The starting statements $P_1, P_2, \ldots P_k$ are called the **basis** of the induction. The step connecting P_n with P_{n+1} is called the **inductive step.** The inductive step is generally made easier by the **inductive assumption** that $P_1, P_2, \ldots, P_n$ are true. In a formal inductive argument, we show all three parts explicitly.

Example 1.5

A binary tree is a tree in which no parent can have more than two children. 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 leaf.

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 1.6

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 1.5.

Inductive reasoning can be difficult to grasp. It helps to notice the close connection between induction and recursion in programming. For example, the recursive definition of a function $f(n)$, where n is any positive integer, often has two parts. One involves the definition of $f(n+1)$ in terms of $f(n), f(n-1), ..., f(1)$. This corresponds to the inductive step. The second part is the "escape" from the recursion, which is accomplished by defining $f(1), f(2), ...f(k)$ nonrecursively. This corresponds to the basis of induction. As in induction, recursion allows us to draw conclusions about all instances of the problem, given only a few starting values and using the recursive nature of the problem.

Proof by contradiction is another powerful technique that often works when everything else fails. Suppose we want to prove that some statement P is true. We then assume, for the moment, that P is false and see where that assumption leads us. If we arrive at a conclusion that we know is incorrect, we can lay the blame on the starting assumption and conclude that P must be true. The following is a classic and elegant example.

Example 1.7 A rational number is a number that can be expressed as the ration of two integers n and m so that n and m have no common factor. A real number that is not rational is said to be irrational. Show that $\sqrt{2}$ is irrational.

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 on the size of S to show that if S is a finite set then $\left|2^S\right| = 2^{|S|}$.

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, show that $|S_1 \times S_2| = |S_1|\,|S_2|$.

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

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

9. Show that

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

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. Give conditions on S_1 and S_2 necessary and sufficient to ensure that

$$S_1 = (S_1 \cup S_2) - S_2.$$

13. Show that if $f(n) = O(g(n))$ and $g(n) = O(f(n))$, then $f(n) = \Theta(g(n))$.

14. Show that $2^n = O(3^n)$ but $2^n \neq \Theta(3^n)$.

15. Show that the following order-of-magnitude results hold.

    (a) $n^2 + 5 \log n = O(n^2)$

    (b) $3^n = O(n!)$

    (c) $n! = O(n^n)$

16. Prove that if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

17. Show that if $f(n) = O(n^2)$ and $g(n) = O(n^3)$, then

$$f(n) + g(n) = O(n^3)$$

and

$$f(n)\, g(n) = O(n^6).$$

18. In Exercise 17, is it true that $g(n)/f(n) = O(n)$?

19. Assume that $f(n) = 2n^2 + n$ and $g(n) = O(n^2)$. What is wrong with the following argument?

$$f(n) = O(n^2) + O(n),$$

so that

$$f(n) - g(n) = O(n^2) + O(n) - O(n^2).$$

Therefore,

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

20. 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.

21. 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 be a path of length no larger than $|V| - 1$ between these two vertices.

22. 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.

23. Show that

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

24. Show that

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

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

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

27. Show that $2 - \sqrt{2}$ is irrational.

28. Prove or disprove the following statements.

    (a) The sum of a rational and an irrational number must be irrational.

    (b) The sum of two positive irrational numbers must be irrational.

    (c) The product of a rational and an irrational number must be irrational.

29. Show that every positive integer can be expressed as the product of prime numbers.

★30. Prove that the set of all prime numbers is infinite.

31. A prime pair consists of two primes that differ by two. There are many prime pairs, for example, 11 and 13, 17 and 19, etc. Prime triplets are three numbers $n, n+2, n+4$ that are all prime. Show that the only prime triplets are $(1, 3, 5)$ and $(3, 5, 7)$.

1.2 Three Basic Concepts

Three fundamental ideas are the major themes of this book: **languages, grammars,** 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 of 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 $a, b, c, ...$ for elements of Σ and $u, v, w, ...$ 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

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

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.

Any string of consecutive characters in some w is said to be a **substring** of w. If

$$w = vu,$$

then the substrings v and u are said to be a **prefix** and a **suffix** of w, respectively. For example, if $w = abbab$, then $\{\lambda, a, ab, abb, abba, abbab\}$ is the set of all prefixes of w, while bab, ab, b are some of its suffixes.

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 situation.

Example 1.8 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 $1, 2, ..., n$. 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 stands for the string obtained by repeating w 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^*. A string in a language L will be called a **sentence** of L. This definition is quite broad; any set of strings on an alphabet Σ can be considered a language. Later we will study methods by which specific languages can be defined and described; this will enable us to give some structure to this rather broad concept. For the moment, though, we will just look at a few specific examples.

Example 1.9 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 sentences, 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 in the language L, but the string abb is not in L. This language is infinite. Most interesting languages are 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

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

The reverse of a language is the set of all string reversals, that is,

$$L^R = \{w^R : w \in L\}.$$

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,

$$L_1L_2 = \{xy : x \in L_1, y \in L_2\}.$$

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

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

and

$$L^1 = L$$

for every language L.

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

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

Example 1.10 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.

The reverse of L is easily described in set notations as

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

but it is considerably harder to describe $\overline{L}$ or L^* this way. A few tries will quickly convince you of the limitation of set notation for the specification of complicated languages. ■

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 1.9 and 1.10 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 simple ones. We start with the top level concept, here $\langle sentence \rangle$, 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

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

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

$$w \stackrel{*}{\Rightarrow} w$$

is always the case.

By applying the production rules in a different order, a given grammar can normally generate many strings. The set of all such terminal 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

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

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 w. The strings $S, w_1, w_2, ..., w_n$, which contain variables as well as terminals, are called **sentential forms** of the derivation.

Example 1.11 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

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

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 $\boxed{S \to \lambda}$ 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 1.12 Find a grammar that generates

$S \to Ab$
$A \to aAb \mid \lambda$

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

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

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

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 1.13 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 take care of all cases, since a string in L can begin and end with the same symbol. If we write down a string of this type, say, $aabbba$, we see that it can be considered as the concatenation of two shorter strings $aabb$ and ba, both of which are in L. Is this true in general? To show that this is indeed so, we can use the following argument: Suppose that, starting at the left end of the string, 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

$$w = w_1w_2,$$

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,

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

Then

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

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

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

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. ■

Normally, a given language has many grammars that generate it. Even though these grammars are different, they are equivalent in some sense. We say that two grammars G_1 and G_2 are **equivalent** if they generate the same language, that is, if

$$L(G_1) = L(G_2).$$

As we will see later, it is not always easy to see if two grammars are equivalent.

Example 1.14 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, $S \to aAb|\lambda$ stands for the two productions $S \to aAb$ and $S \to \lambda$.

This grammar is equivalent to the grammar G in Example 1.11. The equivalence is easy to prove by showing that

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

We leave this as an exercise. ■

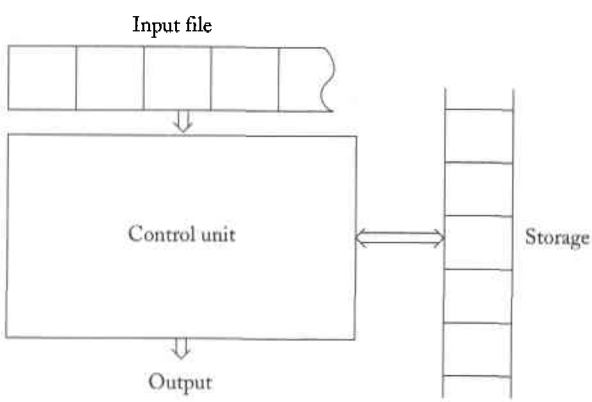
Automata

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 cells, each of which can hold one symbol. The input mechanism can read the input file from 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 shows a schematic representation of a general automation.

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. A finite-state control will be common to all specific cases, but differences

Figure 1.3



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 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. Use induction on n to 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 roles

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

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 $\left(w^R\right)^R = w$ 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 1.12 and 1.13. For which is it true that $L = L^*$?

6. Are there languages for which $\overline{L^*} = \overline{L}^*$?

7. Prove that

$$(L_1 L_2)^R = L_2^R L_1^R$$

for all languages L_1 and L_2.

8. Show that $(L^*)^* = L^*$ for all languages.

9. Prove or disprove the following claims.

(a) $(L_1 \cup L_2)^R = L_1^R \cup L_2^R$ for all languages L_1 and L_2.

(b) $(L^R)^* = (L^*)^R$ for all languages L.

10. 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.

(d) all strings with at least three a's.

In each case, give convincing arguments that the grammar you give does indeed generate the indicated language.

11. Give a simple description of the language generated by the grammar with productions

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

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

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

13. 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) $L_1 - \overline{L_4}$

★ **14.** 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) $L = \{w : |w| \bmod 3 \geq |w| \bmod 2\}$

15. Find a grammar that generates the language

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

Give a complete justification for your answer.

16. Using the notation of Example 1.13, find grammars for the languages below. Assume $\Sigma = \{a, b\}$.

(a) $L = \{w : n_a(w) = n_b(w) + 1\}$

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

★ (c) $L = \{w : n_a(w) = 2n_b(w)\}$

(d) $L = \{w \in \{a, b\}^* : |n_a(w) - n_b(w)| = 1\}$

17. Repeat Exercises 16(a) and 16(d) with $\Sigma = \{a, b, c\}$

18. Complete the arguments in Example 1.14, showing that $L(G_1)$ does in fact generate the given language.

19. 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.

20. 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 1.13.

★21. 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

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

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

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of this larger one.

Example 1.15 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.

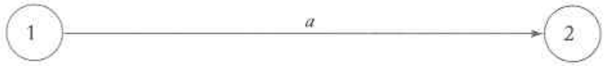
$$\begin{aligned}
\langle id \rangle &\to \langle letter \rangle \langle rest \rangle, \\
\langle rest \rangle &\to \langle letter \rangle \langle rest \rangle \,|\, \langle digit \rangle \langle rest \rangle \,|\, \lambda, \\
\langle letter \rangle &\to a \,|\, b \,|\, \cdots \,|\, z \\
\langle digit \rangle &\to 0 \,|\, 1 \,|\, \cdots \,|\, 9
\end{aligned}$$

In this grammar, the variables are $\langle id \rangle$, $\langle letter \rangle$, $\langle digit \rangle$, and $\langle rest \rangle$, and $a, b, ..., z, 0, 1, ..., 9$ the terminals. A derivation of the identifier $a0$ is

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

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

Figure 1.4

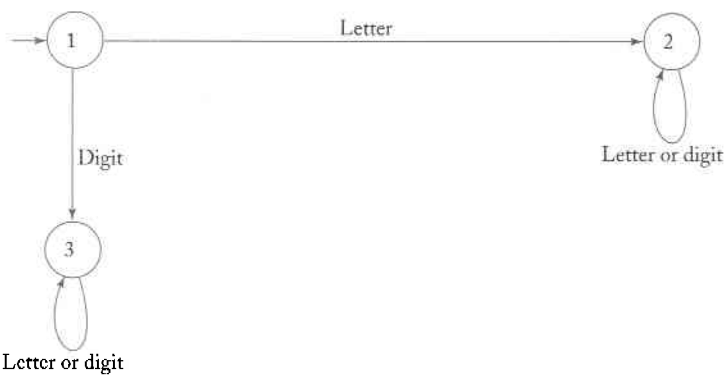


The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will 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 1.16 Figure 1.5 is an automaton that 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 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. ■

Figure 1.5



Compilers and other translators 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 1.15, 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 of 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 that 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 1.17 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 $x = a_0 a_1 \cdots a_n$, and $y = b_0 b_1 \cdots b_n$, 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.

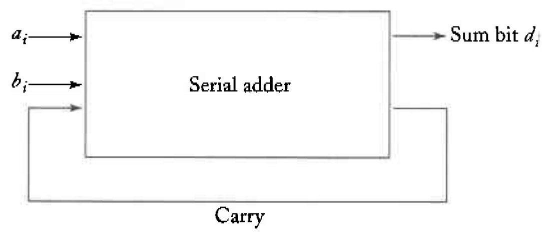
A block diagram of the kind we saw when we first studied computers is given in Figure 1.7. It tells us that an adder is a box that accepts two bits and produces their sum bit and a possible carry. It describes what an adder does, but explains little about its internal workings. An automaton (now a transducer) 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

Figure 1.6

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

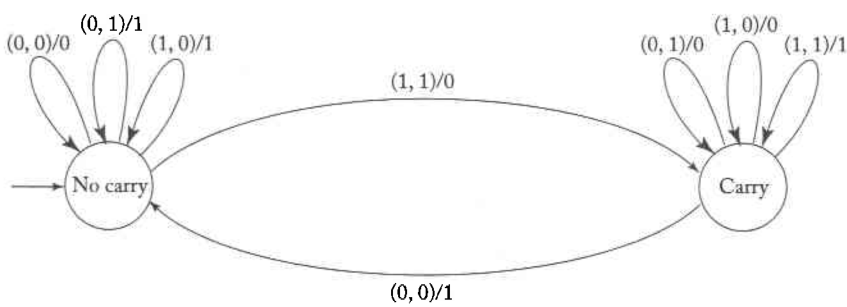
Figure 1.7



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 shows the decision logic, yet it is formal enough to lend itself to precise

Figure 1.8



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 integer numbers in C.
2. Design an accepter for integers in C.
3. Give a grammar that generates all reals in C.
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 the **var** declaration in Pascal.
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 accepts 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 that 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 $\lambda^n a_1a_2\cdots$, 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 1.17, with lower-order bits at the left of the string.

10. Design a transducer to convert a binary string into octal. For example, the bit string 001101110 should produce the output 156.

11. Let $a_1 a_2 \cdots$ 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 000001.

12. Design a transducer that accepts bit strings $a_1 a_2 a_3 \ldots$ and computes the binary value of each set of three consecutive bits modulo five. More specifically, the transducer should produce $m_1, m_2, m_3, \ldots$, where

$$m_1 = m_2 = 0,$$
$$m_i = (4a_i + 2a_{i-1} + a_{i-2}) \bmod 5, \;\; i = 3, 4, \ldots$$

13. 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 $a \to 00$, $b \to 01$, $c \to 10$, $d \to 11$. Construct a transducer for decoding strings on $\{0, 1\}$ into the original message. For example, the input 010011 should generate as output *bad*.

14. Let x and y be two positive binary numbers. Design a transducer whose output is $max(x, y)$.