Chapter 1

INTRODUCTION TO THE THEORY OF COMPUTATION

Computer science is a practical discipline. Those who work in it often have a marked preference for useful and tangible problems over theoretical speculation. This is certainly true of computer science students who are interested mainly in working on difficult applications from the real world. Theoretical questions are interesting to them only if they help in finding good solutions. This attitude is appropriate, since without applications there would be little interest in computers. But given this practical orientation, one might well ask "why study theory?"

The first answer is that theory provides concepts and principles that help us understand the general nature of the discipline. The field of computer science includes a wide range of special topics, from machine design to programming. The use of computers in the real world involves a wealth of specific detail that must be learned for a successful application. This makes computer science a very diverse and broad discipline. But in spite of this diversity, there are some common underlying principles. To study these basic principles, we construct abstract models of computers and computation. These models embody the important features that are common to both hardware and software, and that are essential to many of the special and complex constructs we encounter while working with computers. Even

when such models are too simple to be applicable immediately to real-world situations, the insights we gain from studying them provide the foundations on which specific development is based. This approach is of course not unique to computer science. The construction of models is one of the essentials of any scientific discipline, and the usefulness of a discipline is often dependent on the existence of simple, yet powerful, theories and laws.

A second, and perhaps not so obvious answer, is that the ideas we will discuss have some immediate and important applications. The fields of digital design, programming languages, and compilers are the most obvious examples, but there are many others. The concepts we study here run like a thread through much of computer science, from operating systems to pattern recognition.

The third answer is one of which we hope to convince the reader. The subject matter is intellectually stimulating and fun. It provides many challenging, puzzle-like problems that can lead to some sleepless nights. This is problem-solving in its pure essence.

In this book, we will look at models that represent features at the core of all computers and their applications. To model the hardware of a computer, we introduce the notion of an **automaton** (plural, **automata**). An automaton is a construct that possesses all the indispensable features of a digital computer. It accepts input, produces output, may have some temporary storage, and can make decisions in transforming the input into the output. A **formal language** is an abstraction of the general characteristics of programming languages. A formal language consists of a set of symbols and some rules of formation by which these symbols can be combined into entities called sentences. A formal language is the set of all strings permitted by the rules of formation. Although some of the formal languages we study here are simpler than programming languages, they have many of the same essential features. We can learn a great deal about programming languages from formal languages. Finally, we will formalize the concept of a mechanical computation by giving a precise definition of the term **algorithm** and study the kinds of problems that are (and are not) suitable for solution by such mechanical means. In the course of our study, we will show the close connection between these abstractions and investigate the conclusions we can derive from them.

In the first chapter, we look at these basic ideas in a very broad way to set the stage for later work. In Section 1.1, we review the main ideas from mathematics that will be required. While intuition will frequently be our guide in exploring ideas, the conclusions we draw will be based on rigorous arguments. This will involve some mathematical machinery, although these requirements are not extensive. The reader will need a reasonably good grasp of the terminology and of the elementary results of set theory, functions, and relations. Trees and graph structures will be used frequently, although little is needed beyond the definition of a labeled, directed graph. Perhaps the most stringent requirement is the ability to follow proofs and

an understanding of what constitutes proper mathematical reasoning. This includes familiarity with the basic proof techniques of deduction, induction, and proof by contradiction. We will assume that the reader has this necessary background. Section 1.1 is included to review some of the main results that will be used and to establish a notational common ground for subsequent discussion.

In Section 1.2, we take a first look at the central concepts of languages, grammars, and automata. These concepts occur in many specific forms throughout the book. In Section 1.3, we give some simple applications of these general ideas to illustrate that these concepts have widespread uses in computer science. The discussion in these two sections will be intuitive rather than rigorous. Later, we will make all of this much more precise; but for the moment, the goal is to get a clear picture of the concepts with which we are dealing.

1.1 Mathematical Preliminaries and Notation

Sets

A **set** is a collection of elements, without any structure other than membership. To indicate that x is an element of the set S, we write $x \in S$. The statement that x is not in S is written $x \notin S$. A set is specified by enclosing some description of its elements in curly braces; for example, the set of integers 0, 1, 2 is shown as

$$S = \{0, 1, 2\}.$$

Ellipses are used whenever the meaning is clear. Thus, $\{a, b, \ldots, z\}$ stands for all the lower-case letters of the English alphabet, while $\{2, 4, 6, \ldots\}$ denotes the set of all positive even integers. When the need arises, we use more explicit notation, in which we write

$$S = \{i : i > 0, \, i \text{ is even}\} \tag{1.1}$$

for the last example. We read this as "S is set of all i, such that i is greater than zero, and i is even," implying of course that i is an integer.

The usual set operations are **union** ($\cup$), **intersection** ($\cap$), and **difference** ($-$), defined as

$$\begin{aligned} S_1 \cup S_2 &= \{x : x \in S_1 \text{ or } x \in S_2\}, \\ S_1 \cap S_2 &= \{x : x \in S_1 \text{ and } x \in S_2\}, \\ S_1 - S_2 &= \{x : x \in S_1 \text{ and } x \notin S_2\}. \end{aligned}$$

Another basic operation is **complementation.** The complement of a set S, denoted by $\overline{S}$, consists of all elements not in S. To make this

meaningful, we need to know what the **universal set** U of all possible elements is. If U is specified, then

$$\overline{S} = \{x : x \in U, x \notin S\}.$$

The set with no elements, called the **empty set** or the **null set** is denoted by $\varnothing$. From the definition of a set, it is obvious that

$$S \cup \varnothing = S - \varnothing = S,$$
$$S \cap \varnothing = \varnothing,$$
$$\overline{\varnothing} = U,$$
$$\overline{\overline{S}} = S.$$

The following useful identities, known as the **DeMorgan's laws,**

$$\overline{S_1 \cup S_2} = \overline{S}_1 \cap \overline{S}_2, \tag{1.2}$$
$$\overline{S_1 \cap S_2} = \overline{S}_1 \cup \overline{S}_2, \tag{1.3}$$

are needed on several occasions.

A set S_1 is said to be a **subset** of S if every element of S_1 is also an element of S. We write this as

$$S_1 \subseteq S.$$

If $S_1 \subseteq S$, but S contains an element not in S_1 we say that S_1 is a **proper subset** of S; we write this as

$$S_1 \subset S.$$

If S_1 and S_2 have no common element, that is, $S_1 \cap S_2 = \varnothing$, then the sets are said to be **disjoint.**

A set is said to be finite if it contains a finite number of elements; otherwise it is infinite. The size of a finite set is the number of elements in it; this is denoted by $|S|$.

A given set normally has many subsets. The set of all subsets of a set S is called the **powerset** of S and is denoted by 2^S. Observe that 2^S is a set of sets.

Example 1.1 If S is the set $\{a, b, c\}$, then its powerset is

$$2^S = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}.$$

Here $|S| = 3$ and $\left|2^S\right| = 8$. This is an instance of a general result; if S is finite, then

$$\left|2^S\right| = 2^{|S|}.$$

■

In many of our examples, the elements of a set are ordered sequences of elements from other sets. Such sets are said to be the **Cartesian product** of other sets. For the Cartesian product of two sets, which itself is a set of ordered pairs, we write

$$S = S_1 \times S_2 = \{(x, y) : x \in S_1, y \in S_2\}.$$

Example 1.2 Let $S_1 = \{2, 4\}$ and $S_2 = \{2, 3, 5, 6\}$. Then

$$S_1 \times S_2 = \{(2,2), (2,3), (2,5), (2,6), (4,2), (4,3), (4,5), (4,6)\}.$$

Note that the order in which the elements of a pair are written matters. The pair $(4, 2)$ is in $S_1 \times S_2$, but $(2, 4)$ is not.

The notation is extended in an obvious fashion to the Cartesian product of more than two sets; generally

$$S_1 \times S_2 \times \cdots \times S_n = \{(x_1, x_2, ..., x_n) : x_i \in S_i\}.$$

Functions and Relations

A **function** is a rule that assigns to elements of one set a unique element of another set. If f denotes a function, then the first set is called the **domain** of f, and the second set is its **range.** We write

$$f : S_1 \rightarrow S_2$$

to indicate that the domain of f is a subset of S_1 and that the range of f is a subset of S_2. If the domain of f is all of S_1, we say that f is a **total function** on S_1; otherwise f is said to be a **partial function.**

In many applications, the domain and range of the functions involved are in the set of positive integers. Furthermore, we are often interested only in the behavior of these functions as their arguments become very large. In such cases an understanding of the growth rates is often sufficient and a common order of magnitude notation can be used. Let $f(n)$ and $g(n)$ be functions whose domain is a subset of the positive integers. If there exists a positive constant c such that for all n

$$f(n) \leq cg(n),$$

we say that f has **order at most** g. We write this as

$$f(n) = O(g(n)).$$

If

$$|f(n)| \geq c\,|g(n)|,$$

then f has **order at least** g, for which we use

$$f(n) = \Omega(g(n)).$$

Finally, if there exist constants c_1 and c_2 such that

$$c_1\,|g(n)| \leq |f(n)| \leq c_2\,|g(n)|,$$

f and g have the **same order of magnitude**, expressed as

$$f(n) = \Theta(g(n)).$$

In this order of magnitude notation, we ignore multiplicative constants and lower order terms that become negligible as n increases.

Example 1.3 Let

$$f(n) = 2n^2 + 3n,$$
$$g(n) = n^3,$$
$$h(n) = 10n^2 + 100.$$

Then

$$f(n) = O(g(n)),$$
$$g(n) = \Omega(h(n)),$$
$$f(n) = \Theta(h(n)).$$

In order of magnitude notation, the symbol = should not be interpreted as equality and order of magnitude expressions cannot be treated like ordinary expressions. Manipulations such as

$$O(n) + O(n) = 2O(n)$$

are not sensible and can lead to incorrect conclusions. Still, if used properly, the order of magnitude arguments can be effective, as we will see in later chapters on the analysis of algorithms. ■

Some functions can be represented by a set of pairs

$$\{(x_1, y_1), (x_2, y_2), \ldots\},$$

where x_i is an element in the domain of the function, and y_i is the corresponding value in its range. For such a set to define a function, each x_i can occur at most once as the first element of a pair. If this is not satisfied, the

set is called a **relation.** Relations are more general than functions: in a function each element of the domain has exactly one associated element in the range; in a relation there may be several such elements in the range.

One kind of relation is that of **equivalence,** a generalization of the concept of equality (identity). To indicate that a pair (x, y) is an equivalence relation, we write

$$x \equiv y.$$

A relation denoted by $\equiv$ is considered an equivalence if it satisfies three rules: the reflexivity rule

$$x \equiv x \text{ for all } x,$$

the symmetry rule

$$\text{if } x \equiv y \text{ then } y \equiv x,$$

and the transitivity rule

$$\text{if } x \equiv y \text{ and } y \equiv z, \text{ then } x \equiv z.$$

Example 1.4 Consider the relation on the set of nonnegative integers defined by

$$x \equiv y,$$

if and only if

$$x \bmod 3 = y \bmod 3.$$

Then $2 \equiv 5$, $12 \equiv 0$, and $0 \equiv 36$. Clearly this is an equivalence relation, as it satisfies reflexivity, symmetry, and transitivity. ■

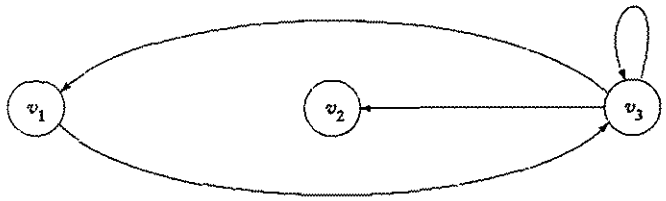
Graphs and Trees

A graph is a construct consisting of two finite sets, the set $V = \{v_1, v_2, ..., v_n\}$ of **vertices** and the set $E = \{e_1, e_2, ..., e_m\}$ of **edges.** Each edge is a pair of vertices from V, for instance

$$e_i = (v_j, v_k)$$

is an edge from v_j to v_k. We say that the edge e_i is an outgoing edge for v_j and an incoming edge for v_k. Such a construct is actually a directed graph (digraph), since we associate a direction (from v_j to v_k) with each edge. Graphs may be labeled, a label being a name or other information associated with parts of the graph. Both vertices and edges may be labeled.

Figure 1.1



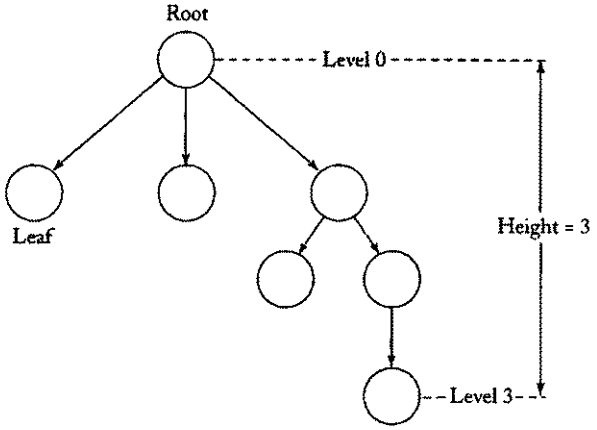
Graphs are conveniently visualized by diagrams in which the vertices are represented as circles and the edges as lines with arrows connecting the vertices. The graph with vertices $\{v_1, v_2, v_3\}$ and edges $\{(v_1, v_3), (v_3, v_1), (v_3, v_2), (v_3, v_3)\}$ is depicted in Figure 1.1.

A sequence of edges $(v_i, v_j), (v_j, v_k), \ldots, (v_m, v_n)$ is said to be a **walk** from v_i to v_n. The length of a walk is the total number of edges traversed in going from the initial vertex to the final one. A walk in which no edge is repeated is said to be a **path;** a path is **simple** if no vertex is repeated. A walk from v_i to itself with no repeated edges is called a **cycle** with **base** v_i. If no vertices other than the base are repeated in a cycle, then it is said to be simple. In Figure 1.1, (v_1, v_3), (v_3, v_2) is a simple path from v_1 to v_2. The sequence of edges (v_1, v_3), (v_3, v_3), (v_3, v_1) is a cycle, but not a simple one. If the edges of a graph are labeled, we can talk about the label of a walk. This label is the sequence of edge labels encountered when the path is traversed. Finally, an edge from a vertex to itself is called a **loop**. In Figure 1.1 there is a loop on vertex v_3.

On several occasions, we will refer to an algorithm for finding all simple paths between two given vertices (or all simple cycles based on a vertex). If we do not concern ourselves with efficiency, we can use the following obvious method. Starting from the given vertex, say v_i, list all outgoing edges (v_i, v_k), $(v_i, v_l), \ldots$. At this point, we have all paths of length one starting at v_i. For all vertices $v_k, v_l, \ldots$ so reached, we list all outgoing edges as long as they do not lead to any vertex already used in the path we are constructing. After we do this, we will have all simple paths of length two originating at v_i. We continue this until all possibilities are accounted for. Since there are only a finite number of vertices, we will eventually list all simple paths beginning at v_i. From these we select those ending at the desired vertex.

Trees are a particular type of graph. A tree is a directed graph that has no cycles, and that has one distinct vertex, called the **root,** such that there is exactly one path from the root to every other vertex. This definition implies that the root has no incoming edges and that there are some vertices without outgoing edges. These are called the **leaves** of the tree. If there is an edge from v_i to v_j, then v_i is said to be the **parent** of v_j, and v_j the **child** of v_i. The **level** associated with each vertex is the number of edges in the path from the root to the vertex. The **height** of the tree is the largest level number of any vertex. These terms are illustrated in Figure 1.2.

Figure 1.2



At times, we want to associate an ordering with the nodes at each level; in such cases we talk about **ordered trees.**

More details on graphs and trees can be found in most books on discrete mathematics.

Proof Techniques

An important requirement for reading this text is the ability to follow proofs. In mathematical arguments, we employ the accepted rules of deductive reasoning, and many proofs are simply a sequence of such steps. Two special proof techniques are used so frequently that it is appropriate to review them briefly. These are **proof by induction** and **proof by contradiction.**

Induction is a technique by which the truth of a number of statements can be inferred from the truth of a few specific instances. Suppose we have a sequence of statements $P_1, P_2, \ldots$ we want to prove to be true. Furthermore, suppose also that the following holds:

1. For some $k \geq 1$, we know that $P_1, P_2, \ldots, P_k$ are true.
2. The problem is such that for any $n \geq k$, the truths of $P_1, P_2, \ldots, P_n$ imply the truth of P_{n+1}.

We can then use induction to show that every statement in this sequence is true.

In a proof by induction, we argue as follows: From Condition 1 we know that the first k statements are true. Then Condition 2 tells us that P_{k+1} also must be true. But now that we know that the first $k+1$ statements are true, we can apply Condition 2 again to claim that P_{k+2} must be true, and so on. We need not explicitly continue this argument because the pattern is clear. The chain of reasoning can be extended to any statement. Therefore, every statement is true.

from v_i to v_n. The length of a walk is the total number of edges traversed in going from the intial vertex to the final one. A walk in which no edge is repeated is said to be a **path**; a path is **simple** if no vertex is repeated. A walk from v_i to itself with no repeated edges is called a **cycle** with base v_i. If no vertices other than the base are repeated in a cycle, then it is said to be simple. In Figure 1.1, (v_1, v_3), (v_3, v_2) is a simple path from v_1 to v_2. The sequence of edges (v_1, v_3), (v_3, v_3), (v_3, v_1) is a cycle, but not a simple one. If the edges of a graph are labeled, we can talk about the label of a walk. This label is the sequence of edge labels encountered when the path is traversed.

On several occasions, we will refer to an algorithm for finding all simple paths between two given vertices (or all simple cycles based on a vertex). If we do not concern ourselves with efficiency, we can use the following obvious method. Starting from the given vertex, say v_i, list all outgoing edges (v_i, v_k), (v_i, v_l), ... At this point, we have all paths of length one starting at v_i. For all vertices v_k, v_l, . . . so reached, we list all outgoing edges as long as they do not lead to any vertex already used in the path we are constructing. After we do this, we will have all simple paths of length two originating at v_i. We continue this until all possibilities are accounted for. Since there are only a finite number of vertices, we will eventually list all simple paths beginning at v_i. From these we select those ending at the desired vertex.

Trees are a particular type of graph. A tree is a directed graph that has no cycles, and that has one distinct vertex, called the **root**, such that there is exactly one path from the root to every other vertex. This definition implies that the root has no incoming edges and that there are some vertices without outgoing edges. These are called the **leaves** of the tree. If there is an edge from v_i to v_j, then v_i is said to be the **parent** of v_j, and v_j the **child** of v_i. The level associated with each vertex is the number of edges in the path from the root to the vertex. The **height** of the tree is the largest level number of any vertex. These terms are illustrated in Figure 1.2.

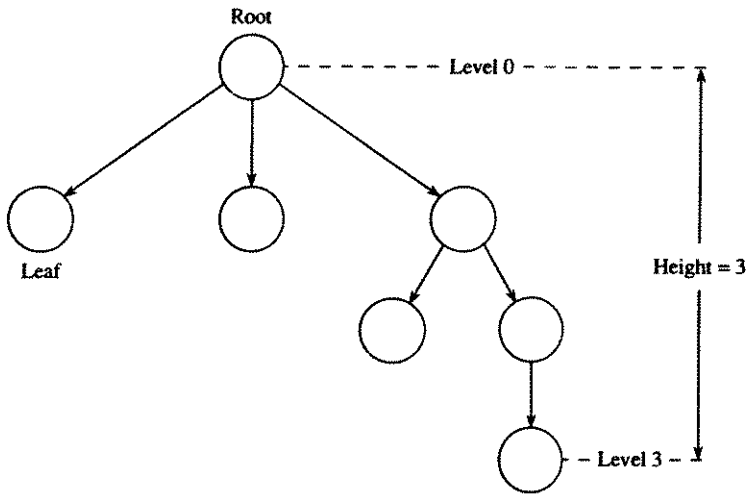
At times, we want to associate an ordering with the nodes at each level; in such cases we talk about **ordered trees**.

More details on graphs and trees can be found in most books on discrete mathematics, such as Johnsonbaugh 1986.

Proof Techniques

An important requirement for reading this text is the ability to follow proofs. In mathematical arguments, we employ the accepted rules of deductive reasoning, and many proofs are simply a sequence of such steps.

Figure 1.2



Two special proof techniques are used so frequently that it is appropriate to review them briefly. These are **proof by induction** and **proof by contradiction.**

In proof by induction, we have a sequence of statements $P_1, P_2, \ldots$ about which we want to make some claim. Suppose that we know that the claim holds for all statements $P_1, P_2, \ldots$ up to P_n. We then try to argue that this implies that the claim holds also for P_{n+1}. If we can carry out this inductive step for all positive n, and if we have some starting point for the induction, we can then say that the claim holds for all statements in the sequence.

The starting point for an induction is called the **basis.** The assumption that the claim holds for statements $P_1, P_2, \ldots, P_n$ is the **inductive assumption,** and the argument connecting the inductive assumption to P_{n+1} is the **inductive step.** At times, inductive arguments become clearer if we explicitly show these three parts.

▶ ***Example 1.4*** Prove that a binary tree of height n has at most 2^n leaves.

Proof: If we denote the maximum number of leaves of a binary tree of height n by $l(n)$, then we want to show that

$$l(n) \leq 2^n.$$

Basis: Clearly $l(0) = 1 = 2^0$ since a tree of height 0 can have no nodes other than the root, that is, it has at most one node.

Inductive Assumption: $l(i) \leq 2^i$, for $i = 0, 1, \ldots, n$.

Inductive Step: To get a binary tree of height $n + 1$ from one of height n, we can create at most two leaves in place of each previous one. Therefore

$$l(n + 1) = 2l(n).$$

Now, using the inductive assumption, we get

$$l(n + 1) \leq 2 \times 2^n = 2^{n+1}.$$

Thus, our claim is also true for $n + 1$. Since n can be any number, the statement must be true for all n. ■

Here we introduce the symbol ■ that is used in this book to denote the end of a proof.

▸ ***Example 1.5*** Show that

$$S_n = \sum_{i=0}^{n} i = \frac{n(n + 1)}{2}. \tag{1.4}$$

First we note that

$$S_{n+1} = S_n + n + 1.$$

We then make the inductive assumption that (1.4) holds for S_n; if this is so, then

$$\begin{aligned} S_{n+1} &= \frac{n(n + 1)}{2} + n + 1 \\ &= \frac{(n + 2)(n + 1)}{2}. \end{aligned}$$

Thus (1.4) holds for S_{n+1} and we have justified the inductive step. Since (1.4) is obviously true for $n = 1$, we have a basis and have proved (1.4) by induction for all n.

In this last example we have been a little less formal in identifying the basis, inductive assumption and inductive step, but they are there and are essential. To keep our subsequent discussions from becoming too formal, we will generally prefer the style of the second example. However, if you have difficulty in following or constructing a proof, go back to the more explicit form of Example 1.4.

Proofs by contradiction are also used often. This technique is powerful in some situations where other attacks fail. The following is a classical and elegant example.

▸ ***Example 1.6*** Show that $\sqrt{2}$ is not a rational number. As in all proofs by contradiction, we assume the contrary of what we want to show. Here we assume that $\sqrt{2}$ is a rational number so that it can be written as

$$\sqrt{2} = \frac{n}{m}, \tag{1.5}$$

where n and m are integers without a common factor. Rearranging (1.5), we have

$$2m^2 = n^2.$$

Therefore n^2 must be even. This implies that n is even, so that we can write $n = 2k$ or

$$2m^2 = 4k^2,$$

and

$$m^2 = 2k^2.$$

Therefore m is even. But this contradicts our assumption that n and m have no common factors. Thus, m and n in (1.5) cannot exist and $\sqrt{2}$ is not a rational number.

This example exhibits the essence of a proof by contradiction. By making a certain assumption we are led to a contradiction of the assumption or some known fact. If all steps in our argument are logically sound, we must conclude that our initial assumption was false.

▸ EXERCISES

1. Use induction to show that if S is a finite set then $|2^S| = 2^{|S|}$.
2. Show that if S_1 and S_2 are finite sets with $|S_1| = n$ and $|S_2| = m$, then

$$|S_1 \cup S_2| \leq n + m.$$

3. If S_1 and S_2 are finite sets with $|S_1| = n$ and $|S_2| = m$, show that $|S_1 \times S_2| = mn$.
4. Consider the relation between two sets defined by $S_1 \equiv S_2$ if and only if $|S_1| = |S_2|$. Show that this is an equivalence relation.
5. Prove deMorgan's laws, equations (1.2) and (1.3).
6. Occasionally, we need to use the union and intersection symbols in a manner analogous to the summation sign Σ. We define

$$\bigcup_{p\in\{i,j,k,\ldots\}} S_p = S_i \cup S_j \cup S_k \cdots$$

with an analogous notation for the intersection of several sets.

With this notation, the general deMorgan's laws are written as

$$\overline{\bigcup_{p\in P} S_p} = \bigcap_{p\in P} \overline{S_p}$$

and

$$\overline{\bigcap_{p\in P} S_p} = \bigcup_{p\in P} \overline{S_p}.$$

Prove these identities when P is a finite set.

7. Show that

$$S_1 \cup S_2 = \overline{\overline{S_1} \cap \overline{S_2}}.$$

8. Show that $S_1 = S_2$ if and only if

$$(S_1 \cap \overline{S_2}) \cup (\overline{S_1} \cap S_2) = \varnothing.$$

9. Show that

$$S_1 \cup S_2 - (S_1 \cap \overline{S_2}) = S_2.$$

10. Show that

$$S_1 \times (S_2 \cup S_3) = (S_1 \times S_2) \cup (S_1 \times S_3).$$

11. Show that if $S_1 \subseteq S_2$, then $\overline{S}_2 \subseteq \overline{S}_1$.

12. Draw a picture of the graph with vertices $\{v_1, v_2, v_3\}$ and edges $\{(v_1, v_1), (v_1, v_2), (v_2, v_3), (v_2, v_1), (v_3, v_1)\}$. Enumerate all cycles with base v_1.

13. Let $G = (V, E)$ be any graph. Prove the following claim: If there is any walk between $v_i \in V$ and $v_j \in V$, then there must a path of length no larger than $|V| - 1$ between these two vertices.

14. Consider graphs in which there is at most one edge between any two vertices. Show that under this condition a graph with n vertices has at most n^2 edges.

15. Show that

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

16. Show that

$$\sum_{i=1}^{n} \frac{1}{i^2} \leq 2 - \frac{1}{n}.$$

17. Prove that for all $n \geq 4$ the inequality $2^n < n!$ holds.

18. Show that $\sqrt{8}$ is not a rational number.

19. Prove that the set of all prime numbers is infinite.

▸ 1.2

THREE BASIC CONCEPTS

Three fundamental ideas are the major themes of this book: **languages**, **grammar**, and **automata**. In the course of our study we will explore many results about these concepts and about their relationship to each other. First, we must understand the meaning of the terms.

Languages

We are all familiar with the notion of natural languages, such as English and French. Still, most of us would probably find it difficult to say exactly what the word "language" means. Dictionaries define the term informally as a system suitable for the expression of certain ideas, facts, or concepts, including a set of symbols and rules for their manipulation. While this gives us an intuitive idea what a language is, it is not sufficient as a definition for the study of formal languages. We need a precise definition for the term.

We start with a finite, nonempty set Σ of symbols, called the **alphabet**. From the individual symbols we construct **strings**, which are finite sequences of symbols from the alphabet. For example, if the alphabet $\Sigma = \{a, b\}$, then *abab* and *aaabbba* are strings on Σ. With few exceptions, we will use lower case letters $a, b, c, \ldots$ for elements of Σ and letters $u, v, w, \ldots$ for string names. We will write, for example,

$$w = abaaa$$

to indicate that the string named w has the specific value *abaaa*.

The **concatenation** of two strings w and v is the string obtained by appending the symbols of v to the right end of w, that is, if

$$w = a_1 a_2 \cdots a_n$$

and

$$v = b_1 b_2 \cdots b_m,$$

then the concatenation of w and v, denoted by wv, is

$$wv = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m.$$

The **reverse** of a string is obtained by writing the symbols in reverse order; if w is a string as shown above, then its reverse w^R is

$$w^R = a_n \cdots a_2 a_1.$$

If

$$w = vu,$$

then v is said to be a **prefix** and u a **suffix** of w.

The **length** of a string w, denoted by $|w|$, is the number of symbols in the string. We will frequently need to refer to the **empty string** which is a string with no symbols at all. It will be denoted by λ. The following simple relations

$$|\lambda| = 0,$$
$$\lambda w = w\lambda = w,$$

hold for all w.

Simple properties of strings, such as their length, are very intuitive and probably need little elaboration. For example, if u and v are strings, then the length of their concatenation is the sum of the individual lengths, that is,

$$|uv| = |u| + |v|. \tag{1.6}$$

But although this relationship is obvious, it is useful to be able to make it precise and prove it. The techniques for doing so are important in more complicated situations.

▸ ***Example 1.7*** Show that (1.6) holds for any u and v. To prove this, we first need a definition of the length of a string. We make such a definition in a recursive fashion by

$$|a| = 1,$$
$$|wa| = |w| + 1,$$

for all $a \in \Sigma$ and w any string on Σ. This definition is a formal statement of our intuitive understanding of the length of a string: the length of a single symbol is one, and the length of any string is increased by one if we add another symbol to it. With this formal definition, we are ready to prove (1.6) by induction.

By definition, (1.6) holds for all u of any length and all v of length 1, so we have a basis. As an inductive assumption, we take that (1.6) holds for all u of any length and all v of length $1, 2, \ldots, n$. Now take any v of length $n + 1$ and write it as $v = wa$. Then,

$$|v| = |w| + 1,$$
$$|uv| = |uwa| = |uw| + 1.$$

But by the inductive hypothesis (which is applicable since w is of length n),

$$|uw| = |u| + |w|,$$

so that

$$|uv| = |u| + |w| + 1 = |u| + |v|.$$

Therefore, (1.6) holds for all u and all v of length up to $n + 1$, completing the inductive step and the argument.

If w is a string, then w^n is the string obtained by concatenating w with itself n times. As a special case, we define

$$w^0 = \lambda,$$

for all w.

If Σ is an alphabet, then we use Σ^* to denote the set of strings obtained by concatenating zero or more symbols from Σ. The set Σ^* always contains λ. To exclude the empty string, we define

$$\Sigma^+ = \Sigma^* - \{\lambda\}.$$

While Σ is finite by assumption, Σ^* and Σ^+ are always infinite since there is no limit on the length of the strings in these sets.

A language is defined very generally as a subset of Σ^*. Any string w in a language L will be called a **word** or a **sentence** of L. In formal language theory, we make no distinction between words and sentences and use the terms interchangeably. To provide more structure to this rather broad definition, we will later study methods by which individual languages can be defined. For the moment, though, we will just look at a few simple examples.

▸ ***Example 1.8*** Let $\Sigma = \{a, b\}$. Then

$$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}.$$

The set

$$\{a, aa, aab\}$$

is a language on Σ. Because it has a finite number of words, we call it a finite language. The set

$$L = \{a^n b^n : n \geq 0\}$$

is also a language on Σ. The strings *aabb* and *aaaabbbb* are words in the language L, but the string *abb* is not in L. This language is infinite.

Since languages are sets, the union, intersection, and difference of two languages are immediately defined. The complement of a language is defined with respect to Σ^*; that is, the complement of L is

$$\bar{L} = \Sigma^* - L.$$

The concatenation of two languages L_1 and L_2 is the set of all strings obtained by concatenating any element of L_1 with any element of L_2; specifically,

$$L_1 L_2 = \{xy : x \in L_1, y \in L_2\}.$$

We define L^n as L concatenated with itself n times, with the special case

$$L^0 = \{\lambda\}$$

for every language L.

▸ ***Example 1.9*** If

$$L = \{a^n b^n : n \geq 0\},$$

then

$$L^2 = \{a^n b^n a^m b^m : n \geq 0, m \geq 0\}.$$

Note that n and m in the above are unrelated; the string *aabbaaabbb* is in L^2.

Finally, we define the **star-closure** of a language as

$$L^* = L^0 \cup L^1 \cup L^2 \cdots$$

and the **positive closure** as

$$L^+ = L^1 \cup L^2 \cdots$$

Grammars

To study languages mathematically, we need a mechanism to describe them. Everyday language is imprecise and ambiguous, so informal descriptions in English are often inadequate. The set notation used in Examples 1.8 and 1.9 is more suitable, but limited. As we proceed we will learn about several language-definition mechanisms that are useful in different circumstances. Here we introduce a common and powerful one, the notion of a **grammar**.

A grammar for the English language tells us whether a particular sentence is well formed or not. A typical rule of English grammar is "a sentence can consist of a noun phrase followed by a predicate." More concisely we write this as

$$\langle sentence \rangle \rightarrow \langle noun\_phrase \rangle \langle predicate \rangle,$$

with the obvious interpretation. This is of course not enough to deal with actual sentences. We must now provide definitions for the newly introduced constructs $\langle noun\_phrase \rangle$ and $\langle predicate \rangle$. If we do so by

$$\langle noun\_phrase \rangle \rightarrow \langle article \rangle \langle noun \rangle,$$
$$\langle predicate \rangle \rightarrow \langle verb \rangle,$$

and if we associate the actual words "a" and "the" with $\langle article \rangle$, "boy" and "dog" with $\langle noun \rangle$, and "runs" and "walks" with $\langle verb \rangle$, then the grammar tells us that the sentences "a boy runs" and "the dog walks" are properly formed. If we were to give a complete grammar, then in theory, every proper sentence could be explained this way.

This example illustrates the definition of a general concept in terms of simpler ones. We start with the top level concept, here ⟨*sentence*⟩, and successively reduce it to the irreducible building blocks of the language. The generalization of these ideas leads us to formal grammars.

Definition 1.1

A grammar G is defined as a quadruple

$$G = (V, T, S, P)$$

where

V is a finite set of objects called **variables**,

T is a finite set of objects called **terminal symbols**,

$S \in V$ is a special symbol called the **start** variable,

P is a finite set of **productions**.

It will be assumed without further mention that the sets V and T are nonempty and disjoint.

The production rules are the heart of a grammar; they specify how the grammar transforms one string into another, and through this they define a language associated with the grammar. In our discussion we will assume that all production rules are of the form

$$x \rightarrow y,$$

where x is an element of $(V \cup T)^+$ and y is in $(V \cup T)^*$. The productions are applied in the following manner: given a string w of the form

$$w = uxv,$$

we say the production $x \rightarrow y$ is applicable to this string, and we may use it to replace x with y, thereby obtaining a new string

$$z = uyv.$$

This is written as

$$w \Rightarrow z.$$

We say that w **derives** z or that z is derived from w. Successive strings are derived by applying the productions of the grammar in arbitrary order. A production can be used whenever it is applicable, and it can be applied as often as desired. If

$$w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n,$$

we say that w_1 derives w_n and write

$$w_1 \overset{*}{\Rightarrow} w_n.$$

The * indicates that an unspecified number of steps (including zero) can be taken to derive w_n from w_1. Thus

$$w \overset{*}{\Rightarrow} w$$

is always the case. To indicate that at least one production must be applied, we write

$$w \overset{+}{\Rightarrow} v.$$

By applying the production rules in a different order, a given grammar can normally generate many strings. The set of all such strings is the language defined or generated by the grammar.

Definition 1.2

Let $G = (V, T, S, P)$ be a grammar. Then the set

$$L(G) = \{w \in T^*: S \overset{*}{\Rightarrow} w\}$$

is the language generated by G.

If $w \in L(G)$, then the sequence

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n \Rightarrow w$$

is a **derivation** of the sentence (or word) w. The strings $S, w_1, w_2, \ldots, w_n$, which contain variables as well as terminals, are called **sentential forms** of the derivation.

▸ ***Example 1.10*** Consider the grammar

$$G = (\{S\}, \{a, b\}, S, P),$$

with P given by

$$S \to aSb,$$
$$S \to \lambda.$$

Then

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb,$$

so we can write

$$S \overset{*}{\Rightarrow} aabb.$$

The string $aabb$ is a sentence in the language generated by G, while $aaSbb$ is a sentential form.

A grammar G completely defines $L(G)$, but it may not be easy to get a very explicit description of the language from the grammar. Here, however, the answer is fairly clear. It is not hard to conjecture that

$$L(G) = \{a^n b^n : n \geq 0\},$$

and it is easy to prove it. If we notice that the rule $S \to aSb$ is recursive, a proof by induction readily suggests itself. We first show that all sentential forms must have the form

$$w_i = a^i S b^i. \tag{1.7}$$

Suppose that (1.7) holds for all sentential forms w_i of length $2i + 1$ or less. To get another sentential form (which is not a sentence), we can only apply the production $S \to aSb$. This gets us

$$a^i S b^i \Rightarrow a^{i+1} S b^{i+1},$$

so that every sentential form of length $2i + 3$ is also of the form (1.7). Since (1.7) is obviously true for $i = 1$, it holds by induction for all i. Finally, to get a sentence we must apply the production $S \to \lambda$, and we see that

$$S \overset{*}{\Rightarrow} a^n S b^n \Rightarrow a^n b^n$$

represents all possible derivations. Thus G can derive only strings of the form $a^n b^n$.

We also have to show that all strings of this form can be derived. This is easy; we simply apply $S \to aSb$ as many times as needed, followed by $S \to \lambda$.

▸ ***Example 1.11*** Find a grammar that generates

$$L = \{a^n b^{n+1} : n \geq 0\}.$$

The idea behind the previous example can be extended to this case. All we need to do is generate an extra b. This can be done with a production $S \to Ab$, with other productions chosen so that A can derive the language in the previous example. Reasoning in this fashion, we get the grammar $G = (\{S, A\}, \{a, b\}, S, P)$, with productions

$$\begin{aligned} S &\to Ab, \\ A &\to aAb, \\ A &\to \lambda. \end{aligned}$$

Derive a few specific sentences to convince yourself that this works.

The above examples are fairly easy ones, so rigorous arguments may seem superfluous. But often it is not so easy to find a grammar for a language described in an informal way or to give an intuitive characterization of the language defined by a grammar. To show that a given language is indeed generated by a certain grammar G, we must be able to show (a) that every $w \in L$ can be derived from S using G, and (b) that every string so derived is in L.

▶ ***Example 1.12*** Take $\Sigma = \{a, b\}$, and let $n_a(w)$ and $n_b(w)$ denote the number of a's and b's in the string w, respectively. Then the grammar G with productions

$$S \to SS,$$
$$S \to \lambda,$$
$$S \to aSb,$$
$$S \to bSa,$$

generates the language

$$L = \{w: n_a(w) = n_b(w)\}.$$

This claim is not so obvious, and we need to provide convincing arguments.

First, it is clear that every sentential form of G has an equal number of a's and b's, since the only productions that generate an a, namely $S \to aSb$ and $S \to bSa$ simultaneously generate a b. Therefore every element of $L(G)$ is in L. It is a little harder to see that every string in L can be derived with G.

Let us begin by looking at the problem in outline, considering the various forms $w \in L$ can have. Suppose w starts with an a and ends with a b. Then it has the form

$$w = aw_1b,$$

where w_1 is also in L. We can think of this case as being derived starting with

$$S \Rightarrow aSb,$$

if S does indeed derive any string in L. A similar argument can be made if w starts with a b and ends with an a. But this does not exhaust all cases. A string in L may start and end with the same symbol. In this case, an argument based on counting a's and b's beginning at the left of the string will give us the answer. Suppose we count $+1$ for an a and -1 for a b. If a string w starts and ends with a, then the count will be $+1$ after the leftmost symbol and -1 immediately before the rightmost one. Therefore, the count has to go through zero somewhere in the middle of the string, indicating

that such a string must have the form

$$w = w_1 w_2,$$

where both w_1 and w_2 are in L. This case can be taken care of by the production $S \to SS$.

Once we see the argument intuitively, we are ready to proceed more rigorously. Again we use induction. Assume that all $w \in L$ with $|w| \leq 2n$ can be derived with G. Take any $w \in L$ of length $2n + 2$. If $w = aw_1b$, then w_1 is in L, and $|w_1| = 2n$. Therefore, by assumption,

$$S \overset{*}{\Rightarrow} w_1.$$

Then

$$S \Rightarrow aSb \overset{*}{\Rightarrow} aw_1b = w$$

is possible, and w can be derived with G. Obviously, similar arguments can be made if $w = bw_1a$.

If w is not of this form, that is, if it starts and ends with the same symbol, then the counting argument tells us that it must have the form $w = w_1w_2$, with w_1 and w_2 both in L and of length less than or equal to $2n$. Hence again we see that

$$S \Rightarrow SS \overset{*}{\Rightarrow} w_1S \overset{*}{\Rightarrow} w_1w_2 = w$$

is possible.

Since the inductive assumption is clearly satisfied for $n = 1$, we have a basis, and the claim is true for all n, completing our argument.

We can define an equivalence relation for grammars by saying that two grammars are equivalent if they generate the same language. As we will see later on, it is not always easy to see whether or not two grammars are equivalent.

▸ ***Example 1.13*** Consider the grammar $G_1 = (\{A, S\}, \{a, b\}, S, P_1)$, with P_1 consisting of the productions

$$S \to aAb|\lambda,$$
$$A \to aAb|\lambda.$$

Here we introduce a convenient shorthand notation in which several production rules with the same left-hand side are written on a single line, with alternative right-hand sides separated by |. In this notation, $S \rightarrow aAb|\lambda$ stands for the two productions $S \rightarrow aAb$ and $S \rightarrow \lambda$.

This grammar is equivalent to the grammar G in Example 1.10. The equivalence is easy to prove by showing that

$$L(G_1) = \{a^n b^n : n \geq 0\}.$$

We leave this as an exercise.

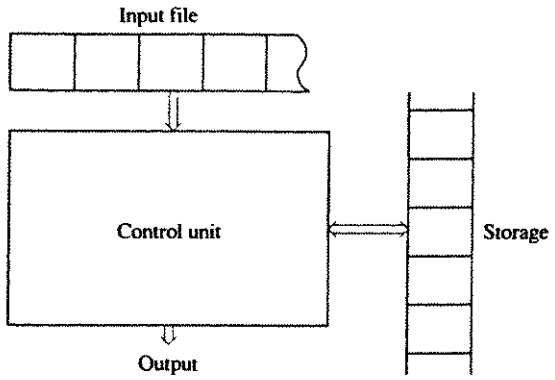
Automata

An automaton is an abstract model of a digital computer. As such, every automaton includes some essential features. It has a mechanism for reading input. It will be assumed that the input is a string over a given alphabet, written on an **input file**, which the automaton can read but not change. The input file is divided into squares, each of which can hold one symbol. The input mechanism can read the input file left to right, one symbol at a time. The input mechanism can also detect the end of the input string (by sensing an end-of-file condition). The automaton can produce output of some form. It may have a temporary **storage** device, consisting of an unlimited number of cells, each capable of holding a single symbol from an alphabet (not necessarily the same one as the input alphabet). The automaton can read and change the contents of the storage cells. Finally, the automaton has a **control unit**, which can be in any one of a finite number of **internal states**, and which can change state in some defined manner. Figure 1.3 (p. 26) shows a schematic representation of a general automaton.

An automaton is assumed to operate in a discrete time frame. At any given time, the control unit is in some internal state, and the input mechanism is scanning a particular symbol on the input file. The internal state of the control unit at the next time step is determined by the **next-state** or **transition function.** This transition function gives the next state in terms of the current state, the current input symbol, and the information currently in the temporary storage. During the transition from one time interval to the next, output may be produced or the information in the temporary storage changed. The term **configuration** will be used to refer to a particular state of the control unit, input file, and temporary storage. The transition of the automaton from one configuration to the next will be called a **move.**

This general model covers all the automata we will discuss in this book.

Figure 1.3



A finite-state control will be common to all specific cases, but differences will arise from the way in which the output can be produced and the nature of the temporary storage. The nature of the temporary storage has the stronger effect on particular types of automata.

For subsequent discussions, it will be necessary to distinguish between **deterministic automata** and **nondeterministic automata**. A deterministic automaton is one in which each move is uniquely determined by the current configuration. If we know the internal state, the input, and the contents of the temporary storage, we can predict the future behavior of the automaton exactly. In a nondeterministic automaton, this is not so. At each point, a nondeterministic automaton may have several possible moves, so we can only predict a set of possible actions. The relation between deterministic and nondeterministic automata of various types will play a significant role in our study.

An automaton whose output response is limited to a simple "yes" or "no" is called an **accepter**. Presented with an input string, an accepter either accepts (recognizes) the string or rejects it. A more general automaton, capable of producing strings of symbols as output, is called a **transducer**. Although we will give some simple examples of transducers in the next section, our primary interest in this book is in accepters.

▸ *EXERCISES*

1. Show that $|u^n| = n\,|u|$ for all strings u and all n.

2. The reverse of a string, introduced informally above, can be defined more precisely by the recursive rules

$$a^R = a,$$
$$(wa)^R = aw^R,$$

for all $a \in \Sigma$, $w \in \Sigma^*$. Use this to prove that

$$(uv)^R = v^R u^R,$$

for all $u, v \in \Sigma^+$.

3. Prove that $(w^R)^R = w$ for all $w \in \Sigma^*$.
4. Let $L = \{ab, aa, baa\}$. Which of the following strings are in L^*: *abaabaaabaa*, *aaaabaaaa*, *baaaaabaaaab*, *baaaaabaa*?
5. Consider the languages in Examples 1.11 and 1.12. For which is it true that $L = L^*$?
6. Are there languages for which $\overline{L^*} = \overline{L}^*$?
7. Find grammars for $\Sigma = \{a, b\}$ that generate the sets of
   (a) all strings with exactly one a,
   (b) all strings with at least one a,
   (c) all strings with no more than three a's.
   In each case, give convincing arguments that the grammar you give does indeed generate the indicated language.
8. Give a simple description of the language generated by the grammar with productions

$$S \to aA,$$
$$A \to bS,$$
$$S \to \lambda.$$

9. What language does the grammar with these productions generate?

$$S \to Aa,$$
$$A \to B,$$
$$B \to Aa.$$

10. For each of the following languages, find a grammar that generates it.
    (a) $L_1 = \{a^n b^m : n \geq 0, m > n\}$
    (b) $L_2 = \{a^n b^{2n} : n \geq 0\}$

(c) $L_3 = \{a^{n+2}b^n: n \geq 1\}$
(d) $L_4 = \{a^n b^{n-3}: n \geq 3\}$
(e) $L_1 L_2$
(f) $L_1 \cup L_2$
(g) L_1^3
(h) L_1^*
(i) $L_1 - \overline{L_4}$

11. Find grammars for the following languages on $\Sigma = \{a\}$.
(a) $L = \{w: |w| \bmod 3 = 0\}$
(b) $L = \{w: |w| \bmod 3 > 0\}$
(c) $L = \{w: |w| \bmod 3 \neq |w| \bmod 2\}$
(d) $L = \{w: |w| \bmod 3 \geq |w| \bmod 2\}$

12. Find a grammar that generates the language

$$L = \{ww^R: w \in \{a, b\}^+\}.$$

Give a complete justification for your answer.

13. Using the notation of Example 1.12, find grammars for the languages below.
(a) $L = \{w: n_a(w) = n_b(w) + 1\}$
(b) $L = \{w: n_a(w) > n_b(w)\}$
(c) $L = \{w: n_a(w) = 2n_b(w)\}$
Give convincing arguments for your answers.

14. Complete the arguments in Example 1.13, showing that $L(G_1)$ does in fact generate the given language.

15. Are the two grammars with respective productions

$$S \to aSb|ab|\lambda,$$

and

$$S \to aAb|ab,$$
$$A \to aAb|\lambda,$$

equivalent? Assume that S is the start symbol in both cases.

16. Show that the grammar $G = (\{S\}, \{a, b\}, S, P)$, with productions

$$S \to SS|SSS|aSb|bSa|\lambda,$$

is equivalent to the grammar in Example 1.12.

17. So far we have given examples of only relatively simple grammars; every production had a single variable on the left side. As we will see, such grammars are very important, but Definition 1.1 allows more general forms.

    Consider the grammar $G = (\{A, B, C, D, E, S\}, \{a\}, S, P)$, with productions

$$\begin{aligned} S &\rightarrow ABaC, \\ Ba &\rightarrow aaB, \\ BC &\rightarrow DC|E, \\ aD &\rightarrow Da, \\ AD &\rightarrow AB, \\ aE &\rightarrow Ea, \\ AE &\rightarrow \lambda. \end{aligned}$$

    Derive three different sentences in $L(G)$. From these, make a conjecture about $L(G)$.

▸ *1.3

SOME APPLICATIONS

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this section, we present some obvious examples to give the reader some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at

*As explained in the Preface, an asterisk preceding a heading indicates optional material.

almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammar that describes a typical language like Pascal is very extensive. For an example, let us take a smaller language that is part of this larger one.

▸ ***Example 1.14*** The set of all legal identifiers in Pascal is a language. Informally, it is the set of all strings starting with a letter and followed by an arbitrary number of letters or digits. The grammar below makes this informal definition precise.

$$\begin{aligned}\langle id\rangle &\to \langle letter\rangle\langle rest\rangle,\\ \langle rest\rangle &\to \langle letter\rangle\langle rest\rangle|\langle digit\rangle\langle rest\rangle|\lambda,\\ \langle letter\rangle &\to a|b|\ldots,\\ \langle digit\rangle &\to 0|1|\ldots\end{aligned}$$

In this grammar, the variables are $\langle id\rangle$, $\langle letter\rangle$, $\langle digit\rangle$, and $\langle rest\rangle$, with a, b, . . . , 0, 1, . . . the terminals. A derivation of the identifier $a0$ is

$$\begin{aligned}\langle id\rangle &\Rightarrow \langle letter\rangle\langle rest\rangle\\ &\Rightarrow a\ \langle rest\rangle\\ &\Rightarrow a\ \langle digit\rangle\langle rest\rangle\\ &\Rightarrow a0\ \langle rest\rangle\\ &\Rightarrow a0.\end{aligned}$$

The definition of programming languages through grammars is common and very useful. But there are alternatives which are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We will do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, Figure 1.4 represents a transition from state 1 to state 2, which is taken when the input symbol is a. With this intuitive picture in mind, let us look at another way of describing Pascal identifiers.

▸ ***Example 1.15*** Figure 1.5 is an automaton which accepts all legal Pascal identifiers. Some interpretation is necessary. We assume that initially the automaton is in state 1; we indicate this by drawing an arrow (not originating in any

Figure 1.4

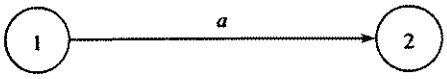
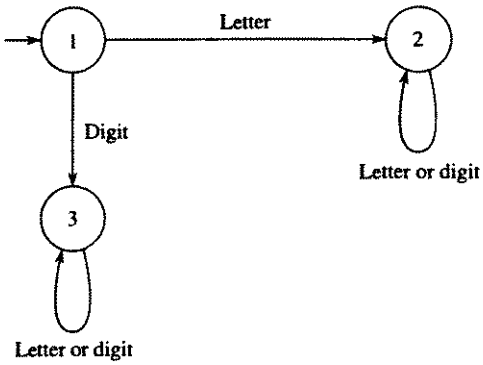


Figure 1.5



vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter, the automaton goes into state 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into state 3, the "no" state, and remain there. In our solution we assume that no input other than letters or digits is possible.

Compilers and other translaters that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.14, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint how this is done; subsequent examples will expand on this observation.

Another important application area is digital design, where transducer concepts are prevalent. Although this is a subject which we will not treat extensively here, we will give a simple example. In principle, any digital computer can be viewed as an automaton, but such a view is not necessarily

appropriate. Suppose we consider the internal registers and main memory of a computer as the automaton's control unit. Then the automaton has a total of 2^n internal states, where n is the total number of bits in the registers and memory. Even for small n, this is such a large number that the result is impossible to work with. But if we look at a much smaller unit, then automata theory becomes a useful design tool.

▸ ***Example 1.16*** A binary adder is an integral part of any general purpose computer. Such an adder takes two bit strings representing numbers, and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

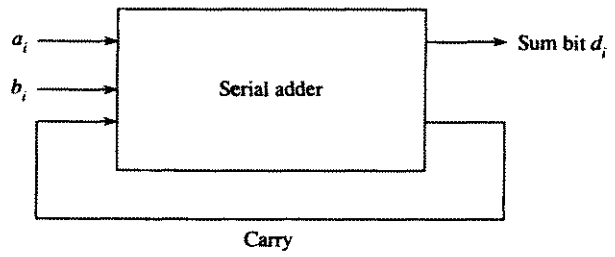
A serial adder processes two such numbers $x = a_0 a_1, \ldots,$ and $y = b_0 b_1, \ldots,$ bit by bit, starting at the left end. Each bit addition creates a digit for the sum as well as a carry digit for the next higher position. A binary addition table (Figure 1.6) summarizes the process.

A block diagram of the kind we saw when we first studied computers is given in Figure 1.7. It tells us that an adder is a box that accepts two bits

Figure 1.6

a_i \ b_i	0	1
0	0 No carry	1 No carry
1	1 No carry	0 Carry

Figure 1.7

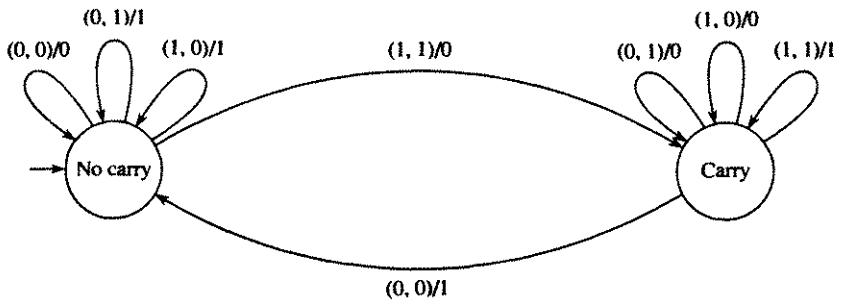


and produces their sum bit and a possible carry. It describes what an adder does. but explains little about its internal workings. An automaton (now a transducer) can make this much more explicit.

The input to the transducer are the bit pairs (a_i, b_i), the output will be the sum bit d_i. Again, we represent the automaton by a graph, now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry." Initially, the transducer will be in state "no carry." It will remain in this state until a bit pair (1, 1) is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in Figure 1.8. Follow this through with a few examples to convince yourself that it works correctly.

As this example indicates, the automaton serves as a bridge between the very high-level, functional description of a circuit and its logical implementation through transistors, gates, and flip-flops. The automaton clearly

Figure 1.8



shows the decision logic, yet it is formal enough to lend itself to precise mathematical manipulation. For this reason, digital design methods rely heavily on concepts from automata theory. The interested reader should look at a typical text on this topic, for example Kovahi 1978.

▸ *EXERCISES*

1. Give a grammar for the set of Pascal integer numbers.
2. Design an accepter for Pascal integers.
3. Give a grammar that generates all Pascal reals.
4. Suppose that a certain programming language permits only identifiers that begin with a letter, contain at least one but no more than three digits, and can have any number of letters. Give a grammar and an accepter for such a set of identifiers.
5. Give a grammar for Pascal var declarations.
6. In the roman number system, numbers are represented by strings on the alphabet $\{M, D, C, L, X, V, I\}$. Design an accepter that recognizes such strings only if they are properly formed roman numbers. For simplicity, replace the "subtraction" convention in which the number nine is represented by *IX* with an addition equivalent that uses *VIIII* instead.
7. We assumed that an automaton works in a framework of discrete time steps, but this aspect has little influence on our subsequent discussion. In digital design, however, the time element assumes considerable significance.

   In order to synchronize signals arriving from different parts of the computer, delay circuitry is needed. A unit-delay transducer is one which simply reproduces the input (viewed as a continual stream of symbols) one time unit later. Specifically, if the transducer reads as input a symbol a at time t, it will reproduce that symbol as output at time $t + 1$. At time $t = 0$, the transducer outputs nothing. We indicate this by saying that the transducer translates input $a_1a_2 \cdots$ into output $\lambda a_1a_2 \cdots$.

   Draw a graph showing how such a unit-delay transducer might be designed for $\Sigma = \{a, b\}$.
8. An n-unit delay transducer is one that reproduces the input n time units later; that is, the input $a_1a_2 \cdots$ is translated into $\lambda^n a_1a_2 \ldots$, meaning again that the transducer produces no output for the first n time slots.

(a) Construct a two-unit delay transducer on $\Sigma = \{a, b\}$.
(b) Show that an n-unit delay transducer must have at least $|\Sigma|^n$ states.

9. The two's complement of a binary string, representing a positive integer, is formed by first complementing each bit, then adding one to the lowest-order bit. Design a transducer for translating bit strings into their two's complement, assuming that the binary number is represented as in Example 1.16, with lower-order bits at the left of the string.

10. Let $a_1a_2 \cdots$ be an input bit string. Design a transducer that computes the parity of every substring of three bits. Specifically, the transducer should produce output

$$\pi_1 = \pi_2 = 0,$$
$$\pi_i = (a_{i-2} + a_{i-1} + a_i) \bmod 2, \; i = 3, 4, \ldots$$

For example, the input 110111 should produce output 000001.

11. Digital computers normally represent all information by bit strings, using some type of encoding. For example, character information can be encoded using the well-known ASCII system.

For this exercise, consider the two alphabets $\{a, b, c, d\}$ and $\{0, 1\}$, respectively, and an encoding from the first to the second, defined by $a \rightarrow 00$, $b \rightarrow 01$, $c \rightarrow 10$, $d \rightarrow 11$. Construct a transducer for decoding strings on $\{0, 1\}$ into the original message. For example, the input 010011 should generate as output *bad*.

12. Let x and y be two positive binary numbers, represented as in Example 1.16. Design a transducer whose output is $max(x, y)$.