOLUTION Notice that $x + (y+1) = (x+y) + 1$, so that

$$f\langle x, y+1\rangle = f\langle x, y\rangle + 1 = S(f\langle x, y\rangle)$$

also

$$f\langle x, 0\rangle = x$$

We can now formally define $f\langle x, y\rangle$ as

$$f\langle x, 0\rangle = x = U_1^1\langle x\rangle$$
$$f\langle x, y+1\rangle = S(U_3^3\langle x, y, f\langle x, y\rangle\rangle)$$

Here the base function is $g(x) = U_1^1(x)$, and the inductive-step function is $h\langle x, y, z\rangle = S(U_3^3\langle x, y, z\rangle)$. In order to see how we can use the above definition to actually compute the value of $f\langle 2, 4\rangle$, for example, we have

$$\begin{aligned} f\langle 2,0\rangle &= 2 \\ f\langle 2,4\rangle &= S(f\langle 2,3\rangle) = S(S(f\langle 2,2\rangle)) \\ &= S(S(S(f\langle 2,1\rangle))) = S(S(S(S(f\langle 2,0\rangle)))) \\ &= S(S(S(S(2)))) = S(S(S(3))) = S(S(4)) \\ &= S(5) = 6 \end{aligned}$$

////

EXAMPLE 2 Using recursion, define the multiplication function $*$ given by

$$g\langle x, y\rangle = x * y$$

SOLUTION Since $g\langle x, 0\rangle = 0$ and $g\langle x, y+1\rangle = g\langle x, y\rangle + x$, we write

$$g\langle x, 0\rangle = Z(x)$$
$$g\langle x, y+1\rangle = f\langle U_3^3\langle x, y, g\langle x, y\rangle\rangle, U_1^3\langle x, y, g\langle x, y\rangle\rangle\rangle$$

where f is the addition function given in Example 1. ////

The following are some of the primitive recursive functions which are used frequently. In some cases we give only an informal definition, although a formal definition using the initial functions and other primitive recursive functions can be given easily.

1 *Sign function or nonzero test function, sg:*

$$sg(0) = 0 \qquad sg(y+1) = 1$$

or

$$sg(0) = Z(0) \qquad sg(y+1) = S(Z(U_2^2\langle y, sg(y)\rangle))$$

2 *Zero test function, $\overline{sg}$*

$$\overline{sg}(0) = 1 \qquad \overline{sg}(y+1) = 0$$

3 *Predecessor function, P:*

$$P(0) = 0 \qquad P(y+1) = y = U_1^2(y, P(y))$$

Note that

$$P(0) = 0 \quad P(1) = 0 \quad P(2) = 1 \quad P(3) = 2 \quad \cdots$$

4 *Odd and even parity function, Pr:*

$$Pr(0) = 0 \qquad Pr(y+1) = \overline{sg}(U_2^2(y, Pr(y)))$$

$$Pr(0) = 0 \quad Pr(1) = 1 \quad Pr(2) = 0 \quad Pr(3) = 1 \quad \cdots$$

Scanned with OKEN Scanner

5 *Proper subtraction function*, $\dot{-}$:

$$x \dot{-} 0 = x \qquad x \dot{-} (y + 1) = P(x \dot{-} y)$$

Note that $x \dot{-} y = 0$ for $x < y$ and $x \dot{-} y = x - y$ for $x \geq y$.

6 *Absolute value function*, $|\ |$:

$$|x - y| = (x \dot{-} y) + (y \dot{-} x)$$

7 $\min \langle x, y \rangle$ = minimum of x and y

$$\min \langle x, y \rangle = x \dot{-} (x \dot{-} y)$$

Similarly, $\max \langle x, y \rangle$ = maximum of x and y

$$\max \langle x, y \rangle = y + (x \dot{-} y)$$

8 *The square function*, $f(y) = y^2$:

$$f(y) = y^2 = U_1^1(y) * U_1^1(y)$$

EXAMPLE 3 Show that $f \langle x, y \rangle = x^y$ is a primitive recursive function.

SOLUTION Note that $x^0 = 1$ for $x \neq 0$, and we put $x^0 = 0$ for $x = 0$. Also $x^{y+1} = x^y * x$; hence $f \langle x, y \rangle = x^y$ is defined as

$$f \langle x, 0 \rangle = sg(x)$$

$$f \langle x, y + 1 \rangle = x * f \langle x, y \rangle = U_1^3 \langle x, y, f \langle x, y \rangle \rangle * U_3^3 \langle x, y, f \langle x, y \rangle \rangle \qquad ////$$

EXAMPLE 4 Show that if $f \langle x, y \rangle$ defines the remainder upon division of y by x, then it is a primitive recursive function.

SOLUTION For $y = 0$, $f \langle x, 0 \rangle = 0$. Also the value of $f \langle x, y \rangle$ increases by 1 when y is increased by 1, until the value becomes equal to x, in which case it is put equal to 0 and the process continues. We therefore build a function which increases by 1 each time y increases by 1, that is, $S(f \langle x, y \rangle)$. Now, we multiply this function by another primitive recursive function which becomes 0 whenever $S(f \langle x, y \rangle) = x$. Also this other function must be 1 whenever $S(f \langle x, y \rangle) \neq x$, but $S(f \langle x, y \rangle)$ is always $\leq x$ and hence such a function is $sg(x \dot{-} S(f \langle x, y \rangle))$. Thus the required definition of $f \langle x, y \rangle$ is

$$f \langle x, 0 \rangle = 0$$

$$f \langle x, y + 1 \rangle = S(f \langle x, y \rangle) * sg(x \dot{-} S(f \langle x, y \rangle)) \qquad ////$$

EXAMPLE 5 Show that the function $[x/2]$ which is equal to the greatest integer which is $\leq x/2$ is primitive recursive.

SOLUTION Now $[0/2] = 0$, $[1/2] = 0$, $[2/2] = 1$, $[3/2] = 1$, etc., so that $[x/2] = x/2$ when x is even and $[x/2] = (x - 1)/2$ when x is odd. In order to distinguish between even and odd functions, we have already defined the parity function which is primitive recursive.

$$[0/2] = 0 \qquad [(y + 1)/2] = [y/2] + Pr(y)$$

where $Pr(y)$ denotes the parity function which is 1 when y is odd and which is 0 when y is even. ////

Recall that a set of ordered pairs defines a binary relation. Similarly, a set of n-tuples defines an n-ary relation. If the n-tuples are defined over the set of natural numbers only, then such an n-ary relation is called *number-theoretic*. Thus any set $R \subseteq \mathbf{N}^n$ defines a number-theoretic n-ary relation. As in the case of functions, in this section we restrict ourselves to only number-theoretic relations. Any such relation can also be described by an n-ary predicate. The characteristic function of a relation R can now be defined as

$$\psi_R \langle x_1, x_2, \ldots, x_n \rangle = \begin{cases} 1 & \text{if } \langle x_1, x_2, \ldots, x_n \rangle \in R \\ 0 & \text{if } \langle x_1, x_2, \ldots, x_n \rangle \notin R \end{cases}$$

Here $R \subseteq \mathbf{N}^n$ and $\langle x_1, x_2, \ldots, x_n \rangle \in \mathbf{N}^n$.

Definition 2-6.2 A relation R is said to be *primitive recursive* if its characteristic function is primitive recursive. The corresponding predicate is also called primitive recursive.

EXAMPLE 6 Show that $\{\langle x, x \rangle \mid x \in \mathbf{N}\}$ which defines the relation of equality is primitive recursive.

SOLUTION Obviously $f \langle x, y \rangle = \overline{sg}(|x - y|)$ defines a primitive recursive function such that $f \langle x, y \rangle = 1$ for $x = y$ and otherwise $f \langle x, y \rangle = 0$. Thus, $f \langle x, y \rangle$ is the required characteristic function which is primitive recursive. ////

EXAMPLE 7 Show that for any fixed k the relation given by $\{\langle k, y \rangle \mid y > k\}$ is primitive recursive.

SOLUTION $sg(y \dot{-} k)$ is the characteristic function of the required relation. ////

EXAMPLE 8 Show that the function $f \langle x_1, x_2, y \rangle$ defined as

$$f \langle x_1, x_2, y \rangle = \begin{cases} x_2 & x_1 > y \\ (x_1 * y) + x_2 & x_1 \leq y \end{cases}$$

is primitive recursive.

SOLUTION The required function can be expressed as

$$x_2 + (x_1 * y) * \overline{sg}(x_1 \dot{-} y) \qquad ////$$

We shall now introduce an operation which will be used to generate a larger class of functions that includes the class of primitive recursive functions.

Definition 2-6.3 Let $g \langle x_1, x_2, \ldots, x_n, y \rangle$ be a total function. If there exists at least one value of y, say $\bar{y} \in \mathbf{N}$, such that the function $g \langle x_1, x_2, \ldots, x_n, \bar{y} \rangle = 0$ for all n-tuples $\langle x_1, x_2, \ldots, x_n \rangle \in \mathbf{N}^n$, then g is called a *regular* function.

Not all total functions are regular, as can be seen from $g \langle x, y \rangle = |y^2 - x|$. Obviously $g \langle x, y \rangle$ is total, but $|y^2 - x| = 0$ for only those values of x which are

Scanned with OKEN Scanner

perfect squares and not for all values of x. This fact shows that there is no value of $y \in \mathbf{N}$ such that $|y^2 - x| = 0$ for all x. On the other hand, the function $y \dot{-} x$ is regular because for $y = 0$, $y \dot{-} x$ is zero for all x.

Definition 2-6.4 A function $f(x_1, x_2, \ldots, x_n)$ is said to be defined from a total function $g(x_1, x_2, \ldots, x_n, y)$ by *minimization* (*minimalization*) or *μ operation* if

$$f(x_1, x_2, \ldots, x_n) = \begin{cases} \mu_y(g(x_1, x_2, \ldots, x_n, y) = 0) & \text{if there is such a } y \\ \text{undefined} & \text{otherwise} \end{cases}$$

where μ_y means the least y greater than or equal to zero.

From the definition it follows that $f(x_1, x_2, \ldots, x_n)$ is well-defined and total if g is regular. If g is not regular, then the operation of minimization may produce a partial function.

Definition 2-6.5 A function is said to be *recursive* iff it can be obtained from the initial functions by a finite number of applications of the operations of composition, recursion, and minimization over regular functions.

It is clear from the definition that the set of recursive functions properly includes the set of primitive recursive functions. Also the set of recursive functions is closed under the operations of composition, recursion, and minimization over regular functions. If we remove the restriction for the operation of minimization, so that it can be performed over any total function and not necessarily over just regular ones, we get a still larger class of functions defined as partial recursive functions.

Definition 2-6.6 A function is said to be *partial recursive* iff it can be obtained from the initial functions by a finite number of applications of the operations of composition, recursion, and minimization.

As was done in the case of primitive recursive functions, it is not necessary to start always from the initial functions in order to construct other functions in the class. In fact, if a set of recursive functions is known, they can be used along with the admissible operations to generate other recursive functions.

EXAMPLE 9 Show that the function $f(x) = x/2$ is a partial recursive function.

SOLUTION Let $g(x, y) = |2y - x|$. The function g is not regular because $|2y - x| = 0$ only for even values of x. Define

$$f(x) = \mu_y(|2y - x| = 0)$$

Then $f(x) = x/2$ for x even. ////

EXAMPLE 10 Let $\lfloor \sqrt{x} \rfloor$ be the greatest integer $\leq \sqrt{x}$. Show that $\lfloor \sqrt{x} \rfloor$ is primitive recursive.

SOLUTION Observe that $(y + 1)^2 \dot{-} x$ is zero for $(y + 1)^2 \leq x$ and nonzero for $(y + 1)^2 > x$. Therefore, $\overline{sg}((y + 1)^2 \dot{-} x)$ is 1 if $(y + 1)^2 \leq x$ and cannot be equal to 0. The smallest value of y for which $(y + 1)^2 > x$ is the required number $\lfloor \sqrt{x} \rfloor$; hence

$$\lfloor \sqrt{x} \rfloor = \mu_y(\overline{sg}((y + 1)^2 \dot{-} x) = 0$$

Note that $\lfloor \sqrt{x} \rfloor$ is defined for all x and hence is a recursive function. ////

Any n-ary relation $R \subseteq \mathbf{N}^n$ was defined to be primitive recursive if its characteristic function ψ_R of R is primitive recursive. In a similar manner, we can extend this definition to recursive and partial recursive relations. In general, any set A is called recursive (partial recursive) if its characteristic function ψ_A is recursive (partial recursive). It is assumed that we are considering only sets whose elements are natural numbers or sets of n-tuples over the natural numbers or those sets whose elements can be mapped into $\mathbf{N}$.

In Sec. 2-4.5 it was shown that for any set A whose characteristic function is ψ_A, the characteristic function of $\sim A$ is given by $1 - \psi_A$. Similarly, for any two sets A and B with characteristic functions ψ_A and ψ_B, the characteristic functions of $A \cup B$ and $A \cap B$ can also be obtained easily. These ideas can now be extended to recursive sets. If a set A is recursive, that is, ψ_A is recursive, then $\sim A$ is also recursive because

$$\psi_{\sim A} = 1 \dot{-} \psi_A = \overline{sg}(\psi_A)$$

Also, if the sets A and B are recursive, then $A \cap B$ and $A \cup B$ are recursive because

$$\psi_{A \cap B} = \psi_A * \psi_B$$
$$\psi_{A \cup B} = (\psi_A + \psi_B) \dot{-} \psi_{A \cap B}$$

These results show that the class of recursive sets (relations) is closed under the set operations $\sim$, $\cap$, and $\cup$, and consequently it is closed under any other set operation.

EXAMPLE 11 Show that the sets of even and odd natural numbers are both recursive.

SOLUTION Recall that the parity function is the required characteristic function for the set E of even natural numbers. Hence E is primitive recursive. Also, the set of odd natural numbers is $\sim E$; hence $\sim E$ is also primitive recursive. ////

EXAMPLE 12 Show that the set of divisors of a positive integer n is recursive.

SOLUTION A number $x \leq n$ is a divisor of n iff $|x * i - n|$ is equal to zero for one fixed value of i, $1 \leq i \leq n$. This means that $|x * i - n|$ is nonzero for all $1 \leq i \leq n$, if x is not a divisor. Therefore, the characteristic function of the required set is

$$\psi_B(x) = \sum_{i=1}^{n} \overline{sg}\,|x * i - n|$$

where B denotes the set of divisors of n. ////

Scanned with OKEN Scanner

The concept of the extension of a predicate was introduced in Sec. 2-1.7. A predicate whose extension is a set of integers is said to be a *number-theoretic* predicate. All the predicates in this section are assumed to be number-theoretic. Such a predicate is primitive recursive (recursive) iff its extension is primitive recursive (recursive). The characteristic function of a predicate is the characteristic function of its extension. If A is the extension of predicate P and ψ_P denotes the characteristic function of the predicate P, then

$$\psi_P = \psi_A$$

For example, the predicates "is even" and "is a divisor of n" are recursive because their extensions are recursive sets.

If A and B are the extensions of predicates P and Q respectively, then, by definition,

$$\psi_{P \neg Q} = \psi_{A \cup B} \qquad \psi_{P \wedge Q} = \psi_{A \cap B} \qquad \psi_{\neg P} = \psi_{\sim A}$$

It directly follows that if P and Q are recursive, then so are predicates $P \vee Q$, $P \wedge Q$, and $\neg P$.

EXAMPLE 13 Let $D(x)$ denote "number of divisors of x." Show that $D(x)$ is primitive recursive.

SOLUTION It was shown in Example 4 that the function which defines the remainder upon division of y by x is primitive recursive. We shall denote such a function by $rm\ \langle x, y \rangle$. If a number x divides y, then the remainder is 0 and $\overline{sg}(rm\ \langle x, y \rangle) = 1$. Therefore, the number of divisors of y is given by

$$D(y) = \sum_{x=1}^{y} \overline{sg}(rm\ \langle x, y \rangle)$$

This shows that $D(y)$ is primitive recursive. ////

EXAMPLE 14 Show that the predicate "x is prime" is primitive recursive.

SOLUTION A number x is a prime iff it has only two divisors 1 and x, also if it is not 1 or 0. Therefore, the characteristic function of the extension of "x is not a prime" is

$$\psi_{\sim P_r}(x) = sg(D(x) \dot{-} 2) + \overline{sg}(|\, x - 1 \,|) + \overline{sg}(|\, x - 0 \,|)$$

Hence $\psi_{P_r}(x)$ is also primitive recursive and is given by $1 \dot{-} \psi_{\sim P_r}(x)$. ////

In our discussion thus far we considered only one induction variable in the definition of recursion. It is possible to consider two or more induction variables. Note that in the definition of $f\ \langle x_1, x_2, \ldots, x_n, y \rangle$ using recursion, $x_1, x_2, \ldots, x_n$ were treated as parameters and only y was treated as the induction variable. Now we define a function in which we have two induction variables and no parameters. This function will be used in the next section and is known as Ackermann's function. The function $A\ \langle x, y \rangle$ is defined by

$$A\ \langle 0, y \rangle = y + 1$$
$$A\ \langle x + 1, 0 \rangle = A\ \langle x, 1 \rangle$$
$$A\ \langle x + 1, y + 1 \rangle = A\ \langle x, A\ \langle x + 1, y \rangle \rangle$$

Observe that one can construct the value of $A\ \langle x, y \rangle$ for fixed values of x and y by using the above definition. Therefore, $A\ \langle x, y \rangle$ is well defined and total. It is known that $A\ \langle x, y \rangle$ is not primitive recursive, but recursive. We now demonstrate how the above definitions can be used in finding the value of $A\ \langle 2, 2 \rangle$.

$$\begin{aligned} A\ \langle 2, 2 \rangle &= A\ \langle 1, A\ \langle 2, 1 \rangle \rangle \\ A\ \langle 2, 1 \rangle &= A\ \langle 1, A\ \langle 2, 0 \rangle \rangle \\ &= A\ \langle 1, A\ \langle 1, 1 \rangle \rangle \\ A\ \langle 1, 1 \rangle &= A\ \langle 0, A\ \langle 1, 0 \rangle \rangle \\ &= A\ \langle 0, A\ \langle 0, 1 \rangle \rangle \\ &= A\ \langle 0, 2 \rangle \\ &= 3 \\ A\ \langle 2, 1 \rangle &= A\ \langle 1, 3 \rangle \\ &= A\ \langle 0, A\ \langle 1, 2 \rangle \rangle \\ A\ \langle 1, 2 \rangle &= A\ \langle 0, A\ \langle 1, 1 \rangle \rangle \\ &= A\ \langle 0, 3 \rangle \\ &= 4 \\ A\ \langle 2, 1 \rangle &= A\ \langle 0, 4 \rangle \\ &= 5 \\ A\ \langle 2, 2 \rangle &= A\ \langle 1, 5 \rangle \\ &= A\ \langle 0, A\ \langle 1, 4 \rangle \rangle \\ A\ \langle 1, 4 \rangle &= A\ \langle 0, A\ \langle 1, 3 \rangle \rangle \\ A\ \langle 1, 3 \rangle &= A\ \langle 0, A\ \langle 1, 2 \rangle \rangle \\ &= A\ \langle 0, 4 \rangle \\ &= 5 \\ A\ \langle 1, 4 \rangle &= A\ \langle 0, 5 \rangle \\ &= 6 \\ A\ \langle 2, 2 \rangle &= A\ \langle 0, 6 \rangle \\ &= 7 \end{aligned}$$

It was shown earlier that a set is recursive iff the characteristic function is recursive. A characteristic function of a set determines whether a particular element x is or is not a member of the set. This determining process is called a *decision problem* associated with the set. It is desirable that such a decision problem be solved mechanically. This solution can be accomplished if the characteristic function of the set is recursive.

Therefore the decision problem for a set of integers can be reformulated as whether the characteristic function of a set, or equivalently its defining predicate, is recursive. If the answer is yes, then the decision problem is said to be *recursively solvable*; otherwise it is recursively unsolvable. By "solvable" we mean

Scanned with OKEN Scanner

mechanically solvable (in the sense of induction). There are problems which are solvable, but this fact cannot be shown by purely mechanical means.

Suppose that we are interested in generating or enumerating elements of a set whose members are integers. We can consider the enumeration process as the generation of elements by a function. A set is said to be *recursively enumerable* if it is the range of a recursive function. Given an element z which is in the set, in a finite number of computations of the recursive function z will be generated. If we start with successive argument values from zero for the function, then at some point an argument x will be reached for which the functional value is z. If, however, z does not belong to the set, then z's not being generated in a finite number of steps is no guarantee that z does not belong to the set.

We have introduced the notion of a *semidecision* problem associated with a set. If an element belongs to a set, then we can say yes, but if it does not, we cannot say no. The following examples distinguish a fully decidable procedure from a semidecidable one.

Algorithm prime Given an integer i greater than 1, this algorithm will determine whether the integer is a prime number. Remember that the only divisors of a prime number are 1 and the number itself.

1 [Initialize] Set $j \leftarrow 2$.
2 [Finished?] If $j \geq i$ then output "i is prime" and Exit.
3 [Does j divide i?] If $j \mid i$ then output "i is not prime" and Exit.
4 [Increment counter] Set $j \leftarrow j + 1$, go to step 2. ////

For any value of i the algorithm will answer yes or no. The algorithm could have been made more efficient by noting that if a number is not a prime, then a divisor must be less than or equal to the square root of the number.

Algorithm perfect This algorithm decides whether there exists a perfect number greater than some integer i. Consider the set of all divisors of a number except the number itself. A perfect number is one whose sum of all such divisors equals the number. The number 6 is a perfect number since $1 + 2 + 3 = 6$.

1 [Initialize k] Set $k \leftarrow i$.
2 [Increment k] Set $k \leftarrow k + 1$.
3 [Initialize SUM] Set $SUM \leftarrow 0$.
4 [Initialize j] Set $j \leftarrow 1$.
5 [Finished with number?] If $j < k$ then go to step 7.
6 [Is number perfect?] If SUM $= k$ then output k and Exit; otherwise go to step 2.
7 [Does j divide k?] If $j \mid k$ then set $SUM \leftarrow SUM + j$. Set $j \leftarrow j + 1$ and go to step 5.

The algorithm is an example of a semidecidable process. It can say yes, but can never say no, because the question of whether there are an infinite number of perfect numbers is not known. If there were a finite number of perfect numbers, then there would be a maximum perfect number and the algorithm could, in this case, answer yes or no to the question. ////

In certain areas of mathematics we are interested in the existence of algorithms for solving problems which are not concerned with integers. The concept of solvability can be extended to domains other than the natural number system. We can set up a one-to-one correspondence between elements in the nonarithmetic system and the natural numbers. This type of numbering was discussed in Sec. 2-5.2. In so doing, statements about elements in the nonnumeric system can be transformed into corresponding statements concerning integers. The same notion applies to predicates.

EXERCISES 2-6.1

1 Show that the function

$$f(x) = \begin{cases} x/2 & \text{when } x \text{ is even} \\ (x-1)/2 & \text{when } x \text{ is odd} \end{cases}$$

is primitive recursive.
2 Show that the function $f\langle x, y\rangle = x - y$ is partial recursive.
3 Show that the function $x!$ (the factorial function) is primitive recursive, where $0! = 1$ and $n! = n * (n-1)!$.
4 Show that the quotient function $g\langle x, y\rangle$= quotient upon division of y by x is primitive recursive.
5 Show that the function $f(x) = k$, where k is a constant, is primitive recursive.
6 If $\Pi(x)$ = number of primes $\leq x$, show that $\Pi(x)$ is primitive recursive.
7 Show that every finite set is primitive recursive. (*Hint*: Use the fact that the union of primitive recursive sets is primitive recursive.)
8 Find a function $f(x)$ such that $f(2) = 3$, $f(4) = 5$, $f(7) = 2$, and $f(x)$ assumes any arbitrary value for other arguments. Show that $f(x)$ is primitive recursive.

2-6.2 Recursion in Programming Languages

In the previous section we defined the set of recursive functions. The basic idea is to define a function for all its argument values in a constructive manner by using induction. The value of a function for a particular argument value can be computed in a finite number of steps by using the recursive definition where at each step of recursion we get nearer to the solution. It should be noted that a recursive definition may not necessarily define a function. This is the reason why we imposed certain restrictions and obtained primitive recursive, recursive, and finally partial recursive functions.

An important facility available to the programmer is the procedure (function or subroutine). Procedures in programming languages are a convenience to the programmer since they enable the programmer to express just once an algorithm which is required in many places in a program. Corresponding to a recursive step in the definition of a function, in certain programming languages such as ALGOL, PL/I, and SNOBOL4 (but not FORTRAN) we have a procedure which may contain a procedure call to any procedure including itself. A procedure that contains a procedure call to itself, or a procedure call to a second procedure which eventually causes the first procedure to be called, is known as a *recursive procedure*.

Scanned with OKEN Scanner

There are two important conditions that must be satisfied by any recursive procedure. First, each time a procedure calls itself (either directly or indirectly) it must be "nearer," in some sense, to a solution. In the case of the factorial function (see Prob. 3 in Exercises 2-6.1), each time that the function calls itself, its argument is decremented by 1, and so the argument of the function is getting smaller. Second, there must be a decision criterion for stopping the process or computation. In the case of the factorial function, the value of n must be zero. The term "nearer" can therefore be defined in terms of the stopping condition.

As was discussed in the previous subsection, there are essentially two types of recursion. The first type is the class of primitive recursive functions, and an example of this kind is the factorial function. The second type of recursion is the class of functions which can be obtained by minimization.

Many people believe that recursion is an unnecessary luxury in a programming language. This view is based on the fact that any primitive recursive function can be solved iteratively.

An "iterative" process can be explained by the flowchart given in Fig. 2-6.1. There are four parts in the process, namely, initialization, decision, computation, and update. The functions of the four parts are as follows:

1 *Initialization.* The parameters of the function and a decision parameter in this part are set to their initial values. The decision parameter is used to determine when to exit from the loop.

2 *Computation.* The required computation is performed in this part.

3 *Decision.* The decision parameter is to determine whether to remain in the loop.

4 *Update.* The decision parameter is updated, and a transfer to the next iteration results.

It is possible to transform mechanically any primitive recursive function into an equivalent iterative process. However, such is *not* the case for nonprimitive recursive functions. Although there does exist an iterative solution for Ackermann's function, in general there are many problems of that form for which iterative solutions either do not exist or are not easily found. Certain inherently recursive processes can be solved in programming languages which do not permit recursion only by essentially setting up a recursive framework. We will have occasion to return to this topic later. Recursion is becoming increasingly important in symbol manipulation and nonnumeric applications.

Throughout the text we encounter problems where recursion is *unavoidable* because of the recursive nature of the process or because of the recursive structure of the data which have to be processed. Even for cases where there is no inherent recursive structure, the recursive solution may be much simpler (though sometimes more time-consuming) than its iterative counterpart.

The drawbacks of recursion are that usually the execution of a recursive program is slower than that of its iterative counterparts, and a recursive program is more difficult to debug than a corresponding iterative one.

There are special problems associated with a recursive procedure that do not exist for a nonrecursive procedure. A recursive procedure can be called from

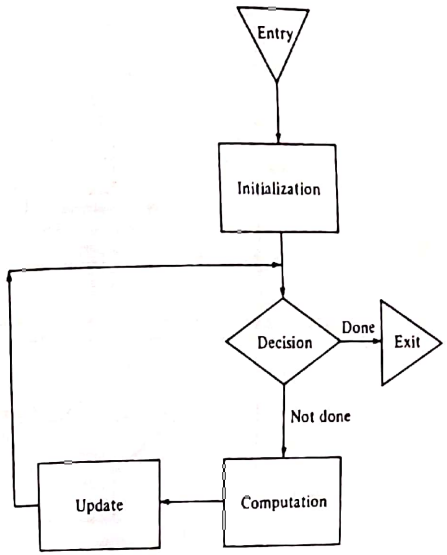


FIGURE 2-6.1 Model of iteration.

within or outside itself, and to ensure its proper functioning, it has to save, in some order, the return addresses so that a return to the proper location will result when return to a calling statement is made. The procedure must also save, in some order, the formal parameters, local variables, etc., upon entry and must restore these parameters and variables at completion.

The saving and restoring of return addresses, parameters, and local variables are incorporated in the flowchart model for a recursive procedure given in Fig. 2-6.2. The model consists of a prologue, a body, and an epilogue. The purpose of the prologue is to save the formal parameters, local variables, and return address, while that of the epilogue is to restore them. Note that the parameters, local variables, and return address which are restored are those which were most recently saved; i.e., the last saved are the first to be restored (Last In, First Out). The body of the procedure contains a procedure call to itself; in fact, there may be more than one call to itself in certain procedures.

It is rather difficult to understand a recursive procedure from its flowchart, and the best one can hope for is to acquire an intuitive understanding of the procedure. The key box contained in the body of the procedure is the one which invokes a call to itself. The dotted-line exit from this box indicates that a call to itself is being initiated within the same procedure. Each time a procedure call to itself is executed, the prologue of the procedure saves all necessary information required for proper functioning.

The procedure body contains two computation boxes, namely, the partial and final computation boxes. Frequently the partial computation box is combined with the procedure call box (this is the case for the computation of the factorial

Scanned with OKEN Scanner

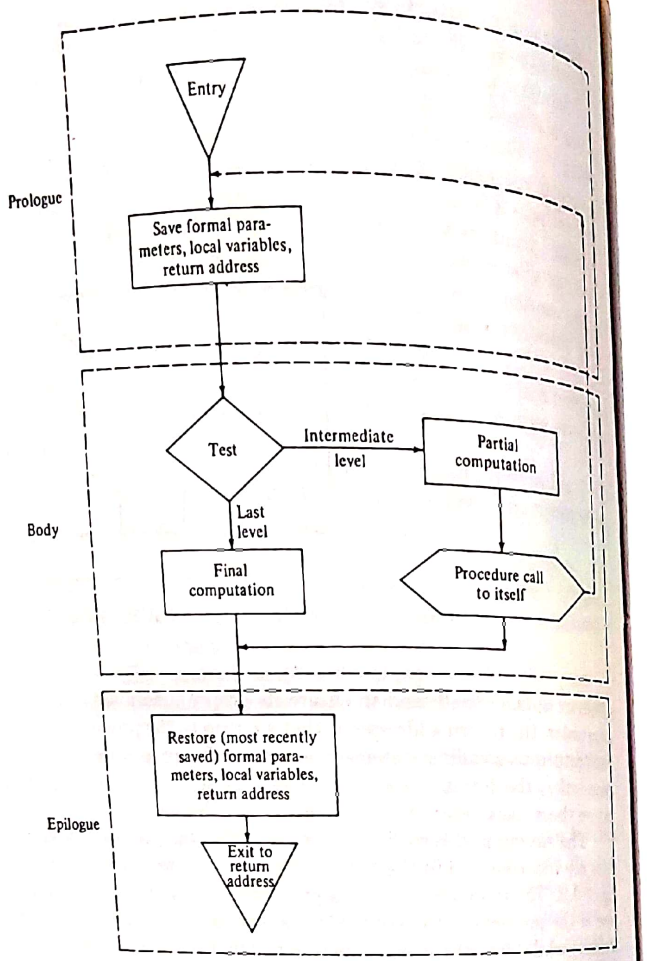


FIGURE 2-6.2 Flowchart for model of a recursive procedure.

function). The final computation box gives the explicit definition of the process for some value or values of the argument(s). The test box determines whether the argument value(s) are those for which an explicit definition of the process is given.

A flowchart for a procedure which computes the factorial function recursively is given in Fig. 2-6.3. The testing box checks n for a value of zero. If the test is successful, the argument (zero) for which the function is explicitly defined ($0! = 1$) has been found. The final computation box specifies the value of the factorial function for the argument $n = 0$. The partial computation box and the call box are combined into one. The prologue of the procedure saves the value of n and the return address on entering the procedure and the epilogue restores the most recently stored value of n and the return address (on a Last In First Out basis).

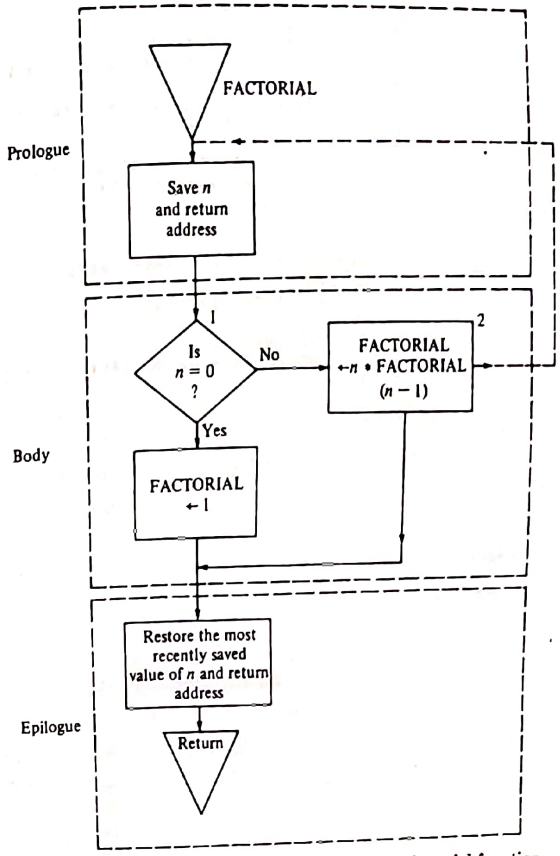


FIGURE 2-6.3 Flowchart for the factorial function.

A recursive PL/I formulation of the factorial function together with a main procedure to test the function is given in Fig. 2-6.4. Note that all variables used in the program are given the characteristics binary and fixed with precision of (31, 0). This assignment was necessary for proper execution of the procedure. Since the identifier FACTORIAL defaults to float decimal, the RETURNS attribute must be used both in a declaration of the function in the invoking procedure and in the PROCEDURE statement. This procedure, as well as all procedures which reactivate themselves, must be declared to have the RECURSIVE attribute.

Assume that this PL/I procedure is initially called from a main program with N equal to 2. We will associate a *level number* with each entry into the procedure FACTORIAL and in particular associate *level* 1 when the function is entered due to the initial call from the main program, which is assumed to have level 0. The following is a trace of the execution of the FACTORIAL procedure.

Scanned with OKEN Scanner

```
RUNFACT:
    PROCEDURE OPTIONS(MAIN);
    /* TEST THE RECURSIVE FACTORIAL FUNCTION */
    DECLARE
        FACTORIAL RETURNS(BINARY FIXED(31)),
        (I,K) BINARY FIXED(31);
    DO I = 3 TO 7 BY 2;
        PUT SKIP(2) EDIT('FACTORIAL(',I,') IS ',FACTORIAL(I))
            (A(10),F(1),A(5),F(5));
    END;

FACTORIAL:
        PROCEDURE (N) RECURSIVE RETURNS(BINARY FIXED(31));
        DECLARE
            N BINARY FIXED(31);
        IF N = 0
        THEN RETURN(1);
        ELSE RETURN(N * FACTORIAL(N - 1));
END FACTORIAL;

END RUNFACT;
```

```
FACTORIAL(3) IS     6
FACTORIAL(5) IS   120
FACTORIAL(7) IS  5040
```

FIGURE 2-6.4 PL/I program for the factorial function.

Enter level 1	called from the main program (level 0)	Save N (initially garbage as far as we are concerned) N ← 2
		N ≠ 0 FACTORIAL(1) is called.
Enter level 2	first recursive call from N * FACTORIAL (N − 1) with (N = 2)	Save N, which has a value of 2. N ← 1
		N ≠ 0 FACTORIAL⟨0⟩ is called.
Enter level 3	second recursive call from N * FACTORIAL(N − 1) (with N = 1)	Save N, which has a value of 1. N ← 0
		N = 0 Return 1.
Return to level 2		Restore N ← 1. The value 1 is passed to the expression in level 2, which called the procedure. The expression N * FACTORIAL(N − 1) is now 1 * FACTORIAL(0) = 1.

Return to level 1	Restore N ← 2. The value 1 computed above is passed to the invoking expression in level 1. The expression N * FACTORIAL(N − 1) is now 2 * FACTORIAL(1) = 2.
Return to main program (level 0)	Restore N to garbage. Return the value 2 to invoking statement. The call in the main program has the value 2.

Let us now consider a more complex recursion example. A well-known algorithm for finding the greatest common divisor of two integers is Euclid's algorithm. The greatest common divisor function is denoted by the following:

$$\text{GCD}\ \langle m, n\rangle = \begin{cases} \text{GCD}\ \langle n, m\rangle & \text{if } n > m \\ m & \text{if } n = 0 \\ \text{GCD}\ \langle n, \text{MOD}\ \langle m, n\rangle\rangle & \text{otherwise} \end{cases}$$

Here MOD $\langle m, n\rangle$ is $m \pmod n$, the remainder on dividing m by n. The first part of the definition interchanges the order of the arguments if $n > m$ (the first argument must be greater than or equal to the second for the algorithm to work). If the second argument is zero, then the greatest common divisor is equal to the first argument (this defines the base values of the function). Finally, GCD is defined in terms of itself. Note that the process must terminate since MOD$\langle m, n\rangle$ will decrease to a value of zero in a finite number of steps. As an example, GCD ⟨20 6⟩ is obtained from the following computation:

$$20 = 6 * 3 + 2$$

By the euclidean algorithm, GCD⟨20, 6⟩ is the same as GCD⟨6, 2⟩. Therefore

$$6 = 2 * 3 + 0$$

and GCD ⟨6, 2⟩ is the same as GCD ⟨2, 0⟩, which is 2. If, instead, we were required to find GCD ⟨6, 20⟩, this problem could be solved by finding a solution to GCD ⟨20, 6⟩ instead.

The PL/I program for the GCD function is given in Fig. 2-6.5. The same comments given for the FACTORIAL procedure apply to the GCD procedure. Note also the use of the ON ENDFILE(SYSIN) statement which specifies the action required on exhaustion of the input card file. PL/I also has a MOD function which is a convenience in programming the GCD procedure.

In Sec. 2-6.1, Ackermann's function was briefly discussed. This classical function will be extensively studied in the remainder of this section. First, a recursive PL/I formulation is given, followed by an iterative solution in algorithmic notation and also in PL/I. Finally, this function will be programmed by faking recursion in FORTRAN.

A program to compute Ackermann's function recursively is simple to write

Scanned with OKEN Scanner

```
RUN_GCD:
    PROCEDURE OPTIONS(MAIN);
    /* TEST THE EUCLIDEAN ALGORITHM */
    DECLARE
        (I,J) BINARY FIXED(31),
        GCD RETURNS(BINARY FIXED(31));
    ON ENDFILE(SYSIN) GO TO END;
READ:  /* GET SOME VALUES TO TEST THE GCD FUNCTION */
    GET SKIP LIST(I,J);
    PUT SKIP(2) EDIT('THE GREATEST COMMON DIVISOR OF ',I,' AND ',
        ' IS ',GCD(I,J))(A(31),F(5),A(5),F(5),A(4),F(5));
    GO TO READ;

GCD:
        PROCEDURE (M,N) RECURSIVE RETURNS(BINARY FIXED(31));
        DECLARE
            (M,N) BINARY FIXED(31);
        IF N > M /* REVERSE THE CALL */
        THEN RETURN(GCD(N,M));
        IF N = 0 /* M IS THE GREATEST COMMON DIVISOR */
        THEN RETURN(M);
        RETURN(GCD(N,MOD(M,N))); /* EQUIVALENT VALUE */
END GCD;

END: END RUN_GCD;
```

```
THE GREATEST COMMON DIVISOR OF    84 AND   246 IS     6
THE GREATEST COMMON DIVISOR OF     6 AND    20 IS     2
THE GREATEST COMMON DIVISOR OF   121 AND    33 IS    11
```

FIGURE 2-6.5 PL/I program for the GCD function.

and is given in Fig. 2-6.6. It is essentially a one-one implementation of the clauses in the definition of the function embodied in a procedure with the RECURSIVE attribute.

Ackermann's function is theoretically significant in that it is a recursive function that is not primitive recursive. Ackermann provided the proof of the non-primitive recursive nature of this function when he established that it grows faster than any primitive recursive function. As an illustration of the growth rate, consider that if we define $g(x) = A\ \langle x, x\rangle$, then $g(x)$ for $x = 0, 1, 2, 3, 4, \ldots$ is $1, 3, 7, 61, 2^{2^{65,536}} - 3$.

To explain the iterative evaluation program for Ackermann's function, it is necessary first to redefine the function in a form which is more easily investigated. We can rewrite the original definition as

1 $A\ \langle 0, n\rangle = n + 1$
2 $A\ \langle m + 1, 0\rangle = A\ \langle m, 1\rangle$
3 $A\ \langle m + 1, n + 1\rangle = A\ \langle m, A\ \langle m + 1, n\rangle\rangle$

We may think of this function as defining an entry in a table for each $\langle m, n\rangle$ argument pair. Let m designate the row number of the table and n the column number in which $A\ \langle m, n\rangle$ may be found. Row 0 is defined for each entry as the column number of that entry plus 1 (by clause (1)). Each entry of column 0 is defined (by (2)) as the column 1 entry of the previous row (except for $A\ \langle 0, 0\rangle$


```
RECTEST:
    PROCEDURE OPTIONS(MAIN);
    /* TEST ACKERMANN'S FUNCTION. */
    DECLARE
        A ENTRY(BINARY FIXED(31),BINARY FIXED(31))
          RETURNS(BINARY FIXED(31));
    PUT EDIT('THE SOLUTION FOR ACKERMANN''S FUNCTION, A(2,2) IS ',
        A(2,2))(A(49),F(5));


A:      PROCEDURE (M,N) RECURSIVE RETURNS (BINARY FIXED(31));
        /* ACKERMANN'S FUNCTION */
        DECLARE
            (M,N) BINARY FIXED(31);
        IF M = 0
        THEN RETURN(N + 1);
        IF N = 0
        THEN RETURN(A(M - 1,1));
        RETURN(A(M - 1,A(M,N - 1)));

END A;

END RECTEST;
```

```
THE SOLUTION FOR ACKERMANN'S FUNCTION, A(2,2) IS      7
```

FIGURE 2-6.6 Recursive formulation of Ackermann's function.

which is defined by (1)). Part (3) states that the entry in row $m + 1$ and column $n + 1$ is equal to the entry in the previous row in the column position given by $A\ \langle m + 1, n\rangle$. These relationships are shown by arrows in Fig. 2-6.7.

Figure 2-6.7 gives an indication of the nature of the iterative solution by showing all values which must be generated to calculate $A\ \langle 4, 0\rangle$. Entries for row 0 are added one by one and, if necessary, are propagated to the following rows, as shown by the arrows. Note that the last entry in each row is equal to $A\ \langle 4, 0\rangle$. The iterative solution can be performed without using a two-dimensional array since there is no need to save an entry when another follows it in the same row. Instead two vectors will be used. The ith elements of vectors *VALUE* and *PLACE* will indicate the last number to be generated and its column position, respectively, in row i. If $A\ \langle m, n\rangle$ is to be found, no more than $m + 1$ elements

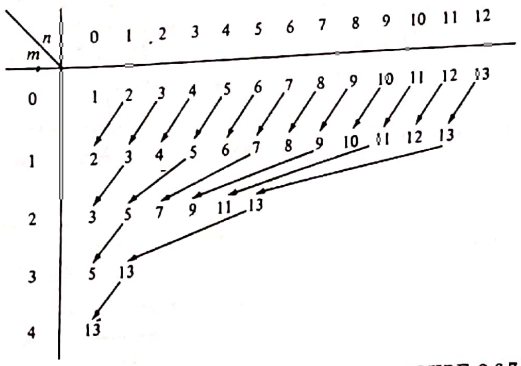


FIGURE 2-6.7

are required in each vector. On completion of the iterative solution to calculate $A\langle 4, 0\rangle$, the vectors will contain the following:

i	$VALUE[i]$	$PLACE[i]$
		12
0	13	11
1	13	5
2	13	1
3	13	0
4	13	

With this introductory knowledge, we are now able to study the algorithm for the iterative solution.

Algorithm *ACKER* Iteratively solve for $A\langle m, n\rangle$, the value of Ackermann's function for valid arguments m and n. The vectors $PLACE$ and $VALUE$ previously described are implemented. *ACKER* is assigned the value of $A\langle m, n\rangle$ before return is made from the algorithm.

1 [Trivial case] If $m = 0$ then set $ACKER \leftarrow n + 1$ and Exit.

2 [Initialize for iteration] Set $VALUE[0] \leftarrow 1$, $PLACE[0] \leftarrow 0$. (This step initializes row 0.)

3 [Iteration loop: get new value] Set $VALUE[0] \leftarrow$ VALUE[0]+1 and $PLACE[0] \leftarrow PLACE[0] + 1$. (Here a new entry is made for row 0.)

4 [Propagate $VALUE[0]$] Repeat steps 5 and 6 for $i = 0, \ldots, m-1$.

5 [Initiate new row] If $PLACE[i] = 1$ then set $VALUE[i+1] \leftarrow VALUE[0]$ and $PLACE[i+1] \leftarrow 0$; otherwise go to step 6. If $i = m-1$ then go to step 7; otherwise go to step 3. (This step performs the assignment $A\langle i+1, 0\rangle \leftarrow A\langle i, 1\rangle$. If $i+1$ is equal to m, the algorithm may be complete. Otherwise we go to step 3 since no new rows may be started.)

6 [Move $VALUE[0]$ to another row] If $VALUE[i+1] = PLACE[i]$ then set $VALUE[i+1] \leftarrow VALUE[0]$ and $PLACE[i+1] \leftarrow PLACE[i+1]+1$; otherwise go to step 3. (If $VALUE[0]$ cannot be propagated to another row, leave the loop; otherwise the value is transferred according to $A\langle i+1, j+1\rangle \leftarrow A\langle i, A\langle i+1, j\rangle\rangle$ where j equals $PLACE[i+1]$ before it is incremented. In this case, if $i+1 < m$, we must remain in the loop.)

7 [Check for end of iteration] If $PLACE[m] = n$ then set $ACKER \leftarrow VALUE[0]$ and Exit; otherwise go to step 3.

It is important to note the correspondence between the steps of the algorithm and the three clauses used to redefine Ackermann's function:

Clause	*Algorithm steps(s)*
(1) $A\langle 0, n\rangle = n + 1$	2 and 3
(2) $A\langle m+1, 0\rangle = A\langle m, 1\rangle$	5
(3) $A\langle m+1, n+1\rangle = A\langle m, A\langle m+1, n\rangle\rangle$	6

////

A PL/I program which tests this algorithm is given in Fig. 2-6.8.


```
ITERATE:
   PROCEDURE OPTIONS(MAIN);
   /* TEST THE ITERATIVE SOLUTION TO ACKERMANN'S FUNCTION */
   DECLARE
      ACKER ENTRY(BIN FIXED(31),BIN FIXED(31)) RETURNS(BIN FIXED(31));

   PUT SKIP(3) EDIT('THE SOLUTION TO ACKERMANN''S FUNCTION FOR M=2 A',
   'ND N=2 IS ',ACKER(2,2))(A(46),A(10),F(2));
   PUT SKIP(3) EDIT('THE SOLUTION TO ACKERMANN''S FUNCTION FOR M=3 A',
   'ND N=3 IS ',ACKER(3,3))(A(46),A(10),F(2));


ACKER:
      PROCEDURE (M,N) RETURNS(BINARY FIXED(31));
      /* AN ITERATIVE SOLUTION TO ACKERMANN'S FUNCTION DUE TO RICE,
         CACM VOLUME 8, NUMBER 2, (1965).
         COMPUTE ACKERMANN'S FUNCTION DEFINED BY
                  ACKER(0,N) = N + 1
                  ACKER(M + 1,0) = ACKER(M,1),
                  ACKER(M + 1,N + 1) = ACKER(M,ACKER(M + 1,N))
         FOR THE ARGUMENTS M AND N.                          */
      DECLARE
         (M,N) BINARY FIXED(31),
         (PLACE,VALUE)(0:M) BINARY FIXED (31);
      IF M = 0 /* TRIVIAL CASE - RETURN N + 1 */
      THEN RETURN(N + 1);
      VALUE(0) = 1; /* INITIALIZE FOR ITERATION */
      PLACE(0) = 0;
LOOP: /* ITERATION LOOP*/
      VALUE(0) = VALUE(0) + 1;  /* EXTEND ROW ZERO */
      PLACE(0) = PLACE(0) + 1;
      DO I = 0 TO M - 1; /* PROPAGATE NEW VALUE TO HIGHER LEVELS */
         IF PLACE(I) = 1
         THEN
            DO; /* INITIATE NEW LEVEL */
               VALUE(I + 1) = VALUE(0);
               PLACE(I + 1) = 0;
               IF I = M - 1 /* END OF PROPAGATION */
               THEN GO TO CHECK_N;
               ELSE GO TO LOOP;
            END;
         IF VALUE(I + 1) = PLACE(I)
         THEN
            DO; /* GET NEXT VALUE FOR LEVEL I + 1 FROM THE COLUMN
                   OF LEVEL I INDICATED BY VALUE(I+1) */
               VALUE(I + 1) = VALUE(0);
               PLACE(I + 1) = PLACE(I + 1) + 1;
            END;
         ELSE GO TO LOOP; /* NO PROPAGATION SO GET NEXT VALUE */
      END; /* OF DO I = 0 TO M - 1 */
CHECK_N: /* CHECK FOR END OF ITERATIVE PROCESS */
      IF PLACE(M) = N /* NTH COLUMN AT LEVEL M HAS BEEN REACHED */
      THEN RETURN(VALUE(0));
      ELSE GO TO LOOP;
END ACKER;

END ITERATE;
```

```
THE SOLUTION TO ACKERMANN'S FUNCTION FOR M=2 AND N=2 IS  7

THE SOLUTION TO ACKERMANN'S FUNCTION FOR M=3 AND N=3 IS 61
```

FIGURE 2-6.8 Iterative formulation of Ackermann's function.

This algorithm could theoretically compute any value of Ackermann's function, but the majority of $\langle m, n\rangle$ pairs will result in function values which are too large for a computer word.

It was pointed out earlier that a FORTRAN subprogram could not contain a call to itself. It is an easy matter to program the recursion mechanism in

Scanned with OKEN Scanner

FORTRAN. This is an interesting exercise which gives an insight and understanding of recursion. Before starting the task of programming Ackermann's function in FORTRAN via the faking of recursion, it is necessary to discuss a data structure known as a *stack*. A stack is similar to a railway system for shunting cars, as shown in Fig. 2-6.9.

In this system, the last railway car to be placed on the stack is the first to leave. Repeatedly using the addition and deletion operations permits the cars to be arranged on the output railway line in some prescribed order. For our purposes, a stack will be represented by a vector consisting of some large number of elements which should be sufficient to handle all possible additions likely to be made to the stack. An approximate representation of such an allocation scheme is given in Fig. 2-6.10.

A pointer *TOP* keeps track of the top element in the stack. Initially, when the stack is empty, *TOP* has a value of zero; when the stack contains a single element, *TOP* has a value of 1; and so on. Each time a new element is inserted in the stack, the pointer is incremented by 1 before the element is placed on the stack. The pointer is decremented by 1 each time a deletion is made from the stack. The rightmost occupied element of the stack represents its top element. The leftmost element of the stack represents the bottom element of the stack.

When a recursive procedure is executed, a stack is used as storage for the items mentioned in the prologue and epilogue of Fig. 2-6.2. More specifically, the formal parameters, local variables, and return address are inserted on the stack in a specific order when the procedure is entered. In the epilogue, the most recently saved items are those which are currently on top of the stack because a stack is worked on a Last In, First Out basis. For this reason, a stack is ideally

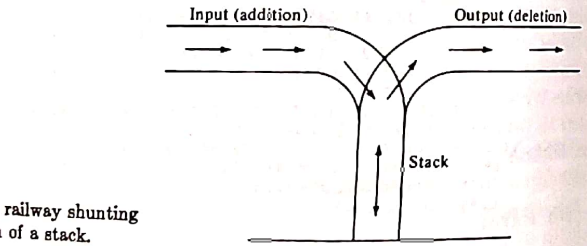


FIGURE 2-6.9 A railway shunting system representation of a stack.

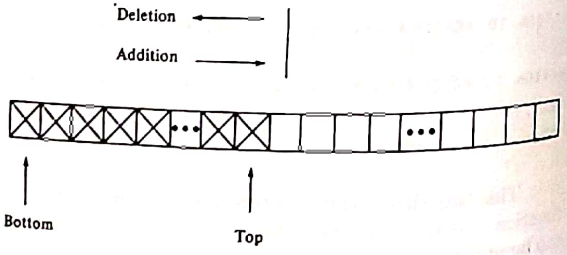


FIGURE 2-6.10 Representation of a stack by a vector.


```
      SUBROUTINE INSERT(S,MAX,TOP,M,N,LABEL)
C     SUBROUTINE TO INSERT THE ARGUMENTS AND RETURN ADDRESS
C     OF ACKERMANN'S FUNCTION ON A STACK.
C     S IS A VECTOR REPRESENTING THE STACK.
C     MAX IS THE NUMBER OF ELEMENTS ALLOCATED TO THE STACK.
C     TOP IS THE POINTER TO THE TOP ELEMENT OF THE STACK.
C     M, N, AND LABEL ARE TO BE PLACED ON THE STACK.
C     ***
      INTEGER S(MAX),TOP
C     ***
C     WILL THE STACK OVERFLOW?
      IF(TOP + 3.GT.MAX) GO TO 1
C     ***
C     STORE NEW ELEMENTS ON STACK.
C     ***
      S(TOP + 1) = M
      S(TOP + 2) = N
      S(TOP + 3) = LABEL
      TOP = TOP + 3
      RETURN
C     ***
C     OUTPUT ERROR MESSAGE.
    1 WRITE(6,100)
      STOP
  100 FORMAT(' ',14HSTACK OVERFLOW)
      END
```

FIGURE 2-6.11 FORTRAN subprogram for stacking M, N, and LABEL.

suited for the handling of recursion. This stack processing, which is performed by the compiler for recursive procedures, must be simulated to permit the faking of recursion.

It should be noted that we have added flexibility in using a vector to represent a stack. We can access elements which are not indicated by *TOP*, a freedom which may not be allowed with other representations of stacks. This flexibility will be used to our advantage in the following FORTRAN subprograms.

For Ackermann's function, only three items, arguments m, n, and the return address, are stacked by the prologue of the procedure. A FORTRAN subroutine which is specifically written to handle the stacking of these items, denoted M, N, and LABEL in the subprogram, is given in Fig. 2-6.11. Stack overflow could occur if the initial choice of arguments M and N is unwise, and so this possibility is investigated in INSERT.

A second subprogram is required by the epilogue for deleting the triplet of items added to the stack by INSERT. This is an obvious procedure, but care must be taken that the proper elements are assigned to the variables M, N, and LABEL. Subroutine DELETE is given in Fig. 2-6.12.

Now that the two necessary stack operations are at our disposal, we must determine how to use them in faking the recursion process. It will be assumed that when execution is taking place at a certain level of recursion, the values of M and N pertaining to that level will be contained in the topmost elements of the stack. The associated return address will indicate the statement to which control is transferred on return to the next lowest level. Thus the arguments required at a certain level must be inserted on the stack immediately on entry, and deleted prior to transfer to a lower level.

The FORTRAN function in Fig. 2-6.13, ACKER, fakes the recursion process of Ackermann's function. The entire recursive framework of Ackermann's

Scanned with OKEN Scanner

```
      SUBROUTINE DELETE(S,MAX,TOP,M,N,LABEL)
C     SUBROUTINE TO DELETE THE ARGUMENTS AND RETURN ADDRESS OF A CALL
C     TO ACKERMANN'S FUNCTION FROM A STACK.
C     S IS THE NAME OF THE STACK.
C     MAX IS THE NUMBER OF ELEMENTS ALLOCATED TO THE STACK.
C     TOP IS THE NUMBER OF THE TOP ELEMENT OF THE STACK.
C     ***
      INTEGER S(MAX),TOP
C     ***
C     REMOVE ARGUMENTS AND RETURN ADDRESS.
C     ***
      LABEL = S(TOP)
      N = S(TOP - 1)
      M = S(TOP - 2)
      TOP = TOP - 3
      RETURN
      END
```

FIGURE 2-6.12 FORTRAN subprogram for deleting elements from a stack.

function is contained within this subprogram. The call to, entry into, and prologue of a level of the function are equivalent to a CALL INSERT statement followed by a transfer to the statement labeled 10. The first call to INSERT places the original values of M and N on the stack along with the number 60 corresponding to LABEL. When these values are unstacked, they will be recognized as corresponding to the initial call, and exit will be made from ACKER.

Statements 8, 9, 21, and 22 are used for tracing the levels of recursion. In statements 8 and 9, the level number from which a call has been made is calculated and printed. Note that the items used in the level being entered are currently on the stack. Thus, if the call was from level 0, there are 3 items on the stack; that is, level 1 is being entered. Similarly, if level $j + 1$ is the highest level entered, there will be $3 * (j + 1)$ items on the stack at that time, but the call to that level was made from level j. Note that the maximum level reached dictates the size of the stack which is necessary. For example, if the maximum level reached is 60, we need 180 elements in the S array. In statements 21 and 22, the level number to which return is being made is printed. If we are returning from level $k + 1$, then the generation of M, N, and return address effective at level $k + 1$ have already been deleted from the stack, and so the number of elements on the stack is $3k$, that is, 3 times the level number being returned to. Thus the level number is easily obtained on dividing TOP by 3.

In statements 6 and 7, the values of M and N to be used at the current level of recursion are retrieved. In statements 11 and 12, if M is zero, ACKER is set to N + 1 and a transfer is made to the statement labeled 50. Such a transfer is equivalent to a return, as will be seen later. Note that the current value of ACKER is always the result of the last completed call. In statements 14 and 15, if N is equal to zero, a recursive call is made by inserting $M - 1$, 1, and return address 50 on the stack. This procedure is in accordance with the definition $A\langle m, n\rangle = A\langle m - 1, 1\rangle$, if $n = 0$, of Ackermann's function. The return address 50 is stacked since, when return is made from this recursive call, the program immediately returns from the level in which the call was made.

If M and N are nonzero, we have a situation where a recursive call must be made to calculate a value for use in another recursive call. That is,

$$A\langle m, n\rangle = A\langle m - 1, A\langle m, n - 1\rangle\rangle$$


```
      INTEGER FUNCTION ACKER(M,N)
C     SUBPROGRAM TO EVALUATE ACKERMANN'S FUNCTION
0001  C BY FAKING RECURSION IN FORTRAN.
C     M AND N ARE ARGUMENTS OF ACKERMANN'S FUNCTION.
C     M AND N ARE REPRESENTING THE STACK.
C     S IS A VECTOR REPRESENTING THE TOP ELEMENT OF THE STACK.
C     TOP IS A POINTER TO THE TOP ELEMENT OF THE STACK.
C     MAX IS THE MAXIMUM NUMBER OF ELEMENTS ALLOCATED TO THE
C     STACK.
C     LABEL REPRESENTS A RETURN ADDRESS.
C     I IS THE LEVEL NUMBER OF A CALL OR RETURN.
      INTEGER S(300),TOP
      MAX = 300
0002  C INITIALIZE THE STACK BY INSERTING THE INITIAL ARGUMENTS AND
0003  C RETURN ADDRESS.  THESE VALUES WILL CAUSE RETURN TO THE
C     INVOKING ROUTINE WHEN UNSTACKED.
      TOP = 0
      CALL INSERT(S,MAX,TOP,M,N,60)
0004  C STATEMENTS 10, 20 AND 30 BEGIN SEGMENTS WHICH HANDLE THE
0005  C THREE CASES OF ACKERMANN'S FUNCTION.
C     ***
C     OBTAIN M AND N VALUES CORRESPONDING TO THIS CALL.
      10 M = S(TOP - 2)
      N = S(TOP - 1)
0006  C CALCULATE AND OUTPUT THE LEVEL NUMBER OF THE CALL.
0007      I = TOP / 3 - 1
      WRITE(6,101)I,M,N
0008  C IF M=0 THEN A(M,N)=N+1 SO TRANSFER TO THE RETURN SEGMENT.
0009      IF(M.NE.0) GO TO 20
      ACKER = N + 1
0010      GO TO 50
0011  C IF N = 0, A(M,N) = A(M - 1,1), SO STACK THE ARGUMENTS AND
0012  C RETURN ADDRESS FOR THE NEXT CALL.
      20 IF(N.NE.0) GO TO 30
0013      CALL INSERT(S,MAX,TOP,M - 1,1,50)
0014      GO TO 10
0015  C STACK THE ARGUMENTS AND RETURN ADDRESS FOR THE INNERMOST
C     RECURSIVE CALL OF A(M,N) = A(M - 1, A(M,N - 1)).
      30 CALL INSERT(S,MAX,TOP,M,N - 1,40)
0016      GO TO 10
0017  C RETURN FROM INNERMOST CALL IN A(M,N) = A(M - 1,A(M,N - 1)).
C     NOW EXECUTE OUTER CALL.
      40 CALL INSERT(S,MAX,TOP,M - 1,ACKER,50)
0018      GO TO 10
0019
C     ***
C     THIS SECTION SIMULATES A RETURN FROM A CALL
C     TO ACKERMANN'S FUNCTION.
      50 CALL DELETE(S,MAX,TOP,M,N,LABEL)
0020  C CALCULATE AND OUTPUT LEVEL NUMBER TO BE RETURNED TO.
0021      I = TOP / 3
0022      WRITE(6,100)I,ACKER
0023      IF(LABEL.EQ.40) GO TO 40
0024      IF(LABEL.EQ.50) GO TO 50
C     RETURN TO INVOKING ROUTINE.
0025      60 RETURN
0026  100 FORMAT(' ',16HRETURN  TO LEVEL,I4,14H OF ACKER  =  ,I2)
0027  101 FORMAT(' ',16HCALL  FROM LEVEL,I4,10H OF ACKER(,I2,1H,
     1,I2,1H))
0028      END
```

FIGURE 2-6.13 FORTRAN function for Ackermann's function.

The innermost call is made by inserting M and N − 1 on the stack with return address 40. The 40 indicates that on return from this call, transfer is made to the statement labeled 40 in which the outermost recursive call is made by stacking $M - 1$ and the previously returned value of ACKER. In this case, 50 is used as a return address, indicating that we return from the original level of invocation on completion of both calls.

Scanned with OKEN Scanner

```
C  PROGRAM TO TEST THE FUNCTION ACKER WHICH FAKES RECURSION
C  TO SOLVE ACKERMANN'S FUNCTION.
C  ***
      INTEGER ACKER,ANSWER
C  ***
C  CALL FROM LEVEL 0 WITH ARGUMENTS M=2 AND N=2.
C  ***
      ANSWER = ACKER(2,2)
      WRITE(6,100)ANSWER
      STOP
  100 FORMAT(' ',13HACKER(2,2) = ,I3)
      END
```

```
CALL  FROM LEVEL  0 OF ACKER( 2, 2)
CALL  FROM LEVEL  1 OF ACKER( 2, 1)
CALL  FROM LEVEL  2 OF ACKER( 2, 0)
CALL  FROM LEVEL  3 OF ACKER( 1, 1)
CALL  FROM LEVEL  4 OF ACKER( 1, 0)
RETURN  TO LEVEL  5 OF ACKER( 0, 1)
RETURN  TO LEVEL  5 OF ACKER =  2
RETURN  TO LEVEL  4 OF ACKER =  2
CALL  FROM LEVEL  4 OF ACKER( 0, 2)
RETURN  TO LEVEL  4 OF ACKER =  3
RETURN  TO LEVEL  3 OF ACKER =  3
RETURN  TO LEVEL  2 OF ACKER =  3
CALL  FROM LEVEL  2 OF ACKER( 1, 3)
CALL  FROM LEVEL  3 OF ACKER( 1, 2)
CALL  FROM LEVEL  4 OF ACKER( 1, 1)
CALL  FROM LEVEL  5 OF ACKER( 1, 0)
CALL  FROM LEVEL  6 OF ACKER( 0, 1)
RETURN  TO LEVEL  6 OF ACKER =  2
RETURN  TO LEVEL  5 OF ACKER =  2
CALL  FROM LEVEL  5 OF ACKER( 0, 2)
RETURN  TO LEVEL  5 OF ACKER =  3
RETURN  TO LEVEL  4 OF ACKER =  3
CALL  FROM LEVEL  4 OF ACKER( 0, 3)
RETURN  TO LEVEL  4 OF ACKER =  4
RETURN  TO LEVEL  3 OF ACKER =  4
CALL  FROM LEVEL  3 OF ACKER( 0, 4)
RETURN  TO LEVEL  3 OF ACKER =  5
RETURN  TO LEVEL  2 OF ACKER =  5
RETURN  TO LEVEL  1 OF ACKER =  5
CALL  FROM LEVEL  1 OF ACKER( 1, 5)
CALL  FROM LEVEL  2 OF ACKER( 1, 4)
CALL  FROM LEVEL  3 OF ACKER( 1, 3)
CALL  FROM LEVEL  4 OF ACKER( 1, 2)
CALL  FROM LEVEL  5 OF ACKER( 1, 1)
CALL  FROM LEVEL  6 OF ACKER( 1, 0)
CALL  FROM LEVEL  7 OF ACKER( 0, 1)
RETURN  TO LEVEL  7 OF ACKER =  2
RETURN  TO LEVEL  6 OF ACKER =  2
CALL  FROM LEVEL  6 OF ACKER( 0, 2)
RETURN  TO LEVEL  6 OF ACKER =  3
RETURN  TO LEVEL  5 OF ACKER =  3
CALL  FROM LEVEL  5 OF ACKER( 0, 3)
RETURN  TO LEVEL  5 OF ACKER =  4
RETURN  TO LEVEL  4 OF ACKER =  4
CALL  FROM LEVEL  4 OF ACKER( 0, 4)
RETURN  TO LEVEL  4 OF ACKER =  5
RETURN  TO LEVEL  3 OF ACKER =  5
CALL  FROM LEVEL  3 OF ACKER( 0, 5)
RETURN  TO LEVEL  3 OF ACKER =  6
RETURN  TO LEVEL  2 OF ACKER =  6
CALL  FROM LEVEL  2 OF ACKER( 0, 6)
RETURN  TO LEVEL  2 OF ACKER =  7
RETURN  TO LEVEL  1 OF ACKER =  7
RETURN  TO LEVEL  0 OF ACKER =
ACKER(2,2) =  7
```

FIGURE 2-6.14 Mainline program and results for faking recursion of Ackermann's function.

In the statement labeled 50, the topmost three elements of the stack are deleted. In statements 23 and 24, control is transferred according to the unstacked return address. If that address is neither 40 nor 50, it must be 60, corresponding to the initial call, and thus we exit from ACKER.

A mainline program to test the FORTRAN function ACKER is given in Fig. 2-6.14, together with the results of computing $A\ \langle 2, 2\rangle$.

A nontrivial application to which recursion can be easily applied is mechanical theorem proving. This is the topic of the following section.

EXERCISES 2-6.2

1 The usual method for evaluating a polynomial of the form

$$p_n(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \cdots + a_{n-1}x + a_n$$

uses the technique known as nesting or Horner's rule. This is an iterative method which can be described as follows:

$$b_0 = a_0$$

$$b_{i+1} = x \cdot b_i + a_{i+1} \qquad i = 0, 1, \ldots, n-1$$

from which one can obtain $b_n = p_n(x)$.

An alternate solution to the problem is to write

$$p_n(x) = x \cdot p_{n-1}(x) + a_n$$

where

$$p_{n-1}(x) = a_0x^{n-1} + a_1x^{n-2} + \cdots + a_{n-2}x + a_{n-1}$$

which is a recursive formulation of the problem. Write a recursive function program to evaluate such a polynomial. Use as data $n = 3$, $a_0 = 1$, $a_1 = 3$, $a_2 = 3$, $a_3 = 1$ and $x = 2$.

2 Consider the set of all valid completely parenthesized infix arithmetic expressions consisting of single-letter variable names, nonnegative integers, and the four operators $+$, $-$, $*$, and $/$. The following recursive definition gives all such valid expressions:

1 Any single-letter variable (A to Z) or a nonnegative integer is a valid infix expression.

2 If γ and β are valid infix expressions, then $(\gamma + \beta)$, $(\gamma - \beta)$, $(\gamma * \beta)$, and (γ/β) are valid infix expressions.

3 The only valid infix expressions are those defined by steps 1 and 2.

Write a recursive function program which will have as input some string of symbols and which is to output "VALID EXPRESSION" if the input string is a valid infix expression and "INVALID EXPRESSION" otherwise. Write a main program to read the input data and invoke this function.

3 Write a recursive function program to compute the square root of a number. Read in triples of numbers N, A, and E, where N is the number for which the square root is to be found, A is an approximation of the square root, and E is the allowable error in the result. Use as your function

$$\text{ROOT(N, A, E)} = \begin{cases} \text{A} & \text{if } |\text{A}^2 - \text{N}| < \text{E} \\ \text{ROOT}\left(\text{N}, \dfrac{\text{A}^2 + \text{N}}{2\text{A}}, \text{E}\right) & \text{otherwise} \end{cases}$$

Scanned with OKEN Scanner

Use the following triples as test data.

2	1.0	0.001
3	1.5	0.001
8	2.5	0.001
225	14.2	0.001

4. Another common application for recursion is the problem of generating all possible permutations of a set of symbols. For the set consisting of symbols A, B, and C, there exist six permutations, namely, ABC, ACB, BAC, BCA, CBA, and CAB. The set of permutations of N symbols is generated by taking each symbol in turn and prefixing it to all the permutations which result from the remaining N − 1 symbols. It is therefore possible to specify the permutations of a set of symbols in terms of permutations of a smaller set of symbols. Write a recursive-function program for generating all possible permutations of a set of symbols.

5 In many applications it is required to know the number of different partitions of a given integer N, that is, how many different ways N can be expressed as a sum of integer summands. If we denote by Q_{MN} the number of ways in which an integer M can be expressed as a sum, each summand of which is no larger than N, then the number of partitions of N is given by Q_{NN}. The function Q_{MN} is defined recursively as

$$Q_{MN} = \begin{cases} 1 & \text{if } M = 1 \text{ and for all } N \\ 1 & \text{if } N = 1 \text{ and for all } M \\ Q_{MM} & \text{if } M < N \\ 1 + Q_{M,M-1} & \text{if } M = N \\ Q_{M,N-1} + Q_{M-N,N} & \text{if } M > N \end{cases}$$

Write a recursive-function program and use values of N = 3, 4, 5, 6 as data.

6 Recall from Sec. 1-2.7 the recursive definition of a well-formed formula (wff) for the propositional calculus. Develop algorithms and give a PL/I program for determining whether a formula is well-formed.

7 Formulate an algorithm and write a program for determining whether two statement formulas are equivalent. The approach which should be taken for solving this problem is to generate and compare the principal disjunctive (or conjunctive) normal forms for the statements. If they are the same, then the statement formulas are equivalent; otherwise they are not.

A procedure for manually obtaining the principal disjunctive normal form for a statement formula, as given in Sec. 1-3.3, consisted of the following steps:

1 Eliminate the unwanted connectives (such as $\rightarrow$ and $\rightleftarrows$).

2 Eliminate all occurrences of double negation and apply De Morgan's laws to distribute the negation operators. Repeat this step until De Morgan's laws can no longer be applied.

3 Distribute the conjunctions to obtain a disjunctive standard form.

4 In each term T_i of the standard form that has a variable missing, say Q, replace T_i by $(Q \vee \neg Q) \wedge T_i$. Duplicate terms and those containing $Q \wedge \neg Q$ are eliminated. This step is repeated until the principal disjunctive normal form is obtained.

This manual procedure can be formulated algorithmically by using string manipulation and recursion. The outline of an algorithm for generating the principal disjunctive normal form of a statement formula follows. It is assumed that the statement formula is well-formed.

The binary operators $\rightarrow$ and $\rightleftarrows$ are replaced by their equivalents in terms of operators $\vee$, $\wedge$, and $\neg$. Expressions in which a negation operator affects more than a single variable—for example, $\neg\neg(\neg(A \vee \neg B) \vee (\neg C \wedge D))$—must be dealt with. Occurrences of double negation are removed, and then De Morgan's laws are applied. Since this procedure may cause more occurrences of double negation, the procedure must be repeated until De Morgan's laws can no longer be applied.

When negation operators affect only single variables, it is possible to convert the statement formula to a disjunctive standard form by distributing conjunctions. For example, $((A \vee \neg B) \wedge (C \vee \neg D))$ must be converted to $(((A \wedge C) \vee (A \wedge \neg D)) \vee ((\neg B \wedge C) \vee (\neg B \wedge \neg D)))$. Recursion may prove useful for this step.

Finally, the disjunctive standard form must be converted to principal disjunctive normal form. Recursion should be used for generating the minterms, although the use of stacks is an alternative. Make sure that the duplicate or contradictory terms are eliminated. Since the normal form is to be compared with another normal form, both must depend on the same variables. It may therefore be necessary to include in a normal form variables which were not present in the original statement formula. Remember that the variables and minterms must be ordered to simplify the comparison of two normal forms.

This algorithm is the basis for determining the equivalence of two statement formulas. When both statements have been converted to their principal disjunctive normal form with ordering, they may easily be compared.

2-7 RECURSION IN MECHANICAL THEOREM PROVING

In Sec. 1-4.4 we presented a formulation of statement logic suitable for mechanical theorem proving. We now present algorithms that implement the mechanical theorem proving techniques based on this formulation.

Recall that in Sec. 1-4.4 when we were determining whether a particular argument was valid, we converted this problem to one of determining whether a proof existed for some formula corresponding to this argument. If a proof did exist, then the proof was constructed, thereby establishing the formula as a theorem. The procedure involved consisted of the following phases:

1 An analysis of the formula to determine if it had a proof

2 The construction of the proof if it was found to exist

The first phase was performed for all formulas, but for certain formulas (the nontheorems) the second phase could not take place. For the formulas which had proofs, the second phase consisted of merely retracing in reverse order the steps generated during the first phase. Therefore, the first phase was the important part of the procedure.

Let us consider in more detail this phase of analysis. Recall that each rule in this formulation of statement logic had one of the following forms:

1 If sequent α is a theorem, then sequent γ is a theorem.

2 If sequents α and β are theorems, then sequent γ is also a theorem.

Given a particular sequent, which we assumed to be a theorem, we selected a

Scanned with OKEN Scanner

"rule" which converted this sequent to one or two simpler sequents which were also expected to be theorems. This process continued until the only sequents left were connective-free sequents. Since each of these sequents was expected to be a theorem (because of our original assumptions), and since every connective-free sequent that is a theorem is also an axiom (and conversely), it was then a simple matter to determine whether the original formula was a theorem. We simply determined if all these connective-free sequents were axioms. If they were, then the original formula was a theorem, and its proof consisted of the sequents generated by the analysis phase taken in reverse order (i.e., starting with the axioms and moving via the rules to the original formula).

However, what were the "rules" applied in the analysis phase to produce simpler sequents? The rules, as written, produced more complex sequents rather than simpler sequents; they introduced connectives. Yet, we were eliminating connectives in order to produce simpler formulas. Actually, we were applying the converse of the rules. The rules, as written, introduce connectives; but each rule has a converse rule which is also valid. These converse rules eliminate connectives, and it is these that were used during the analysis phase.

Another way of looking at this process is to observe that the application of rules of elimination generated all the conditions necessary for the given formula to be a theorem. If at least one of these conditions was not met (i.e., some connective-free sequent turned out to be a nonaxiom), then the given formula could not be a theorem. If all conditions were met, then, jointly, these conditions were a sufficient condition for the given formula to be a theorem.

It may already have been noticed that the proof of a formula in this presentation of statement logic can be represented by a treelike structure. The original formula is the root of the tree, and the axioms are the leaves. If we adhere to the convention suggested in Sec. 1-4.4, in which the main connective of the leftmost formula is always chosen for elimination, then this tree will be generated in what is known as a "preorder fashion" during the analysis (i.e., first the root, then the left branch, and finally the right branch, assuming that a root which has but one branch has a left branch). The proof of the formula, if it exists, is simply the steps of the analysis taken in reverse order. This corresponds to what we might call a converse-postorder traversal of this proof tree, i.e., first the right branch, then the left branch, and finally the root where a single branch leading to a root is assumed to be a right branch. Since the theory of graphs (which includes such trees) is discussed in Chap. 5, we will not discuss trees any further at this point.

The phase of analysis in this formulation of theorem proving is a good example of a recursive procedure. For any given sequent, there is a termination test—is this sequent connective-free? If it is, then recursion stops and some explicit value (e.g., *true* meaning "the sequent is a theorem" or *false* for "the sequent is not a theorem") is assigned, and then this value is returned to the step that generated this sequent.

If the sequent is not connective-free, then another connective is eliminated to produce one or two simpler sequents; these are then analyzed by a recursive call to the analysis procedure. The new sequents, because they are simpler, are one step nearer a successful application of the termination test.

We now present the algorithms that implement the form of mechanical

theorem proving illustrated in Sec. 1-4.4. These algorithms are designed to perform the analysis phase. The construction of a proof (or synthesis phase), once it has been discovered to exist, reduces to the simple task of following, in reverse order, the steps produced during analysis. Consequently, we omit any treatment of actual proof construction. Since little tracing is used in explaining the algorithms, readers are encouraged to do their own tracing for sample inputs.

Note that the formulas in the sequents are all assumed to have no embedded blanks and to be in prefix form (see Sec. 1-3.6). Prefix formulas are used for two reasons:

1 The leftmost main connective is easier to determine for formulas in prefix form.

2 The determination of the operands of a connective is made easier when the formula is in prefix form.

This restriction to prefix formulas, though not affecting the validity of the technique, does require that the 10 rules of the system be expressed in a form modified to deal with prefix formulas.

Algorithm *DETECT* determines which connective, if any, is to be eliminated. The convention suggested in Sec. 1-4.4 is followed; namely, the main connective of the leftmost formula in the sequent is always chosen.

Algorithm *DETECT* Given a string of formulas, *FORM*, this procedure returns a numeric code for the type of leftmost connective found in *FORM*. Codes of 1, 2, 3, 4, and 5 are used for the symbols $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\rightleftarrows$ respectively. The code associated with no connective is 0. The position of the connective, i, is returned as a second parameter. *SYMBOLS* is defined in algorithm *MAIN* and is the string consisting of the five connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\rightleftarrows$.

1 [Initialize position] Set $i \leftarrow 0$.
2 [Formulas exhausted?] If $i \geq LENGTH(FORM)$ then set $DETECT \leftarrow 0$ and Exit.
3 [Get next symbol] Set $i \leftarrow i+1$ and $DETECT \leftarrow INDEX(SYMBOLS, SUB(FORM, i, 1))$.
4 [Is it a connective?] If $DETECT > 0$ then Exit; otherwise go to step 2. ////

Algorithm *OPERANDS* uses the results of the previous algorithm to extract from the sequent the operands of this connective and to delete from the sequent the entire formula.

The method used in algorithm *OPERANDS* for isolating the operands of a connective is based on certain theorems (presented in Chap. 3) that deal with criteria for well-formed prefix expressions. These criteria make use of the notion of the rank of a formula. The *rank* of a formula is defined as the sum of the ranks of its component symbols. The ranks of a binary connective, a unary connective, and a variable are defined to be 1, 0, and -1 respectively. It can be shown that a prefix formula is well-formed if and only if its rank is -1 and the rank of every proper substring that includes the first symbol of the formula is nonnegative. Thus, starting with the symbols immediately after a connective and adding the

Scanned with OKEN Scanner

ranks of these symbols one by one to a counter initially set to 1, as soon as the counter has a value of 0, we know that we have just taken the last symbol of a well-formed operand. To find a second operand (if the connective is a binary connective), we simply reset the counter to 1 and follow the same procedure.

To illustrate this method, we use it to determine the operands of the formula $\rightleftarrows P \wedge \vee P \neg QR$, the prefix form of $P \rightleftarrows ((P \vee \neg Q) \wedge R)$. The numbers below the symbols are the values of the counter as it is used to record rank totals for each operand.

$\rightleftarrows$	P	$\wedge$	$\vee$	P	$\neg$	Q	R
1	0						
	1	2	3	2	2	1	0

The first and second operands of the connective $\rightleftarrows$ are P and $\wedge \vee P \neg QR$ respectively.

Algorithm *OPERANDS* Given a string of formulas *FORM*, the position in the string of the leftmost connective, *POS*, and the number of operands appropriate to that connective, *NUM*, this procedure returns these operands as one of the parameters, the array *OPER*, and deletes the entire formula from *FORM*. *MAXL* serves two purposes in this algorithm. In the part that finds operands, it indicates the length of the string in which an operand must be found; in the part that deletes the entire formula, it indicates the length of the formula that must be deleted. Note that two blanks follow each formula within a string of formulas; these blanks, rather than commas, are used as separators. *LOC* is the position of the last symbol immediately preceding the substring that must be searched for an operand. *RANK* is the counter that records the total of the rank values, while j is an index counter used to select symbols one at a time from the substring to be searched. The symbol i is a counter that records the number of operands extracted, and *TEMP* is merely a temporary location holding one symbol from the substring. *SIGN* is a built-in function which returns a value of -1, 0, or 1, depending on its argument being negative, zero, or positive, respectively. The symbol '' is used to denote a string of length 0.

1 [Initialize position and operand array] Set $LOC \leftarrow POS$, $OPER[1] \leftarrow$ '', $OPER[2] \leftarrow$ ''.

2 [Operand-finding loop] Repeat steps 3 to 7 for $i = 1, 2, \ldots, NUM$.

3 [Initialize rank and index counters and set length limit] Set $RANK \leftarrow 1$, $j \leftarrow 0$, and $MAXL \leftarrow LENGTH(SUB(FORM, LOC + 1))$.

4 [Found an operand?] If $RANK = 0$ then go to step 7.

5 [Get next symbol if still a wff] Set $j \leftarrow j + 1$. If $j > MAXL$ then print error message and Exit; otherwise set $TEMP \leftarrow SUB(FORM, LOC + j, 1)$.

6 [Form new rank value if still a wff] If $TEMP =$ 'b' then print error message and Exit; otherwise set $RANK \leftarrow RANK + SIGN(INDEX(SYMBOLS, TEMP) - 1)$ and go to step 4.

7 [Extract operand and reset position] Set $OPER[i] \leftarrow SUB(FORM, LOC + 1, j)$ and $LOC \leftarrow LOC + j$.

8 [Get length of formula to be deleted] Set $MAXL \leftarrow LENGTH(OPER[1]) + LENGTH(OPER[2]) + 3$.

9 [Delete formula and exit] Set $FORM \leftarrow SUB(FORM, 1, POS - 1) \cdot SUB(FORM, POS + MAXL)$ and Exit. ////

Algorithm *HAO-WANG* is the main algorithm that performs the recursive analysis, and it operates as follows:

The sequent is printed. Algorithm *DETECT* is invoked to find a connective to eliminate. If one is found, then algorithm *OPERANDS* is used to isolate the operands of this connective and algorithm *HAO-WANG* is recursively invoked in a manner appropriate for whichever rule is to be applied. If no connective is found, then the sequent is tested to determine if it is an axiom.

All the arguments for any invocation of the algorithm are assumed to be passed by value (see Sec. 1-1).

Algorithm *HAO-WANG* This recursive procedure attempts to apply an elimination rule to a sequent which is presented as the pair of arguments *ANTE*, the antecedent, and *CONS*, the consequent. The sequent is printed, along with a nesting level number, *LEVEL* (initialized in algorithm *MAIN*), which aids in grouping sequents of proof sequences. In following the "leftmost first" rule, the antecedent is searched for a connective to eliminate. *TYPE* records the numeric code value for the connective found by algorithm *DETECT*, and *POS* is the position at which it is found in *ANTE*. If a connective is found, then algorithm *OPERANDS* is invoked to isolate the operands of that connective. These operands are returned via the array parameter *OP*, and *ANTE* is returned with this entire formula deleted from it. The appropriate elimination rule for the connective found in the antecedent is then applied by recursively invoking *HAO-WANG* with arguments (passed by value) constructed to conform to the requirements of the chosen rule. If no connective is found in the antecedent, then the consequent, *CONS*, is checked in an identical manner.

Finding a leftmost connective in *CONS* is followed by an invocation of *OPERANDS* to isolate the operands and an application of the appropriate elimination rule for a connective found in the consequent. If *CONS* should be connective-free, then the sequent is examined to determine whether it is an axiom. The variable i is an index of *ANTE*, while *TOKEN* contains one symbol of *ANTE* which is matched against each symbol of *CONS*. One successful match means the sequent was an axiom. In such a case, the procedure returns a value *true*. If the sequent was not an axiom, the procedure returns *false*.

1 [Increment level counter and print sequent] Set $LEVEL \leftarrow LEVEL + 1$ and print the sequent and level counter.

2 [Connective in antecedent?] Set $TYPE \leftarrow DETECT(ANTE, POS)$. If $TYPE = 0$ then go to step 4. If $TYPE = 1$ then call *OPERANDS* (1, *OP*, *ANTE*, *POS*); otherwise call *OPERANDS* (2, *OP*, *ANTE*, *POS*).

3 [Transfer to antecedent elimination rule] Go to the antecedent rule determined by the value of *TYPE* (one of steps 10 to 14 below).

4 [Connective in consequent?] Set $TYPE \leftarrow DETECT(CONS, POS)$.

Scanned with OKEN Scanner

If $TYPE = 0$ then go to step 6. If $TYPE = 1$ then call $OPERANDS$ (1, OP, $CONS$, POS); otherwise call $OPERANDS$ (2, OP, $CONS$, POS).

5 [Transfer to consequent elimination rule] Go to the consequent rule determined by the value of $TYPE$ (one of steps 15 to 19 below).

6 [Axiom-test loop for connective-free sequent] Repeat steps 7 and 8 for $i = 1, \ldots, LENGTH(ANTE)$.

7 [Get a symbol from antecedent] Set $TOKEN \leftarrow SUB(ANTE, i, 1)$.

8 [If nonblank, test against consequent symbols] If $TOKEN \neq$ 'B' then if $INDEX(CONS, TOKEN) \neq 0$ print 'Axiom', set $HAO\text{-}WANG \leftarrow true$, and Exit.

9 [A nonaxiom!] Print 'Nonaxiom', set $HAO\text{-}WANG \leftarrow false$, and Exit.

The next five steps are the antecedent rules. All arguments are passed by value.

10 [$TYPE = 1$; $\neg\Rightarrow$: If $\alpha, \beta \stackrel{*}{\Rightarrow} X, \gamma$ then $\alpha, \neg X, \stackrel{*}{\Rightarrow} \gamma$] Set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(ANTE, OP[1] \circ$ 'BB' $\circ CONS)$, $LEVEL \leftarrow LEVEL - 1$, and Exit.

11 [$TYPE = 2$; $\wedge\Rightarrow$: If $X, Y, \alpha, \beta \stackrel{*}{\Rightarrow} \gamma$ then $\alpha, \wedge XY, \beta \stackrel{*}{\Rightarrow} \gamma$] Set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(OP[1] \circ$ 'BB' $\circ OP[2] \circ$ 'BB' $\circ ANTE, CONS)$, $LEVEL \leftarrow LEVEL - 1$, and Exit.

12 [$TYPE = 3$; $\vee\Rightarrow$: If $X, \alpha, \beta \stackrel{*}{\Rightarrow} \gamma$ and $Y, \alpha, \beta \stackrel{*}{\Rightarrow} \gamma$ then $\alpha, VXY, \beta \stackrel{*}{\Rightarrow} \gamma$] Set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(OP[1] \circ$ 'BB' $\circ ANTE, CONS)$, $LEVEL \leftarrow LEVEL - 1$. If $HAO\text{-}WANG$ is *false* then Exit; otherwise set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(OP[2] \circ$ 'BB' $\circ ANTE, CONS)$, $LEVEL \leftarrow LEVEL - 1$, and Exit.

13 [$TYPE = 4$; $\rightarrow\Rightarrow$: If $Y, \alpha, \beta \stackrel{*}{\Rightarrow} \gamma$ and $\alpha, \beta \stackrel{*}{\Rightarrow} X, \gamma$ then $\alpha, \rightarrow XY, \beta \stackrel{*}{\Rightarrow} \gamma$] Set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(OP[2] \circ$ 'BB' $\circ ANTE, CONS)$, $LEVEL \leftarrow LEVEL - 1$. If $HAO\text{-}WANG$ is *false* then Exit; otherwise set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(ANTE, OP[1] \circ$ 'BB' $\circ CONS)$, $LEVEL \leftarrow LEVEL - 1$, and Exit.

14 [$TYPE = 5$; $\rightleftarrows\Rightarrow$: If $X, Y, \alpha, \beta \stackrel{*}{\Rightarrow} \gamma$ and $\alpha, \beta \stackrel{*}{\Rightarrow} X, Y, \gamma$ then $\alpha, \rightleftarrows XY, \beta \stackrel{*}{\Rightarrow} \gamma$] Set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(OP[1] \circ$ 'BB' $\circ OP[2] \circ$ 'BB' $\circ ANTE, CONS)$, $LEVEL \leftarrow LEVEL - 1$. If $HAO\text{-}WANG$ is *false* then Exit, otherwise set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(ANTE, OP[1] \circ$ 'BB' $\circ OP[2] \circ$ 'BB' $\circ CONS)$, $LEVEL \leftarrow LEVEL - 1$, and Exit.

The next five steps are the consequent rules. All arguments are passed by value.

15 [$TYPE = 1$; $\Rightarrow\neg$: If $X, \alpha \stackrel{*}{\Rightarrow} \beta, \gamma$ then $\alpha \stackrel{*}{\Rightarrow} \beta, \neg X, \gamma$] Set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(OP[1] \circ$ 'BB' $\circ ANTE, CONS)$, $LEVEL \leftarrow LEVEL - 1$, and Exit.

16 [$TYPE = 2$; $\Rightarrow\wedge$: If $\alpha \stackrel{*}{\Rightarrow} X, \beta, \gamma$ and $\alpha \stackrel{*}{\Rightarrow} Y, \beta, \gamma$ then $\alpha \stackrel{*}{\Rightarrow} \beta, \wedge XY, \gamma$] Set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(ANTE, OP[1] \circ$ 'BB' $\circ CONS)$, $LEVEL \leftarrow LEVEL - 1$. If $HAO\text{-}WANG$ is *false* then Exit; otherwise set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(ANTE, OP[2] \circ$ 'BB' $\circ CONS)$, $LEVEL \leftarrow LEVEL - 1$, and Exit.

17 [$TYPE = 3$; $\Rightarrow\vee$: If $\alpha \stackrel{*}{\Rightarrow} X, Y, \beta, \gamma$ then $\alpha \stackrel{*}{\Rightarrow} \beta, \vee XY, \gamma$] Set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(ANTE, OP[1] \circ$ 'BB' $\circ OP[2] \circ$ 'BB' $\circ CONS)$, $LEVEL \leftarrow LEVEL - 1$, and Exit.

18 [$TYPE = 4$; $\Rightarrow\rightarrow$: If $X, \alpha \stackrel{*}{\Rightarrow} Y, \beta, \gamma$ then $\alpha \stackrel{*}{\Rightarrow} \beta, \rightarrow XY, \gamma$] Set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(OP[1] \circ$ 'BB' $\circ ANTE, OP[2] \circ$ 'BB' $\circ CONS)$, $LEVEL \leftarrow LEVEL - 1$, and Exit.

19 [$TYPE = 5$; $\Rightarrow\rightleftarrows$: If $X, \alpha \stackrel{*}{\Rightarrow} Y, \beta, \gamma$ and $Y, \alpha \stackrel{*}{\Rightarrow} X, \beta, \gamma$ then $\alpha \stackrel{*}{\Rightarrow} \beta, \rightleftarrows XY, \gamma$] Set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(OP[1] \circ$ 'BB' $\circ ANTE, OP[2] \circ$ 'BB' $\circ CONS)$, $LEVEL \leftarrow LEVEL - 1$. If $HAO\text{-}WANG$ is *false* then Exit; otherwise set $HAO\text{-}WANG \leftarrow HAO\text{-}WANG(OP[2] \circ$ 'BB' $\circ ANTE, OP[1] \circ$ 'BB' $\circ CONS)$, $LEVEL \leftarrow LEVEL - 1$, and Exit. ////

Algorithm *MAIN* This procedure reads in a well-formed prefix expression, *EXPRESSION*, prints it, and then calls algorithm *HAO-WANG* to analyze it. The results of the analysis are then printed. The variables *SYMBOLS* and

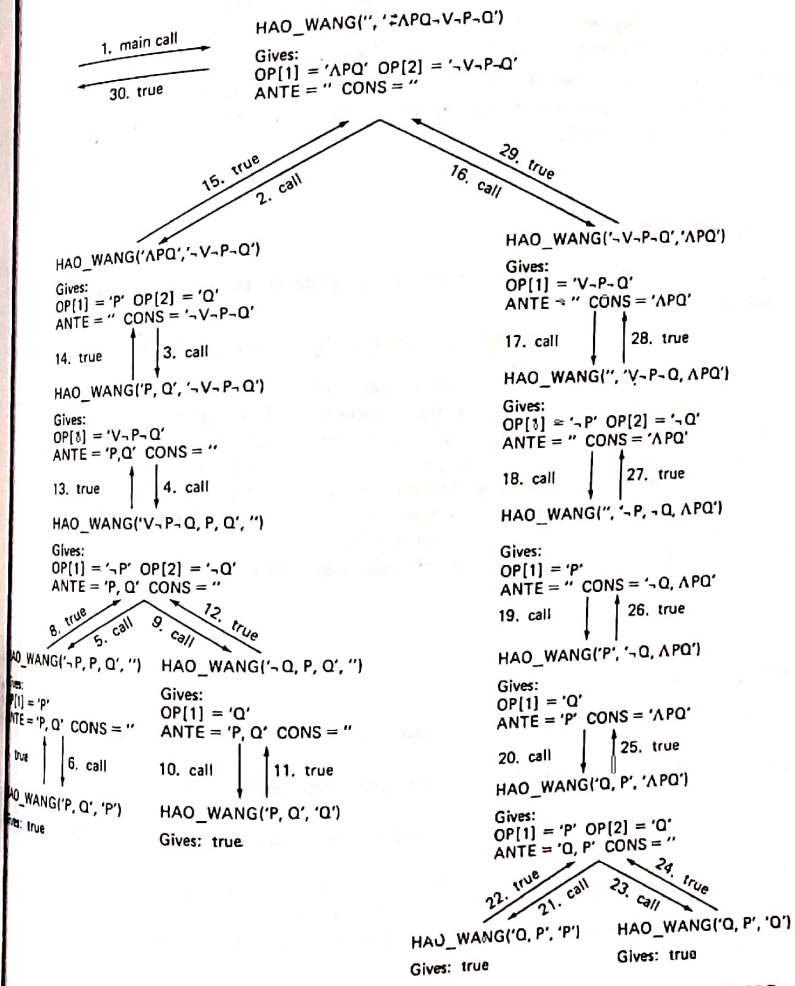


FIGURE 2-7.1 Call and return sequence of steps for algorithm HAO-WANG.

Scanned with OKEN Scanner

LEVEL, used in the other algorithms, are defined and initialized here. *SYMBOLS* is simply a string formed from the valid connectives, while *LEVEL* indicates the depth to which recursive nesting of calls has occurred.

1 [Define the connectives] Set $SYMBOLS \leftarrow '\neg \wedge \vee \rightarrow \rightleftarrows'$.

2 [Finished processing?] If out of data then Exit.

3 [Set level counter and do I/O] Set $LEVEL \leftarrow -1$, read *EXPRESSION*, and print *EXPRESSION*.

4 [Process and print results] If *HAO-WANG*('', *EXPRESSION* • '⇐') is *true* then print 'Theorem'; otherwise print 'Nontheorem'. Go to step 2. ////

Figure 2-7.1 is a flow-of-control diagram showing the call and return sequence of steps for algorithm *HAO-WANG* during the analysis of the theorem $\rightleftarrows \wedge PQ \neg \vee \neg P \neg Q$, the prefix form of $(P \wedge Q) \rightleftarrows \neg(\neg P \vee \neg Q)$. The transfers are numbered according to the order of their execution. The diagram also indicates, for each invocation of *HAO-WANG*, the operands that are found and the new antecedent and consequent pair from which the formula containing these operands has been deleted.

EXERCISES 2-7

1 Trace the operation of the theorem-proving program on the following formulas as input:

$$\rightleftarrows \neg \rightleftarrows PQ \vee \wedge \neg PQ \wedge P \neg Q \qquad \rightleftarrows \wedge S \vee P \neg PS$$

Do this by drawing the trees that could be associated with the analysis phase as it would be performed if these formulas were processed by the program.

2 Develop the appropriate rules for *NAND* and *NOR*, assuming their usual definitions in propositional logic (that is, $\uparrow \Rightarrow, \downarrow \Rightarrow, \Rightarrow \uparrow, \Rightarrow \downarrow$). Usng these rules, extend and modify the previous algorithms so that they are applicable to formulas that contain $\uparrow$ and $\downarrow$ as well as the usual connectives. (To do this, you will have to make minor changes in *DETECT*, *HAO-WANG*, and *MAIN*.)

3 Write a PL/I program which implements the algorithms *DETECT*, *HAO-WANG*, and *MAIN*.

BIBLIOGRAPHY

ABBOTT, JAMES C.: "Sets, Lattices, and Boolean Algebras," Allyn and Bacon, Inc., Boston, 1969.

BARRON, D. W.: "Recursive Techniques in Programming," American Elsevier Publishing Co., Inc., New York, 1968.

BERTISS, A. T.: "Data Structures: Theory and Practice," Academic Press, Inc., New York, 1971.

D'IMPERIO, M. E.: Data Structures and Their Representation in Storage, "Annual Review in Automatic Programming," vol. 5, pp. 1–75, Pergamon Press, Oxford, 1969.

HARRISON, M. C.: "Data Structures and Programming," Scott, Foresman and Company, Glenview, Illinois, 1973.

KNUTH, D. E.: "The Art of Computer Programming, vol. 1, Fundamental Algorithms," Addison-Wesley Publishing Company, Inc., Reading, Mass., 1968.

LIPSCHUTZ, SEYMOUR: "Theory and Problems of Set Theory and Related Topics," Schaum Publishing Company, New York, 1964.

NELSON, RAYMOND J.: "Introduction to Automata," John Wiley & Sons, Inc., New York, 1968.

PRATHER, RONALD E.: "Introduction to Switching Theory," Allyn and Bacon, Inc., Boston, 1967.

RALSTON, ANTHONY: "Introduction to Programming and Computer Science," McGraw-Hill Book Company, New York, 1971.

RICE, H. G.: Recursion and Iteration, *Communications of the Association for Computing Machinery*, 8(2): 114–115 (February, 1965).

STOLL, ROBERT R.: "Set Theory and Logic," W. H. Freeman and Company, Publishers, San Francisco, 1963.

SUPPES, E. H.: "Axiomatic Set Theory," D. Van Nostrand and Company, Inc., Princeton, N. J., 1960.

TREMBLAY, J. P., and P. G. SORENSON: "An Introduction to Data Structures with Applications," Lecture notes, University of Saskatchewan, Saskatoon, Saskatchewan, 1974. (To be published by the McGraw-Hill Book Company, New York, 1976.)

WANG, H.: Toward Mechanical Mathematics, *IBM J. Res. and Devel.*, **4**: 2–22 (January, 1960).

WELLS, CHARLES: "Mathematical Structures," Lecture notes, Case Western Reserve University, Cleveland, Ohio, 1968.

Scanned with OKEN Scanner