0

INTRODUCTION

We begin with an overview of those areas in the theory of computation that we present in this course. Following that, you'll have a chance to learn and/or review some mathematical concepts that you will need later.

0.1

AUTOMATA, COMPUTABILITY, AND COMPLEXITY

This book focuses on three traditionally central areas of the theory of computation: automata, computability, and complexity. They are linked by the question:

What are the fundamental capabilities and limitations of computers?

This question goes back to the 1930s when mathematical logicians first began to explore the meaning of computation. Technological advances since that time have greatly increased our ability to compute and have brought this question out of the realm of theory into the world of practical concern.

In each of the three areas—automata, computability, and complexity—this question is interpreted differently, and the answers vary according to the interpretation. Following this introductory chapter, we explore each area in a

separate part of this book. Here, we introduce these parts in reverse order because by starting from the end you can better understand the reason for the beginning.

COMPLEXITY THEORY

Computer problems come in different varieties; some are easy, and some are hard. For example, the sorting problem is an easy one. Say that you need to arrange a list of numbers in ascending order. Even a small computer can sort a million numbers rather quickly. Compare that to a scheduling problem. Say that you must find a schedule of classes for the entire university to satisfy some reasonable constraints, such as that no two classes take place in the same room at the same time. The scheduling problem seems to be much harder than the sorting problem. If you have just a thousand classes, finding the best schedule may require centuries, even with a supercomputer.

What makes some problems computationally hard and others easy?

This is the central question of complexity theory. Remarkably, we don't know the answer to it, though it has been intensively researched for over 40 years. Later, we explore this fascinating question and some of its ramifications.

In one important achievement of complexity theory thus far, researchers have discovered an elegant scheme for classifying problems according to their computational difficulty. It is analogous to the periodic table for classifying elements according to their chemical properties. Using this scheme, we can demonstrate a method for giving evidence that certain problems are computationally hard, even if we are unable to prove that they are.

You have several options when you confront a problem that appears to be computationally hard. First, by understanding which aspect of the problem is at the root of the difficulty, you may be able to alter it so that the problem is more easily solvable. Second, you may be able to settle for less than a perfect solution to the problem. In certain cases, finding solutions that only approximate the perfect one is relatively easy. Third, some problems are hard only in the worst case situation, but easy most of the time. Depending on the application, you may be satisfied with a procedure that occasionally is slow but usually runs quickly. Finally, you may consider alternative types of computation, such as randomized computation, that can speed up certain tasks.

One applied area that has been affected directly by complexity theory is the ancient field of cryptography. In most fields, an easy computational problem is preferable to a hard one because easy ones are cheaper to solve. Cryptography is unusual because it specifically requires computational problems that are hard, rather than easy. Secret codes should be hard to break without the secret key or password. Complexity theory has pointed cryptographers in the direction of computationally hard problems around which they have designed revolutionary new codes.

COMPUTABILITY THEORY

During the first half of the twentieth century, mathematicians such as Kurt Gödel, Alan Turing, and Alonzo Church discovered that certain basic problems cannot be solved by computers. One example of this phenomenon is the problem of determining whether a mathematical statement is true or false. This task is the bread and butter of mathematicians. It seems like a natural for solution by computer because it lies strictly within the realm of mathematics. But no computer algorithm can perform this task.

Among the consequences of this profound result was the development of ideas concerning theoretical models of computers that eventually would help lead to the construction of actual computers.

The theories of computability and complexity are closely related. In complexity theory, the objective is to classify problems as easy ones and hard ones; whereas in computability theory, the classification of problems is by those that are solvable and those that are not. Computability theory introduces several of the concepts used in complexity theory.

AUTOMATA THEORY

Automata theory deals with the definitions and properties of mathematical models of computation. These models play a role in several applied areas of computer science. One model, called the *finite automaton*, is used in text processing, compilers, and hardware design. Another model, called the *context-free grammar*, is used in programming languages and artificial intelligence.

Automata theory is an excellent place to begin the study of the theory of computation. The theories of computability and complexity require a precise definition of a *computer*. Automata theory allows practice with formal definitions of computation as it introduces concepts relevant to other nontheoretical areas of computer science.

0.2 MATHEMATICAL NOTIONS AND TERMINOLOGY

As in any mathematical subject, we begin with a discussion of the basic mathematical objects, tools, and notation that we expect to use.

SETS

A ***set*** is a group of objects represented as a unit. Sets may contain any type of object, including numbers, symbols, and even other sets. The objects in a set are called its ***elements*** or ***members***. Sets may be described formally in several ways.

One way is by listing a set's elements inside braces. Thus the set

$$S = \{7, 21, 57\}$$

contains the elements 7, 21, and 57. The symbols $\in$ and $\notin$ denote set membership and nonmembership. We write $7 \in \{7, 21, 57\}$ and $8 \notin \{7, 21, 57\}$. For two sets A and B, we say that A is a ***subset*** of B, written $A \subseteq B$, if every member of A also is a member of B. We say that A is a ***proper subset*** of B, written $A \subsetneq B$, if A is a subset of B and not equal to B.

The order of describing a set doesn't matter, nor does repetition of its members. We get the same set S by writing $\{57, 7, 7, 7, 21\}$. If we do want to take the number of occurrences of members into account, we call the group a ***multiset*** instead of a set. Thus $\{7\}$ and $\{7, 7\}$ are different as multisets but identical as sets. An ***infinite set*** contains infinitely many elements. We cannot write a list of all the elements of an infinite set, so we sometimes use the "..." notation to mean "continue the sequence forever." Thus we write the set of ***natural numbers*** $\mathcal{N}$ as

$$\{1, 2, 3, \ldots\}.$$

The set of ***integers*** $\mathcal{Z}$ is written as

$$\{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

The set with zero members is called the ***empty set*** and is written $\emptyset$. A set with one member is sometimes called a ***singleton set***, and a set with two members is called an ***unordered pair***.

When we want to describe a set containing elements according to some rule, we write $\{n |$ rule about $n\}$. Thus $\{n | \, n = m^2$ for some $m \in \mathcal{N}\}$ means the set of perfect squares.

If we have two sets A and B, the ***union*** of A and B, written $A \cup B$, is the set we get by combining all the elements in A and B into a single set. The ***intersection*** of A and B, written $A \cap B$, is the set of elements that are in both A and B. The ***complement*** of A, written $\overline{A}$, is the set of all elements under consideration that are *not* in A.

As is often the case in mathematics, a picture helps clarify a concept. For sets, we use a type of picture called a ***Venn diagram***. It represents sets as regions enclosed by circular lines. Let the set START-t be the set of all English words that start with the letter "t". For example, in the figure, the circle represents the set START-t. Several members of this set are represented as points inside the circle.

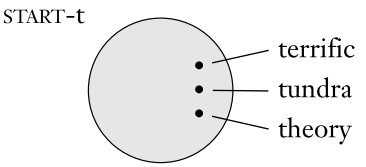


FIGURE 0.1
Venn diagram for the set of English words starting with "t"

Similarly, we represent the set END-z of English words that end with "z" in the following figure.

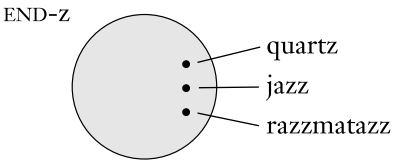


FIGURE 0.2
Venn diagram for the set of English words ending with "z"

To represent both sets in the same Venn diagram, we must draw them so that they overlap, indicating that they share some elements, as shown in the following figure. For example, the word *topaz* is in both sets. The figure also contains a circle for the set START-j. It doesn't overlap the circle for START-t because no word lies in both sets.

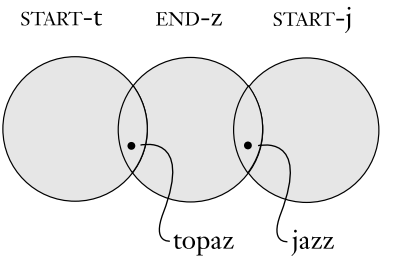


FIGURE 0.3
Overlapping circles indicate common elements

The next two Venn diagrams depict the union and intersection of sets A and B.

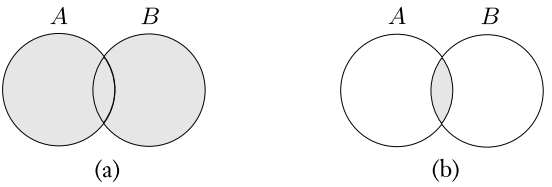


FIGURE 0.4
Diagrams for (a) $A \cup B$ and (b) $A \cap B$

SEQUENCES AND TUPLES

A ***sequence*** of objects is a list of these objects in some order. We usually designate a sequence by writing the list within parentheses. For example, the sequence 7, 21, 57 would be written

$$(7, 21, 57).$$

The order doesn't matter in a set, but in a sequence it does. Hence $(7, 21, 57)$ is not the same as $(57, 7, 21)$. Similarly, repetition does matter in a sequence, but it doesn't matter in a set. Thus $(7, 7, 21, 57)$ is different from both of the other sequences, whereas the set $\{7, 21, 57\}$ is identical to the set $\{7, 7, 21, 57\}$.

As with sets, sequences may be finite or infinite. Finite sequences often are called ***tuples***. A sequence with k elements is a ***k-tuple***. Thus $(7, 21, 57)$ is a 3-tuple. A 2-tuple is also called an ***ordered pair***.

Sets and sequences may appear as elements of other sets and sequences. For example, the ***power set*** of A is the set of all subsets of A. If A is the set $\{0, 1\}$, the power set of A is the set $\{ \emptyset, \{0\}, \{1\}, \{0, 1\} \}$. The set of all ordered pairs whose elements are 0s and 1s is $\{ (0, 0), (0, 1), (1, 0), (1, 1) \}$.

If A and B are two sets, the ***Cartesian product*** or ***cross product*** of A and B, written $A \times B$, is the set of all ordered pairs wherein the first element is a member of A and the second element is a member of B.

EXAMPLE 0.5

If $A = \{1, 2\}$ and $B = \{x, y, z\}$,

$$A \times B = \{ (1, x), (1, y), (1, z), (2, x), (2, y), (2, z) \}.$$

We can also take the Cartesian product of k sets, $A_1, A_2, \ldots, A_k$, written $A_1 \times A_2 \times \cdots \times A_k$. It is the set consisting of all k-tuples $(a_1, a_2, \ldots, a_k)$ where $a_i \in A_i$.

EXAMPLE 0.6

If A and B are as in Example 0.5,

$$A \times B \times A = \{\, (1,x,1),\ (1,x,2),\ (1,y,1),\ (1,y,2),\ (1,z,1),\ (1,z,2),\ (2,x,1),\ (2,x,2),\ (2,y,1),\ (2,y,2),\ (2,z,1),\ (2,z,2) \,\}.$$

If we have the Cartesian product of a set with itself, we use the shorthand

$$\overbrace{A \times A \times \cdots \times A}^{k} = A^k.$$

EXAMPLE 0.7

The set $\mathcal{N}^2$ equals $\mathcal{N} \times \mathcal{N}$. It consists of all ordered pairs of natural numbers. We also may write it as $\{(i,j) |\ i,j \geq 1\}$.

FUNCTIONS AND RELATIONS

Functions are central to mathematics. A ***function*** is an object that sets up an input–output relationship. A function takes an input and produces an output. In every function, the same input always produces the same output. If f is a function whose output value is b when the input value is a, we write

$$f(a) = b.$$

A function also is called a ***mapping***, and, if $f(a) = b$, we say that f maps a to b.

For example, the absolute value function abs takes a number x as input and returns x if x is positive and $-x$ if x is negative. Thus $abs(2) = abs(-2) = 2$. Addition is another example of a function, written add. The input to the addition function is an ordered pair of numbers, and the output is the sum of those numbers.

The set of possible inputs to the function is called its ***domain***. The outputs of a function come from a set called its ***range***. The notation for saying that f is a function with domain D and range R is

$$f\colon D \longrightarrow R.$$

In the case of the function abs, if we are working with integers, the domain and the range are $\mathcal{Z}$, so we write $abs\colon \mathcal{Z} \longrightarrow \mathcal{Z}$. In the case of the addition function for integers, the domain is the set of pairs of integers $\mathcal{Z} \times \mathcal{Z}$ and the range is $\mathcal{Z}$, so we write $add\colon \mathcal{Z} \times \mathcal{Z} \longrightarrow \mathcal{Z}$. Note that a function may not necessarily use all the elements of the specified range. The function abs never takes on the value -1 even though $-1 \in \mathcal{Z}$. A function that does use all the elements of the range is said to be ***onto*** the range.

We may describe a specific function in several ways. One way is with a procedure for computing an output from a specified input. Another way is with a table that lists all possible inputs and gives the output for each input.

EXAMPLE 0.8

Consider the function $f\colon \{0,1,2,3,4\} \longrightarrow \{0,1,2,3,4\}$.

n	$f(n)$
0	1
1	2
2	3
3	4
4	0

This function adds 1 to its input and then outputs the result modulo 5. A number modulo m is the remainder after division by m. For example, the minute hand on a clock face counts modulo 60. When we do modular arithmetic, we define $\mathcal{Z}_m = \{0,1,2,\ldots,m-1\}$. With this notation, the aforementioned function f has the form $f\colon \mathcal{Z}_5 \longrightarrow \mathcal{Z}_5$.

EXAMPLE 0.9

Sometimes a two-dimensional table is used if the domain of the function is the Cartesian product of two sets. Here is another function, $g\colon \mathcal{Z}_4 \times \mathcal{Z}_4 \longrightarrow \mathcal{Z}_4$. The entry at the row labeled i and the column labeled j in the table is the value of $g(i,j)$.

g	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

The function g is the addition function modulo 4.

When the domain of a function f is $A_1 \times \cdots \times A_k$ for some sets $A_1, \ldots, A_k$, the input to f is a k-tuple $(a_1, a_2, \ldots, a_k)$ and we call the a_i the ***arguments*** to f. A function with k arguments is called a ***k-ary function***, and k is called the ***arity*** of the function. If k is 1, f has a single argument and f is called a ***unary function***. If k is 2, f is a ***binary function***. Certain familiar binary functions are written in a special ***infix notation***, with the symbol for the function placed between its two arguments, rather than in ***prefix notation***, with the symbol preceding. For example, the addition function add usually is written in infix notation with the $+$ symbol between its two arguments as in $a + b$ instead of in prefix notation $add(a,b)$.

A ***predicate*** or ***property*** is a function whose range is {TRUE, FALSE}. For example, let $even$ be a property that is TRUE if its input is an even number and FALSE if its input is an odd number. Thus $even(4) =$ TRUE and $even(5) =$ FALSE.

A property whose domain is a set of k-tuples $A \times \cdots \times A$ is called a ***relation***, a ***k-ary relation***, or a ***k-ary relation on A***. A common case is a 2-ary relation, called a ***binary relation***. When writing an expression involving a binary relation, we customarily use infix notation. For example, "less than" is a relation usually written with the infix operation symbol $<$. "Equality", written with the $=$ symbol, is another familiar relation. If R is a binary relation, the statement aRb means that $aRb =$ TRUE. Similarly, if R is a k-ary relation, the statement $R(a_1, \ldots, a_k)$ means that $R(a_1, \ldots, a_k) =$ TRUE.

EXAMPLE 0.10

In a children's game called Scissors–Paper–Stone, the two players simultaneously select a member of the set {SCISSORS, PAPER, STONE} and indicate their selections with hand signals. If the two selections are the same, the game starts over. If the selections differ, one player wins, according to the relation *beats*.

beats	SCISSORS	PAPER	STONE
SCISSORS	FALSE	TRUE	FALSE
PAPER	FALSE	FALSE	TRUE
STONE	TRUE	FALSE	FALSE

From this table we determine that SCISSORS *beats* PAPER is TRUE and that PAPER *beats* SCISSORS is FALSE. ■

Sometimes describing predicates with sets instead of functions is more convenient. The predicate $P\colon D \longrightarrow \{\text{TRUE, FALSE}\}$ may be written (D, S), where $S = \{a \in D |\ P(a) = \text{TRUE}\}$, or simply S if the domain D is obvious from the context. Hence the relation *beats* may be written

$$\{(\text{SCISSORS, PAPER}),\ (\text{PAPER, STONE}),\ (\text{STONE, SCISSORS})\}.$$

A special type of binary relation, called an ***equivalence relation***, captures the notion of two objects being equal in some feature. A binary relation R is an equivalence relation if R satisfies three conditions:

1. R is ***reflexive*** if for every x, xRx;
2. R is ***symmetric*** if for every x and y, xRy implies yRx; and
3. R is ***transitive*** if for every x, y, and z, xRy and yRz implies xRz.

EXAMPLE 0.11

Define an equivalence relation on the natural numbers, written $\equiv_7$. For $i, j \in \mathcal{N}$, say that $i \equiv_7 j$, if $i - j$ is a multiple of 7. This is an equivalence relation because it satisfies the three conditions. First, it is reflexive, as $i - i = 0$, which is a multiple of 7. Second, it is symmetric, as $i - j$ is a multiple of 7 if $j - i$ is a multiple of 7. Third, it is transitive, as whenever $i - j$ is a multiple of 7 and $j - k$ is a multiple of 7, then $i - k = (i - j) + (j - k)$ is the sum of two multiples of 7 and hence a multiple of 7, too. ■

GRAPHS

An ***undirected graph***, or simply a ***graph***, is a set of points with lines connecting some of the points. The points are called ***nodes*** or ***vertices***, and the lines are called ***edges***, as shown in the following figure.

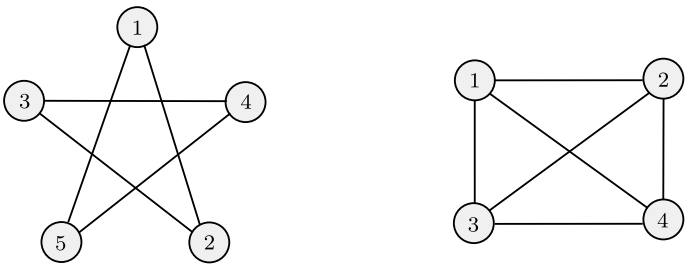


FIGURE 0.12
Examples of graphs

The number of edges at a particular node is the ***degree*** of that node. In Figure 0.12(a), all the nodes have degree 2. In Figure 0.12(b), all the nodes have degree 3. No more than one edge is allowed between any two nodes. We may allow an edge from a node to itself, called a ***self-loop***, depending on the situation.

In a graph G that contains nodes i and j, the pair (i, j) represents the edge that connects i and j. The order of i and j doesn't matter in an undirected graph, so the pairs (i, j) and (j, i) represent the same edge. Sometimes we describe undirected edges with unordered pairs using set notation as in $\{i, j\}$. If V is the set of nodes of G and E is the set of edges, we say $G = (V, E)$. We can describe a graph with a diagram or more formally by specifying V and E. For example, a formal description of the graph in Figure 0.12(a) is

$$\big(\{1, 2, 3, 4, 5\},\ \{(1, 2),\ (2, 3),\ (3, 4),\ (4, 5),\ (5, 1)\}\big),$$

and a formal description of the graph in Figure 0.12(b) is

$$\big(\{1,2,3,4\},\ \{(1,2),\ (1,3),\ (1,4),\ (2,3),\ (2,4),\ (3,4)\}\big).$$

Graphs frequently are used to represent data. Nodes might be cities and edges the connecting highways, or nodes might be people and edges the friendships between them. Sometimes, for convenience, we label the nodes and/or edges of a graph, which then is called a ***labeled graph***. Figure 0.13 depicts a graph whose nodes are cities and whose edges are labeled with the dollar cost of the cheapest nonstop airfare for travel between those cities if flying nonstop between them is possible.

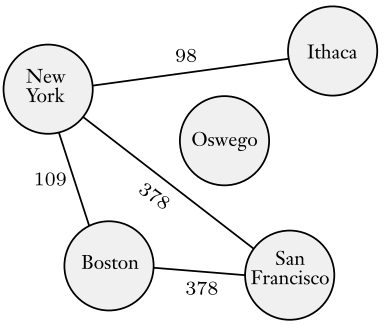


FIGURE 0.13
Cheapest nonstop airfares between various cities

We say that graph G is a ***subgraph*** of graph H if the nodes of G are a subset of the nodes of H, and the edges of G are the edges of H on the corresponding nodes. The following figure shows a graph H and a subgraph G.

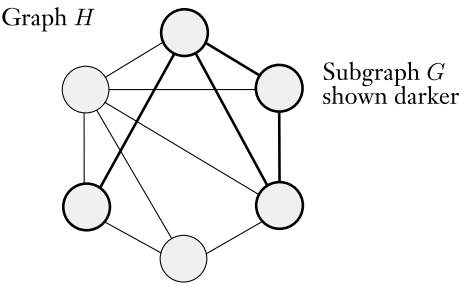


FIGURE 0.14
Graph G (shown darker) is a subgraph of H

A ***path*** in a graph is a sequence of nodes connected by edges. A ***simple path*** is a path that doesn't repeat any nodes. A graph is ***connected*** if every two nodes have a path between them. A path is a ***cycle*** if it starts and ends in the same node. A ***simple cycle*** is one that contains at least three nodes and repeats only the first and last nodes. A graph is a ***tree*** if it is connected and has no simple cycles, as shown in Figure 0.15. A tree may contain a specially designated node called the ***root***. The nodes of degree 1 in a tree, other than the root, are called the ***leaves*** of the tree.

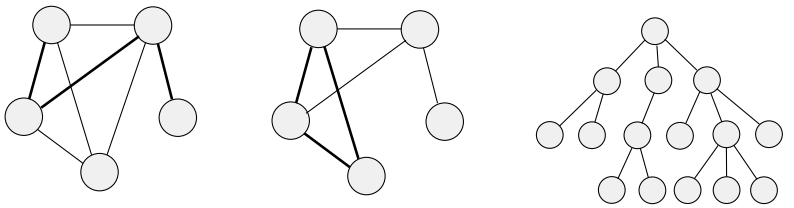


FIGURE 0.15
(a) A path in a graph, (b) a cycle in a graph, and (c) a tree

A ***directed graph*** has arrows instead of lines, as shown in the following figure. The number of arrows pointing from a particular node is the ***outdegree*** of that node, and the number of arrows pointing to a particular node is the ***indegree***.

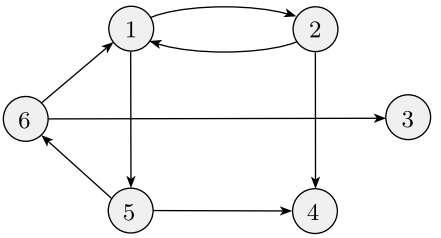


FIGURE 0.16
A directed graph

In a directed graph, we represent an edge from i to j as a pair (i, j). The formal description of a directed graph G is (V, E), where V is the set of nodes and E is the set of edges. The formal description of the graph in Figure 0.16 is

$$\big(\{1,2,3,4,5,6\},\ \{(1,2),(1,5),(2,1),(2,4),(5,4),(5,6),(6,1),(6,3)\}\big).$$

A path in which all the arrows point in the same direction as its steps is called a ***directed path***. A directed graph is ***strongly connected*** if a directed path connects every two nodes. Directed graphs are a handy way of depicting binary relations. If R is a binary relation whose domain is $D \times D$, a labeled graph $G = (D, E)$ represents R, where $E = \{(x, y) |\ xRy\}$.

EXAMPLE 0.17

The directed graph shown here represents the relation given in Example 0.10.

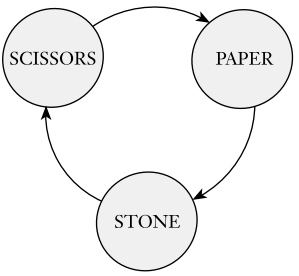


FIGURE 0.18
The graph of the relation *beats*

STRINGS AND LANGUAGES

Strings of characters are fundamental building blocks in computer science. The alphabet over which the strings are defined may vary with the application. For our purposes, we define an ***alphabet*** to be any nonempty finite set. The members of the alphabet are the ***symbols*** of the alphabet. We generally use capital Greek letters Σ and Γ to designate alphabets and a typewriter font for symbols from an alphabet. The following are a few examples of alphabets.

$$\Sigma_1 = \{\texttt{0},\texttt{1}\}$$

$$\Sigma_2 = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}, \texttt{g}, \texttt{h}, \texttt{i}, \texttt{j}, \texttt{k}, \texttt{l}, \texttt{m}, \texttt{n}, \texttt{o}, \texttt{p}, \texttt{q}, \texttt{r}, \texttt{s}, \texttt{t}, \texttt{u}, \texttt{v}, \texttt{w}, \texttt{x}, \texttt{y}, \texttt{z}\}$$

$$\Gamma = \{\texttt{0}, \texttt{1}, \texttt{x}, \texttt{y}, \texttt{z}\}$$

A ***string over an alphabet*** is a finite sequence of symbols from that alphabet, usually written next to one another and not separated by commas. If $\Sigma_1 = \{0,1\}$, then 01001 is a string over Σ_1. If $\Sigma_2 = \{a, b, c, \ldots, z\}$, then abracadabra is a string over Σ_2. If w is a string over Σ, the ***length*** of w, written $|w|$, is the number of symbols that it contains. The string of length zero is called the ***empty string*** and is written ε. The empty string plays the role of 0 in a number system. If w has length n, we can write $w = w_1 w_2 \cdots w_n$ where each $w_i \in \Sigma$. The ***reverse*** of w, written $w^{\mathcal{R}}$, is the string obtained by writing w in the opposite order (i.e., $w_n w_{n-1} \cdots w_1$). String z is a ***substring*** of w if z appears consecutively within w. For example, cad is a substring of abracadabra.

If we have string x of length m and string y of length n, the ***concatenation*** of x and y, written xy, is the string obtained by appending y to the end of x, as in $x_1 \cdots x_m y_1 \cdots y_n$. To concatenate a string with itself many times, we use the superscript notation x^k to mean

$$\overbrace{xx \cdots x}^{k}.$$

The ***lexicographic order*** of strings is the same as the familiar dictionary order. We'll occasionally use a modified lexicographic order, called ***shortlex order*** or simply ***string order***, that is identical to lexicographic order, except that shorter strings precede longer strings. Thus the string ordering of all strings over the alphabet {0,1} is

$$(\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots).$$

Say that string x is a ***prefix*** of string y if a string z exists where $xz = y$, and that x is a ***proper prefix*** of y if in addition $x \neq y$. A ***language*** is a set of strings. A language is ***prefix-free*** if no member is a proper prefix of another member.

BOOLEAN LOGIC

Boolean logic is a mathematical system built around the two values TRUE and FALSE. Though originally conceived of as pure mathematics, this system is now considered to be the foundation of digital electronics and computer design. The values TRUE and FALSE are called the ***Boolean values*** and are often represented by the values 1 and 0. We use Boolean values in situations with two possibilities, such as a wire that may have a high or a low voltage, a proposition that may be true or false, or a question that may be answered yes or no.

We can manipulate Boolean values with the ***Boolean operations***. The simplest Boolean operation is the ***negation*** or **NOT** operation, designated with the symbol $\neg$. The negation of a Boolean value is the opposite value. Thus $\neg 0 = 1$ and $\neg 1 = 0$. We designate the ***conjunction*** or **AND** operation with the symbol $\wedge$. The conjunction of two Boolean values is 1 if both of those values are 1. The ***disjunction*** or **OR** operation is designated with the symbol $\vee$. The disjunc-

tion of two Boolean values is 1 if either of those values is 1. We summarize this information as follows.

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & \neg 0 = 1 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & \neg 1 = 0 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & \end{array}$$

We use Boolean operations for combining simple statements into more complex Boolean expressions, just as we use the arithmetic operations $+$ and $\times$ to construct complex arithmetic expressions. For example, if P is the Boolean value representing the truth of the statement "the sun is shining" and Q represents the truth of the statement "today is Monday", we may write $P \wedge Q$ to represent the truth value of the statement "the sun is shining *and* today is Monday" and similarly for $P \vee Q$ with *and* replaced by *or*. The values P and Q are called the ***operands*** of the operation.

Several other Boolean operations occasionally appear. The ***exclusive or***, or **XOR**, operation is designated by the $\oplus$ symbol and is 1 if either but not both of its two operands is 1. The ***equality*** operation, written with the symbol $\leftrightarrow$, is 1 if both of its operands have the same value. Finally, the ***implication*** operation is designated by the symbol $\rightarrow$ and is 0 if its first operand is 1 and its second operand is 0; otherwise, $\rightarrow$ is 1. We summarize this information as follows.

$$\begin{array}{lll} 0 \oplus 0 = 0 & 0 \leftrightarrow 0 = 1 & 0 \rightarrow 0 = 1 \\ 0 \oplus 1 = 1 & 0 \leftrightarrow 1 = 0 & 0 \rightarrow 1 = 1 \\ 1 \oplus 0 = 1 & 1 \leftrightarrow 0 = 0 & 1 \rightarrow 0 = 0 \\ 1 \oplus 1 = 0 & 1 \leftrightarrow 1 = 1 & 1 \rightarrow 1 = 1 \end{array}$$

We can establish various relationships among these operations. In fact, we can express all Boolean operations in terms of the AND and NOT operations, as the following identities show. The two expressions in each row are equivalent. Each row expresses the operation in the left-hand column in terms of operations above it and AND and NOT.

$$\begin{array}{ll} P \vee Q & \neg(\neg P \wedge \neg Q) \\ P \rightarrow Q & \neg P \vee Q \\ P \leftrightarrow Q & (P \rightarrow Q) \wedge (Q \rightarrow P) \\ P \oplus Q & \neg(P \leftrightarrow Q) \end{array}$$

The ***distributive law*** for AND and OR comes in handy when we manipulate Boolean expressions. It is similar to the distributive law for addition and multiplication, which states that $a \times (b + c) = (a \times b) + (a \times c)$. The Boolean version comes in two forms:

- $P \wedge (Q \vee R)$ equals $(P \wedge Q) \vee (P \wedge R)$, and its dual
- $P \vee (Q \wedge R)$ equals $(P \vee Q) \wedge (P \vee R)$.

SUMMARY OF MATHEMATICAL TERMS

Alphabet	A finite, nonempty set of objects called symbols
Argument	An input to a function
Binary relation	A relation whose domain is a set of pairs
Boolean operation	An operation on Boolean values
Boolean value	The values TRUE or FALSE, often represented by 1 or 0
Cartesian product	An operation on sets forming a set of all tuples of elements from respective sets
Complement	An operation on a set, forming the set of all elements not present
Concatenation	An operation that joins strings together
Conjunction	Boolean AND operation
Connected graph	A graph with paths connecting every two nodes
Cycle	A path that starts and ends in the same node
Directed graph	A collection of points and arrows connecting some pairs of points
Disjunction	Boolean OR operation
Domain	The set of possible inputs to a function
Edge	A line in a graph
Element	An object in a set
Empty set	The set with no members
Empty string	The string of length zero
Equivalence relation	A binary relation that is reflexive, symmetric, and transitive
Function	An operation that translates inputs into outputs
Graph	A collection of points and lines connecting some pairs of points
Intersection	An operation on sets forming the set of common elements
k-tuple	A list of k objects
Language	A set of strings
Member	An object in a set
Node	A point in a graph
Ordered pair	A list of two elements
Path	A sequence of nodes in a graph connected by edges
Predicate	A function whose range is {TRUE, FALSE}
Property	A predicate
Range	The set from which outputs of a function are drawn
Relation	A predicate, most typically when the domain is a set of k-tuples
Sequence	A list of objects
Set	A group of objects
Simple path	A path without repetition
Singleton set	A set with one member
String	A finite list of symbols from an alphabet
Symbol	A member of an alphabet
Tree	A connected graph without simple cycles
Union	An operation on sets combining all elements into a single set
Unordered pair	A set with two members
Vertex	A point in a graph

0.3 DEFINITIONS, THEOREMS, AND PROOFS

Theorems and proofs are the heart and soul of mathematics and definitions are its spirit. These three entities are central to every mathematical subject, including ours.

Definitions describe the objects and notions that we use. A definition may be simple, as in the definition of *set* given earlier in this chapter, or complex as in the definition of *security* in a cryptographic system. Precision is essential to any mathematical definition. When defining some object, we must make clear what constitutes that object and what does not.

After we have defined various objects and notions, we usually make ***mathematical statements*** about them. Typically, a statement expresses that some object has a certain property. The statement may or may not be true; but like a definition, it must be precise. No ambiguity about its meaning is allowed.

A ***proof*** is a convincing logical argument that a statement is true. In mathematics, an argument must be airtight; that is, convincing in an absolute sense. In everyday life or in the law, the standard of proof is lower. A murder trial demands proof "beyond any reasonable doubt." The weight of evidence may compel the jury to accept the innocence or guilt of the suspect. However, evidence plays no role in a mathematical proof. A mathematician demands proof beyond *any* doubt.

A ***theorem*** is a mathematical statement proved true. Generally we reserve the use of that word for statements of special interest. Occasionally we prove statements that are interesting only because they assist in the proof of another, more significant statement. Such statements are called ***lemmas***. Occasionally a theorem or its proof may allow us to conclude easily that other, related statements are true. These statements are called ***corollaries*** of the theorem.

FINDING PROOFS

The only way to determine the truth or falsity of a mathematical statement is with a mathematical proof. Unfortunately, finding proofs isn't always easy. It can't be reduced to a simple set of rules or processes. During this course, you will be asked to present proofs of various statements. Don't despair at the prospect! Even though no one has a recipe for producing proofs, some helpful general strategies are available.

First, carefully read the statement you want to prove. Do you understand all the notation? Rewrite the statement in your own words. Break it down and consider each part separately.

Sometimes the parts of a multipart statement are not immediately evident. One frequently occurring type of multipart statement has the form "P if and only if Q", often written "P iff Q", where both P and Q are mathematical statements. This notation is shorthand for a two-part statement. The first part is "P only if Q," which means: If P is true, then Q is true, written $P \Rightarrow Q$. The second is "P if Q," which means: If Q is true, then P is true, written $P \Leftarrow Q$. The first of these parts is the ***forward direction*** of the original statement and the second is the ***reverse direction***. We write "P if and only if Q" as $P \iff Q$. To prove a statement of this form, you must prove each of the two directions. Often, one of these directions is easier to prove than the other.

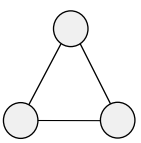
Another type of multipart statement states that two sets A and B are equal. The first part states that A is a subset of B, and the second part states that B is a subset of A. Thus one common way to prove that $A = B$ is to prove that every member of A also is a member of B, and that every member of B also is a member of A.

Next, when you want to prove a statement or part thereof, try to get an intuitive, "gut" feeling of why it should be true. Experimenting with examples is especially helpful. Thus if the statement says that all objects of a certain type have a particular property, pick a few objects of that type and observe that they actually do have that property. After doing so, try to find an object that fails to have the property, called a ***counterexample***. If the statement actually is true, you will not be able to find a counterexample. Seeing where you run into difficulty when you attempt to find a counterexample can help you understand why the statement is true.

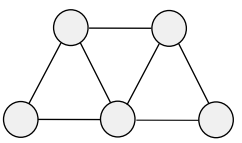
EXAMPLE 0.19

Suppose that you want to prove the statement *for every graph G, the sum of the degrees of all the nodes in G is an even number*.

First, pick a few graphs and observe this statement in action. Here are two examples.

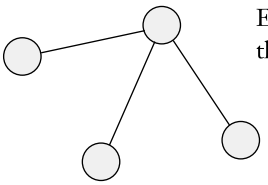


sum = 2+2+2
= 6



sum = 2+3+4+3+2
= 14

Next, try to find a counterexample; that is, a graph in which the sum is an odd number.



Every time an edge is added, the sum increases by 2.

Can you now begin to see why the statement is true and how to prove it? ■

If you are still stuck trying to prove a statement, try something easier. Attempt to prove a special case of the statement. For example, if you are trying to prove that some property is true for every $k > 0$, first try to prove it for $k = 1$. If you succeed, try it for $k = 2$, and so on until you can understand the more general case. If a special case is hard to prove, try a different special case or perhaps a special case of the special case.

Finally, when you believe that you have found the proof, you must write it up properly. A well-written proof is a sequence of statements, wherein each one follows by simple reasoning from previous statements in the sequence. Carefully writing a proof is important, both to enable a reader to understand it, and for you to be sure that it is free from errors.

The following are a few tips for producing a proof.

- *Be patient*. Finding proofs takes time. If you don't see how to do it right away, don't worry. Researchers sometimes work for weeks or even years to find a single proof.

- *Come back to it*. Look over the statement you want to prove, think about it a bit, leave it, and then return a few minutes or hours later. Let the unconscious, intuitive part of your mind have a chance to work.

- *Be neat*. When you are building your intuition for the statement you are trying to prove, use simple, clear pictures and/or text. You are trying to develop your insight into the statement, and sloppiness gets in the way of insight. Furthermore, when you are writing a solution for another person to read, neatness will help that person understand it.

- *Be concise*. Brevity helps you express high-level ideas without getting lost in details. Good mathematical notation is useful for expressing ideas concisely. But be sure to include enough of your reasoning when writing up a proof so that the reader can easily understand what you are trying to say.

For practice, let's prove one of DeMorgan's laws.

THEOREM 0.20

For any two sets A and B, $\overline{A \cup B} = \overline{A} \cap \overline{B}$.

First, is the meaning of this theorem clear? If you don't understand the meaning of the symbols $\cup$ or $\cap$ or the overbar, review the discussion on page 4.

To prove this theorem, we must show that the two sets $\overline{A \cup B}$ and $\overline{A} \cap \overline{B}$ are equal. Recall that we may prove that two sets are equal by showing that every member of one set also is a member of the other and vice versa. Before looking at the following proof, consider a few examples and then try to prove it yourself.

PROOF This theorem states that two sets, $\overline{A \cup B}$ and $\overline{A} \cap \overline{B}$, are equal. We prove this assertion by showing that every element of one also is an element of the other and vice versa.

Suppose that x is an element of $\overline{A \cup B}$. Then x is not in $A \cup B$ from the definition of the complement of a set. Therefore, x is not in A and x is not in B, from the definition of the union of two sets. In other words, x is in $\overline{A}$ and x is in $\overline{B}$. Hence the definition of the intersection of two sets shows that x is in $\overline{A} \cap \overline{B}$.

For the other direction, suppose that x is in $\overline{A} \cap \overline{B}$. Then x is in both $\overline{A}$ and $\overline{B}$. Therefore, x is not in A and x is not in B, and thus not in the union of these two sets. Hence x is in the complement of the union of these sets; in other words, x is in $\overline{A \cup B}$, which completes the proof of the theorem.

Let's now prove the statement in Example 0.19.

THEOREM 0.21

For every graph G, the sum of the degrees of all the nodes in G is an even number.

PROOF Every edge in G is connected to two nodes. Each edge contributes 1 to the degree of each node to which it is connected. Therefore, each edge contributes 2 to the sum of the degrees of all the nodes. Hence, if G contains e edges, then the sum of the degrees of all the nodes of G is $2e$, which is an even number.

0.4 TYPES OF PROOF

Several types of arguments arise frequently in mathematical proofs. Here, we describe a few that often occur in the theory of computation. Note that a proof may contain more than one type of argument because the proof may contain within it several different subproofs.

PROOF BY CONSTRUCTION

Many theorems state that a particular type of object exists. One way to prove such a theorem is by demonstrating how to construct the object. This technique is a ***proof by construction***.

Let's use a proof by construction to prove the following theorem. We define a graph to be ***k-regular*** if every node in the graph has degree k.

THEOREM 0.22

For each even number n greater than 2, there exists a 3-regular graph with n nodes.

PROOF Let n be an even number greater than 2. Construct graph $G = (V, E)$ with n nodes as follows. The set of nodes of G is $V = \{0, 1, \ldots, n-1\}$, and the set of edges of G is the set

$$E = \{\, \{i, i+1\} \mid \text{ for } 0 \le i \le n-2\} \cup \{\, \{n-1, 0\}\,\} \\ \cup \{\, \{i, i+n/2\} \mid \text{ for } 0 \le i \le n/2-1\}.$$

Picture the nodes of this graph written consecutively around the circumference of a circle. In that case, the edges described in the top line of E go between adjacent pairs around the circle. The edges described in the bottom line of E go between nodes on opposite sides of the circle. This mental picture clearly shows that every node in G has degree 3.

PROOF BY CONTRADICTION

In one common form of argument for proving a theorem, we assume that the theorem is false and then show that this assumption leads to an obviously false consequence, called a contradiction. We use this type of reasoning frequently in everyday life, as in the following example.

EXAMPLE 0.23

Jack sees Jill, who has just come in from outdoors. On observing that she is completely dry, he knows that it is not raining. His "proof" that it is not raining is that *if it were raining* (the assumption that the statement is false), *Jill would be wet* (the obviously false consequence). Therefore, it must not be raining. ■

Next, let's prove by contradiction that the square root of 2 is an irrational number. A number is ***rational*** if it is a fraction $\frac{m}{n}$, where m and n are integers; in other words, a rational number is the *ratio* of integers m and n. For example, $\frac{2}{3}$ obviously is a rational number. A number is ***irrational*** if it is not rational.

THEOREM 0.24

$\sqrt{2}$ is irrational.

PROOF First, we assume for the purpose of later obtaining a contradiction that $\sqrt{2}$ is rational. Thus

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

where m and n are integers. If both m and n are divisible by the same integer greater than 1, divide both by the largest such integer. Doing so doesn't change the value of the fraction. Now, at least one of m and n must be an odd number.

We multiply both sides of the equation by n and obtain

$$n\sqrt{2} = m.$$

We square both sides and obtain

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

Because m^2 is 2 times the integer n^2, we know that m^2 is even. Therefore, m, too, is even, as the square of an odd number always is odd. So we can write $m = 2k$ for some integer k. Then, substituting $2k$ for m, we get

$$\begin{aligned} 2n^2 &= (2k)^2 \\ &= 4k^2. \end{aligned}$$

Dividing both sides by 2, we obtain

$$n^2 = 2k^2.$$

But this result shows that n^2 is even and hence that n is even. Thus we have established that both m and n are even. But we had earlier reduced m and n so that they were *not* both even—a contradiction.

PROOF BY INDUCTION

Proof by induction is an advanced method used to show that all elements of an infinite set have a specified property. For example, we may use a proof by induction to show that an arithmetic expression computes a desired quantity for

every assignment to its variables, or that a program works correctly at all steps or for all inputs.

To illustrate how proof by induction works, let's take the infinite set to be the natural numbers, $\mathcal{N} = \{1, 2, 3, \ldots\}$, and say that the property is called $\mathcal{P}$. Our goal is to prove that $\mathcal{P}(k)$ is true for each natural number k. In other words, we want to prove that $\mathcal{P}(1)$ is true, as well as $\mathcal{P}(2)$, $\mathcal{P}(3)$, $\mathcal{P}(4)$, and so on.

Every proof by induction consists of two parts, the ***basis*** and the ***induction step***. Each part is an individual proof on its own. The basis proves that $\mathcal{P}(1)$ is true. The induction step proves that for each $i \geq 1$, if $\mathcal{P}(i)$ is true, then so is $\mathcal{P}(i+1)$.

When we have proven both of these parts, the desired result follows—namely, that $\mathcal{P}(i)$ is true for each i. Why? First, we know that $\mathcal{P}(1)$ is true because the basis alone proves it. Second, we know that $\mathcal{P}(2)$ is true because the induction step proves that if $\mathcal{P}(1)$ is true then $\mathcal{P}(2)$ is true, and we already know that $\mathcal{P}(1)$ is true. Third, we know that $\mathcal{P}(3)$ is true because the induction step proves that if $\mathcal{P}(2)$ is true then $\mathcal{P}(3)$ is true, and we already know that $\mathcal{P}(2)$ is true. This process continues for all natural numbers, showing that $\mathcal{P}(4)$ is true, $\mathcal{P}(5)$ is true, and so on.

Once you understand the preceding paragraph, you can easily understand variations and generalizations of the same idea. For example, the basis doesn't necessarily need to start with 1; it may start with any value b. In that case, the induction proof shows that $\mathcal{P}(k)$ is true for every k that is at least b.

In the induction step, the assumption that $\mathcal{P}(i)$ is true is called the ***induction hypothesis***. Sometimes having the stronger induction hypothesis that $\mathcal{P}(j)$ is true for every $j \leq i$ is useful. The induction proof still works because when we want to prove that $\mathcal{P}(i+1)$ is true, we have already proved that $\mathcal{P}(j)$ is true for every $j \leq i$.

The format for writing down a proof by induction is as follows.

Basis: Prove that $\mathcal{P}(1)$ is true.

$\vdots$

Induction step: For each $i \geq 1$, assume that $\mathcal{P}(i)$ is true and use this assumption to show that $\mathcal{P}(i+1)$ is true.

$\vdots$

Now, let's prove by induction the correctness of the formula used to calculate the size of monthly payments of home mortgages. When buying a home, many people borrow some of the money needed for the purchase and repay this loan over a certain number of years. Typically, the terms of such repayments stipulate that a fixed amount of money is paid each month to cover the interest, as well as part of the original sum, so that the total is repaid in 30 years. The formula for calculating the size of the monthly payments is shrouded in mystery, but actually is quite simple. It touches many people's lives, so you should find it interesting. We use induction to prove that it works, making it a good illustration of that technique.

First, we set up the names and meanings of several variables. Let P be the *principal*, the amount of the original loan. Let $I > 0$ be the yearly *interest rate* of the loan, where $I = 0.06$ indicates a 6% rate of interest. Let Y be the monthly payment. For convenience, we use I to define another variable M, the monthly multiplier. It is the rate at which the loan changes each month because of the interest on it. Following standard banking practice, the monthly interest rate is one-twelfth of the annual rate so $M = 1 + I/12$, and interest is paid monthly (monthly compounding).

Two things happen each month. First, the amount of the loan tends to increase because of the monthly multiplier. Second, the amount tends to decrease because of the monthly payment. Let P_t be the amount of the loan outstanding after the tth month. Then $P_0 = P$ is the amount of the original loan, $P_1 = MP_0 - Y$ is the amount of the loan after one month, $P_2 = MP_1 - Y$ is the amount of the loan after two months, and so on. Now we are ready to state and prove a theorem by induction on t that gives a formula for the value of P_t.

THEOREM 0.25

For each $t \geq 0$,

$$P_t = PM^t - Y\left(\frac{M^t - 1}{M - 1}\right).$$

PROOF

Basis: Prove that the formula is true for $t = 0$. If $t = 0$, then the formula states that

$$P_0 = PM^0 - Y\left(\frac{M^0 - 1}{M - 1}\right).$$

We can simplify the right-hand side by observing that $M^0 = 1$. Thus we get

$$P_0 = P,$$

which holds because we have defined P_0 to be P. Therefore, we have proved that the basis of the induction is true.

Induction step: For each $k \geq 0$, assume that the formula is true for $t = k$ and show that it is true for $t = k + 1$. The induction hypothesis states that

$$P_k = PM^k - Y\left(\frac{M^k - 1}{M - 1}\right).$$

Our objective is to prove that

$$P_{k+1} = PM^{k+1} - Y\left(\frac{M^{k+1} - 1}{M - 1}\right).$$

We do so with the following steps. First, from the definition of P_{k+1} from P_k, we know that

$$P_{k+1} = P_k M - Y.$$

Therefore, using the induction hypothesis to calculate P_k,

$$P_{k+1} = \left[PM^k - Y\left(\frac{M^k - 1}{M - 1}\right)\right] M - Y.$$

Multiplying through by M and rewriting Y yields

$$\begin{aligned} P_{k+1} &= PM^{k+1} - Y\left(\frac{M^{k+1} - M}{M - 1}\right) - Y\left(\frac{M - 1}{M - 1}\right) \\ &= PM^{k+1} - Y\left(\frac{M^{k+1} - 1}{M - 1}\right). \end{aligned}$$

Thus the formula is correct for $t = k + 1$, which proves the theorem.

Problem 0.15 asks you to use the preceding formula to calculate actual mortgage payments.

EXERCISES

0.1 Examine the following formal descriptions of sets so that you understand which members they contain. Write a short informal English description of each set.

- **a.** $\{1, 3, 5, 7, \ldots\}$
- **b.** $\{\ldots, -4, -2, 0, 2, 4, \ldots\}$
- **c.** $\{n \mid n = 2m \text{ for some } m \text{ in } \mathcal{N}\}$
- **d.** $\{n \mid n = 2m \text{ for some } m \text{ in } \mathcal{N}, \text{ and } n = 3k \text{ for some } k \text{ in } \mathcal{N}\}$
- **e.** $\{w \mid w \text{ is a string of } \texttt{0}\text{s and } \texttt{1}\text{s and } w \text{ equals the reverse of } w\}$
- **f.** $\{n \mid n \text{ is an integer and } n = n + 1\}$

0.2 Write formal descriptions of the following sets.

- **a.** The set containing the numbers 1, 10, and 100
- **b.** The set containing all integers that are greater than 5
- **c.** The set containing all natural numbers that are less than 5
- **d.** The set containing the string `aba`
- **e.** The set containing the empty string
- **f.** The set containing nothing at all

0.3 Let A be the set $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ and B be the set $\{\mathtt{x}, \mathtt{y}\}$.

- **a.** Is A a subset of B?
- **b.** Is B a subset of A?
- **c.** What is $A \cup B$?
- **d.** What is $A \cap B$?
- **e.** What is $A \times B$?
- **f.** What is the power set of B?

0.4 If A has a elements and B has b elements, how many elements are in $A \times B$? Explain your answer.

0.5 If C is a set with c elements, how many elements are in the power set of C? Explain your answer.

0.6 Let X be the set $\{1, 2, 3, 4, 5\}$ and Y be the set $\{6, 7, 8, 9, 10\}$. The unary function $f\colon X \longrightarrow Y$ and the binary function $g\colon X \times Y \longrightarrow Y$ are described in the following tables.

n	$f(n)$
1	6
2	7
3	6
4	7
5	6

g	6	7	8	9	10
1	10	10	10	10	10
2	7	8	9	10	6
3	7	7	8	8	9
4	9	8	7	6	10
5	6	6	6	6	6

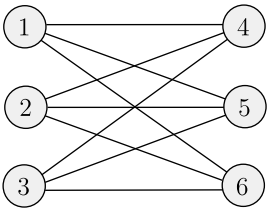
- **a.** What is the value of $f(2)$?
- **b.** What are the range and domain of f?
- **c.** What is the value of $g(2, 10)$?
- **d.** What are the range and domain of g?
- **e.** What is the value of $g(4, f(4))$?

0.7 For each part, give a relation that satisfies the condition.

- **a.** Reflexive and symmetric but not transitive
- **b.** Reflexive and transitive but not symmetric
- **c.** Symmetric and transitive but not reflexive

0.8 Consider the undirected graph $G = (V, E)$ where V, the set of nodes, is $\{1, 2, 3, 4\}$ and E, the set of edges, is $\{\{1, 2\}, \{2, 3\}, \{1, 3\}, \{2, 4\}, \{1, 4\}\}$. Draw the graph G. What are the degrees of each node? Indicate a path from node 3 to node 4 on your drawing of G.

0.9 Write a formal description of the following graph.



PROBLEMS

0.10 Find the error in the following proof that $2 = 1$.
Consider the equation $a = b$. Multiply both sides by a to obtain $a^2 = ab$. Subtract b^2 from both sides to get $a^2 - b^2 = ab - b^2$. Now factor each side, $(a+b)(a-b) = b(a-b)$, and divide each side by $(a-b)$ to get $a+b = b$. Finally, let a and b equal 1, which shows that $2 = 1$.

0.11 Let $S(n) = 1 + 2 + \cdots + n$ be the sum of the first n natural numbers and let $C(n) = 1^3 + 2^3 + \cdots + n^3$ be the sum of the first n cubes. Prove the following equalities by induction on n, to arrive at the curious conclusion that $C(n) = S^2(n)$ for every n.

a. $S(n) = \frac{1}{2}n(n+1)$.
b. $C(n) = \frac{1}{4}(n^4 + 2n^3 + n^2) = \frac{1}{4}n^2(n+1)^2$.

0.12 Find the error in the following proof that all horses are the same color.
CLAIM: In any set of h horses, all horses are the same color.
PROOF: By induction on h.

Basis: For $h = 1$. In any set containing just one horse, all horses clearly are the same color.

Induction step: For $k \geq 1$, assume that the claim is true for $h = k$ and prove that it is true for $h = k+1$. Take any set H of $k+1$ horses. We show that all the horses in this set are the same color. Remove one horse from this set to obtain the set H_1 with just k horses. By the induction hypothesis, all the horses in H_1 are the same color. Now replace the removed horse and remove a different one to obtain the set H_2. By the same argument, all the horses in H_2 are the same color. Therefore, all the horses in H must be the same color, and the proof is complete.

0.13 Show that every graph with two or more nodes contains two nodes that have equal degrees.

[A⋆]**0.14** **Ramsey's theorem.** Let G be a graph. A ***clique*** in G is a subgraph in which every two nodes are connected by an edge. An ***anti-clique***, also called an ***independent set***, is a subgraph in which every two nodes are not connected by an edge. Show that every graph with n nodes contains either a clique or an anti-clique with at least $\frac{1}{2}\log_2 n$ nodes.

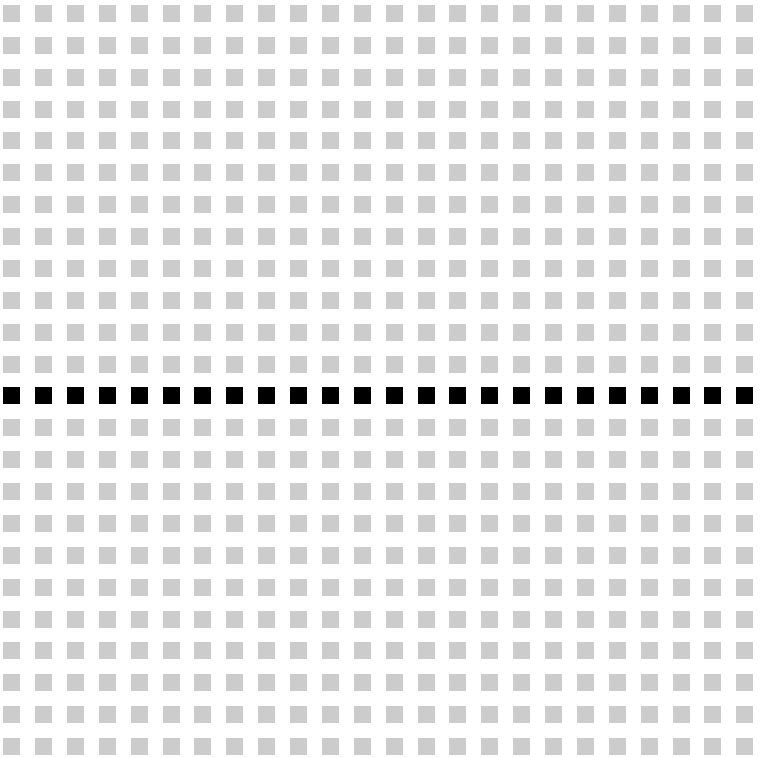
[A]**0.15** Use Theorem 0.25 to derive a formula for calculating the size of the monthly payment for a mortgage in terms of the principal P, the interest rate I, and the number of payments t. Assume that after t payments have been made, the loan amount is reduced to 0. Use the formula to calculate the dollar amount of each monthly payment for a 30-year mortgage with 360 monthly payments on an initial loan amount of \$100,000 with a 5% annual interest rate.

SELECTED SOLUTIONS

0.14 Make space for two piles of nodes: A and B. Then, starting with the entire graph, repeatedly add each remaining node x to A if its degree is greater than one half the number of remaining nodes and to B otherwise, and discard all nodes to which x isn't (is) connected if it was added to A (B). Continue until no nodes are left. At most half of the nodes are discarded at each of these steps, so at least $\log_2 n$ steps will occur before the process terminates. Each step adds a node to one of the piles, so one of the piles ends up with at least $\frac{1}{2}\log_2 n$ nodes. The A pile contains the nodes of a clique and the B pile contains the nodes of an anti-clique.

0.15 We let $P_t = 0$ and solve for Y to get the formula: $Y = PM^t(M-1)/(M^t-1)$. For $P = \$100{,}000$, $I = 0.05$, and $t = 360$, we have $M = 1 + (0.05)/12$. We use a calculator to find that $Y \approx \$536.82$ is the monthly payment.

PART ONE



AUTOMATA AND LANGUAGES

1

REGULAR LANGUAGES

The theory of computation begins with a question: What is a computer? It is perhaps a silly question, as everyone knows that this thing I type on is a computer. But these real computers are quite complicated—too much so to allow us to set up a manageable mathematical theory of them directly. Instead, we use an idealized computer called a ***computational model***. As with any model in science, a computational model may be accurate in some ways but perhaps not in others. Thus we will use several different computational models, depending on the features we want to focus on. We begin with the simplest model, called the ***finite state machine*** or ***finite automaton***.

1.1 FINITE AUTOMATA

Finite automata are good models for computers with an extremely limited amount of memory. What can a computer do with such a small memory? Many useful things! In fact, we interact with such computers all the time, as they lie at the heart of various electromechanical devices.

The controller for an automatic door is one example of such a device. Often found at supermarket entrances and exits, automatic doors swing open when the controller senses that a person is approaching. An automatic door has a pad

in front to detect the presence of a person about to walk through the doorway. Another pad is located to the rear of the doorway so that the controller can hold the door open long enough for the person to pass all the way through and also so that the door does not strike someone standing behind it as it opens. This configuration is shown in the following figure.

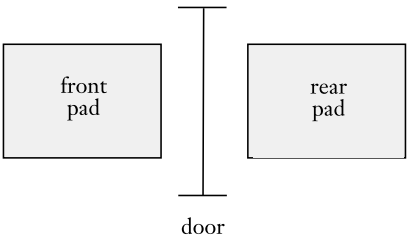


FIGURE 1.1
Top view of an automatic door

The controller is in either of two states: "OPEN" or "CLOSED," representing the corresponding condition of the door. As shown in the following figures, there are four possible input conditions: "FRONT" (meaning that a person is standing on the pad in front of the doorway), "REAR" (meaning that a person is standing on the pad to the rear of the doorway), "BOTH" (meaning that people are standing on both pads), and "NEITHER" (meaning that no one is standing on either pad).

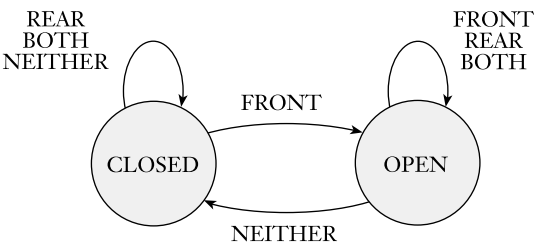


FIGURE 1.2
State diagram for an automatic door controller

		input signal			
		NEITHER	FRONT	REAR	BOTH
state	CLOSED	CLOSED	OPEN	CLOSED	CLOSED
	OPEN	CLOSED	OPEN	OPEN	OPEN

FIGURE 1.3
State transition table for an automatic door controller

The controller moves from state to state, depending on the input it receives. When in the CLOSED state and receiving input NEITHER or REAR, it remains in the CLOSED state. In addition, if the input BOTH is received, it stays CLOSED because opening the door risks knocking someone over on the rear pad. But if the input FRONT arrives, it moves to the OPEN state. In the OPEN state, if input FRONT, REAR, or BOTH is received, it remains in OPEN. If input NEITHER arrives, it returns to CLOSED.

For example, a controller might start in state CLOSED and receive the series of input signals FRONT, REAR, NEITHER, FRONT, BOTH, NEITHER, REAR, and NEITHER. It then would go through the series of states CLOSED (starting), OPEN, OPEN, CLOSED, OPEN, OPEN, CLOSED, CLOSED, and CLOSED.

Thinking of an automatic door controller as a finite automaton is useful because that suggests standard ways of representation as in Figures 1.2 and 1.3. This controller is a computer that has just a single bit of memory, capable of recording which of the two states the controller is in. Other common devices have controllers with somewhat larger memories. In an elevator controller, a state may represent the floor the elevator is on and the inputs might be the signals received from the buttons. This computer might need several bits to keep track of this information. Controllers for various household appliances such as dishwashers and electronic thermostats, as well as parts of digital watches and calculators, are additional examples of computers with limited memories. The design of such devices requires keeping the methodology and terminology of finite automata in mind.

Finite automata and their probabilistic counterpart ***Markov chains*** are useful tools when we are attempting to recognize patterns in data. These devices are used in speech processing and in optical character recognition. Markov chains have even been used to model and predict price changes in financial markets.

We will now take a closer look at finite automata from a mathematical perspective. We will develop a precise definition of a finite automaton, terminology for describing and manipulating finite automata, and theoretical results that describe their power and limitations. Besides giving you a clearer understanding of what finite automata are and what they can and cannot do, this theoretical development will allow you to practice and become more comfortable with mathematical definitions, theorems, and proofs in a relatively simple setting.

In beginning to describe the mathematical theory of finite automata, we do so in the abstract, without reference to any particular application. The following figure depicts a finite automaton called M_1.

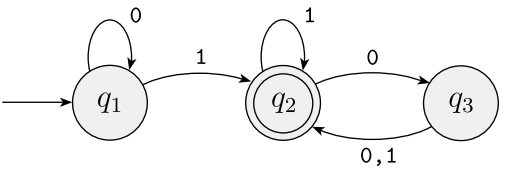


FIGURE 1.4
A finite automaton called M_1 that has three states

Figure 1.4 is called the ***state diagram*** of M_1. It has three ***states***, labeled q_1, q_2, and q_3. The ***start state***, q_1, is indicated by the arrow pointing at it from nowhere. The ***accept state***, q_2, is the one with a double circle. The arrows going from one state to another are called ***transitions***.

When this automaton receives an input string such as `1101`, it processes that string and produces an output. The output is either ***accept*** or ***reject***. We will consider only this yes/no type of output for now to keep things simple. The processing begins in M_1's start state. The automaton receives the symbols from the input string one by one from left to right. After reading each symbol, M_1 moves from one state to another along the transition that has that symbol as its label. When it reads the last symbol, M_1 produces its output. The output is *accept* if M_1 is now in an accept state and *reject* if it is not.

For example, when we feed the input string `1101` into the machine M_1 in Figure 1.4, the processing proceeds as follows:

1. Start in state q_1.
2. Read `1`, follow transition from q_1 to q_2.
3. Read `1`, follow transition from q_2 to q_2.
4. Read `0`, follow transition from q_2 to q_3.
5. Read `1`, follow transition from q_3 to q_2.
6. *Accept* because M_1 is in an accept state q_2 at the end of the input.

Experimenting with this machine on a variety of input strings reveals that it accepts the strings `1`, `01`, `11`, and `0101010101`. In fact, M_1 accepts any string that ends with a `1`, as it goes to its accept state q_2 whenever it reads the symbol `1`. In addition, it accepts strings `100`, `0100`, `110000`, and `0101000000`, and any string that ends with an even number of `0`s following the last `1`. It rejects other strings, such as `0`, `10`, `101000`. Can you describe the language consisting of all strings that M_1 accepts? We will do so shortly.

FORMAL DEFINITION OF A FINITE AUTOMATON

In the preceding section, we used state diagrams to introduce finite automata. Now we define finite automata formally. Although state diagrams are easier to grasp intuitively, we need the formal definition, too, for two specific reasons.

First, a formal definition is precise. It resolves any uncertainties about what is allowed in a finite automaton. If you were uncertain about whether finite automata were allowed to have 0 accept states or whether they must have exactly one transition exiting every state for each possible input symbol, you could consult the formal definition and verify that the answer is yes in both cases. Second, a formal definition provides notation. Good notation helps you think and express your thoughts clearly.

The language of a formal definition is somewhat arcane, having some similarity to the language of a legal document. Both need to be precise, and every detail must be spelled out.

A finite automaton has several parts. It has a set of states and rules for going from one state to another, depending on the input symbol. It has an input alphabet that indicates the allowed input symbols. It has a start state and a set of accept states. The formal definition says that a finite automaton is a list of those five objects: set of states, input alphabet, rules for moving, start state, and accept states. In mathematical language, a list of five elements is often called a 5-tuple. Hence we define a finite automaton to be a 5-tuple consisting of these five parts.

We use something called a ***transition function***, frequently denoted δ, to define the rules for moving. If the finite automaton has an arrow from a state x to a state y labeled with the input symbol 1, that means that if the automaton is in state x when it reads a 1, it then moves to state y. We can indicate the same thing with the transition function by saying that $\delta(x, 1) = y$. This notation is a kind of mathematical shorthand. Putting it all together, we arrive at the formal definition of finite automata.

DEFINITION 1.5

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta\colon Q \times \Sigma \longrightarrow Q$ is the ***transition function***,[1]
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.[2]

[1] Refer back to page 7 if you are uncertain about the meaning of $\delta\colon Q \times \Sigma \longrightarrow Q$.

[2] Accept states sometimes are called ***final states***.

The formal definition precisely describes what we mean by a finite automaton. For example, returning to the earlier question of whether 0 accept states is allowable, you can see that setting F to be the empty set $\emptyset$ yields 0 accept states, which is allowable. Furthermore, the transition function δ specifies exactly one next state for each possible combination of a state and an input symbol. That answers our other question affirmatively, showing that exactly one transition arrow exits every state for each possible input symbol.

We can use the notation of the formal definition to describe individual finite automata by specifying each of the five parts listed in Definition 1.5. For example, let's return to the finite automaton M_1 we discussed earlier, redrawn here for convenience.

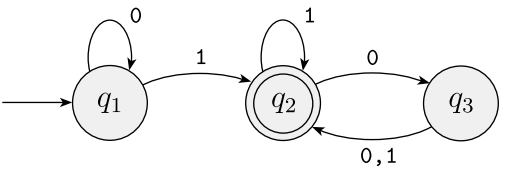


FIGURE 1.6
The finite automaton M_1

We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

4. q_1 is the start state, and
5. $F = \{q_2\}$.

If A is the set of all strings that machine M accepts, we say that A is the ***language of machine M*** and write $L(M) = A$. We say that ***M recognizes A*** or that ***M accepts A***. Because the term *accept* has different meanings when we refer to machines accepting strings and machines accepting languages, we prefer the term *recognize* for languages in order to avoid confusion.

A machine may accept several strings, but it always recognizes only one language. If the machine accepts no strings, it still recognizes one language—namely, the empty language $\emptyset$.

In our example, let

$$A = \{w | \ w \text{ contains at least one } \texttt{1} \text{ and an even number of } \texttt{0}\text{s follow the last } \texttt{1}\}.$$

Then $L(M_1) = A$, or equivalently, M_1 recognizes A.

EXAMPLES OF FINITE AUTOMATA

EXAMPLE 1.7

Here is the state diagram of finite automaton M_2.

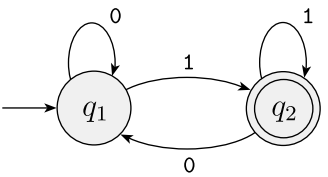


FIGURE 1.8
State diagram of the two-state finite automaton M_2

In the formal description, M_2 is $(\{q_1, q_2\}, \{\texttt{0},\texttt{1}\}, \delta, q_1, \{q_2\})$. The transition function δ is

	0	1
q_1	q_1	q_2
q_2	q_1	q_2.

Remember that the state diagram of M_2 and the formal description of M_2 contain the same information, only in different forms. You can always go from one to the other if necessary.

A good way to begin understanding any machine is to try it on some sample input strings. When you do these "experiments" to see how the machine is working, its method of functioning often becomes apparent. On the sample string 1101, the machine M_2 starts in its start state q_1 and proceeds first to state q_2 after reading the first 1, and then to states q_2, q_1, and q_2 after reading 1, 0, and 1. The string is accepted because q_2 is an accept state. But string 110 leaves M_2 in state q_1, so it is rejected. After trying a few more examples, you would see that M_2 accepts all strings that end in a 1. Thus $L(M_2) = \{w | \ w \text{ ends in a } \texttt{1}\}$.

EXAMPLE 1.9

Consider the finite automaton M_3.

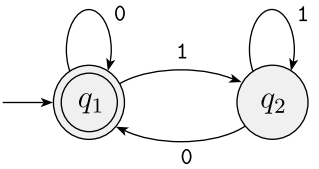


FIGURE 1.10
State diagram of the two-state finite automaton M_3

Machine M_3 is similar to M_2 except for the location of the accept state. As usual, the machine accepts all strings that leave it in an accept state when it has finished reading. Note that because the start state is also an accept state, M_3 accepts the empty string ε. As soon as a machine begins reading the empty string, it is at the end; so if the start state is an accept state, ε is accepted. In addition to the empty string, this machine accepts any string ending with a `0`. Here,

$$L(M_3) = \{w | \ w \text{ is the empty string } \varepsilon \text{ or ends in a } \mathtt{0}\}.$$

EXAMPLE 1.11

The following figure shows a five-state machine M_4.

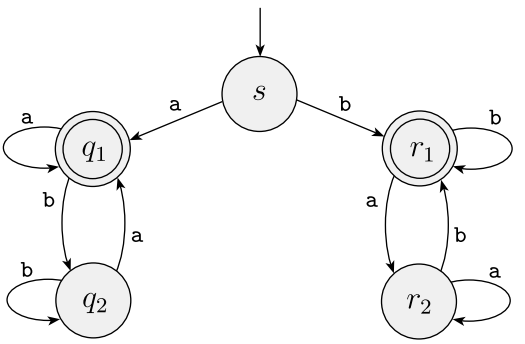


FIGURE 1.12
Finite automaton M_4

Machine M_4 has two accept states, q_1 and r_1, and operates over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$. Some experimentation shows that it accepts strings `a`, `b`, `aa`, `bb`, and `bab`, but not strings `ab`, `ba`, or `bbba`. This machine begins in state s, and after it reads the first symbol in the input, it goes either left into the q states or right into the r states. In both cases, it can never return to the start state (in contrast to the previous examples), as it has no way to get from any other state back to s. If the first symbol in the input string is `a`, then it goes left and accepts when the string ends with an `a`. Similarly, if the first symbol is a `b`, the machine goes right and accepts when the string ends in `b`. So M_4 accepts all strings that start and end with `a` or that start and end with `b`. In other words, M_4 accepts strings that start and end with the same symbol. ■

EXAMPLE 1.13

Figure 1.14 shows the three-state machine M_5, which has a four-symbol input alphabet, $\Sigma = \{\langle\text{RESET}\rangle, \texttt{0}, \texttt{1}, \texttt{2}\}$. We treat $\langle\text{RESET}\rangle$ as a single symbol.

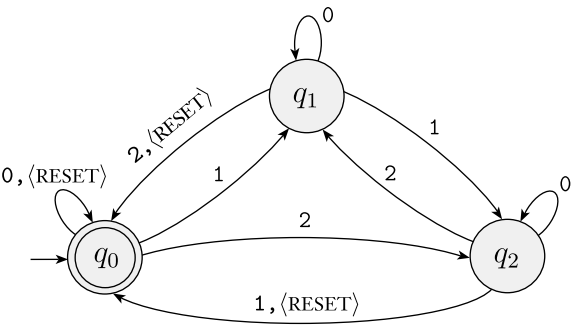


FIGURE 1.14
Finite automaton M_5

Machine M_5 keeps a running count of the sum of the numerical input symbols it reads, modulo 3. Every time it receives the $\langle\text{RESET}\rangle$ symbol, it resets the count to 0. It accepts if the sum is 0 modulo 3, or in other words, if the sum is a multiple of 3. ■

Describing a finite automaton by state diagram is not possible in some cases. That may occur when the diagram would be too big to draw or if, as in the next example, the description depends on some unspecified parameter. In these cases, we resort to a formal description to specify the machine.

EXAMPLE 1.15

Consider a generalization of Example 1.13, using the same four-symbol alphabet Σ. For each $i \geq 1$ let A_i be the language of all strings where the sum of the numbers is a multiple of i, except that the sum is reset to 0 whenever the symbol $\langle \text{RESET} \rangle$ appears. For each A_i we give a finite automaton B_i, recognizing A_i. We describe the machine B_i formally as follows: $B_i = (Q_i, \Sigma, \delta_i, q_0, \{q_0\})$, where Q_i is the set of i states $\{q_0, q_1, q_2, \ldots, q_{i-1}\}$, and we design the transition function δ_i so that for each j, if B_i is in q_j, the running sum is j, modulo i. For each q_j let

$$\delta_i(q_j, \mathtt{0}) = q_j,$$
$$\delta_i(q_j, \mathtt{1}) = q_k, \text{where } k = j + 1 \text{ modulo } i,$$
$$\delta_i(q_j, \mathtt{2}) = q_k, \text{where } k = j + 2 \text{ modulo } i, \text{ and}$$
$$\delta_i(q_j, \langle \text{RESET} \rangle) = q_0.$$

FORMAL DEFINITION OF COMPUTATION

So far we have described finite automata informally, using state diagrams, and with a formal definition, as a 5-tuple. The informal description is easier to grasp at first, but the formal definition is useful for making the notion precise, resolving any ambiguities that may have occurred in the informal description. Next we do the same for a finite automaton's computation. We already have an informal idea of the way it computes, and we now formalize it mathematically.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M ***accepts*** w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$, and
3. $r_n \in F$.

Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accept state. We say that M ***recognizes language*** A if $A = \{w | \ M \text{ accepts } w\}$.

DEFINITION 1.16

A language is called a ***regular language*** if some finite automaton recognizes it.

EXAMPLE 1.17

Take machine M_5 from Example 1.13. Let w be the string

$$\texttt{10}\langle\text{RESET}\rangle\texttt{22}\langle\text{RESET}\rangle\texttt{012}.$$

Then M_5 accepts w according to the formal definition of computation because the sequence of states it enters when computing on w is

$$q_0, q_1, q_1, q_0, q_2, q_1, q_0, q_0, q_1, q_0,$$

which satisfies the three conditions. The language of M_5 is

$$L(M_5) = \{w | \text{ the sum of the symbols in } w \text{ is 0 modulo 3,} \\ \text{except that } \langle\text{RESET}\rangle \text{ resets the count to 0}\}.$$

As M_5 recognizes this language, it is a regular language. ■

DESIGNING FINITE AUTOMATA

Whether it be of automaton or artwork, design is a creative process. As such, it cannot be reduced to a simple recipe or formula. However, you might find a particular approach helpful when designing various types of automata. That is, put *yourself* in the place of the machine you are trying to design and then see how you would go about performing the machine's task. Pretending that you are the machine is a psychological trick that helps engage your whole mind in the design process.

Let's design a finite automaton using the "reader as automaton" method just described. Suppose that you are given some language and want to design a finite automaton that recognizes it. Pretending to be the automaton, you receive an input string and must determine whether it is a member of the language the automaton is supposed to recognize. You get to see the symbols in the string one by one. After each symbol, you must decide whether the string seen so far is in the language. The reason is that you, like the machine, don't know when the end of the string is coming, so you must always be ready with the answer.

First, in order to make these decisions, you have to figure out what you need to remember about the string as you are reading it. Why not simply remember all you have seen? Bear in mind that you are pretending to be a finite automaton and that this type of machine has only a finite number of states, which means a finite memory. Imagine that the input is extremely long—say, from here to the moon—so that you could not possibly remember the entire thing. You have a finite memory—say, a single sheet of paper—which has a limited storage capacity. Fortunately, for many languages you don't need to remember the entire input. You need to remember only certain crucial information. Exactly which information is crucial depends on the particular language considered.

For example, suppose that the alphabet is $\{\texttt{0},\texttt{1}\}$ and that the language consists of all strings with an odd number of 1s. You want to construct a finite automaton E_1 to recognize this language. Pretending to be the automaton, you start getting

an input string of 0s and 1s symbol by symbol. Do you need to remember the entire string seen so far in order to determine whether the number of 1s is odd? Of course not. Simply remember whether the number of 1s seen so far is even or odd and keep track of this information as you read new symbols. If you read a 1, flip the answer; but if you read a 0, leave the answer as is.

But how does this help you design E_1? Once you have determined the necessary information to remember about the string as it is being read, you represent this information as a finite list of possibilities. In this instance, the possibilities would be

1. even so far, and
2. odd so far.

Then you assign a state to each of the possibilities. These are the states of E_1, as shown here.

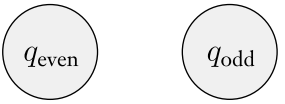


FIGURE 1.18
The two states q_{even} and q_{odd}

Next, you assign the transitions by seeing how to go from one possibility to another upon reading a symbol. So, if state q_{even} represents the even possibility and state q_{odd} represents the odd possibility, you would set the transitions to flip state on a 1 and stay put on a 0, as shown here.

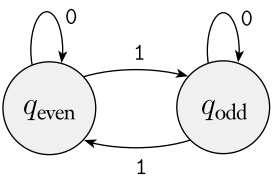


FIGURE 1.19
Transitions telling how the possibilities rearrange

Next, you set the start state to be the state corresponding to the possibility associated with having seen 0 symbols so far (the empty string ε). In this case, the start state corresponds to state q_{even} because 0 is an even number. Last, set the accept states to be those corresponding to possibilities where you want to accept the input string. Set q_{odd} to be an accept state because you want to accept

when you have seen an odd number of 1s. These additions are shown in the following figure.

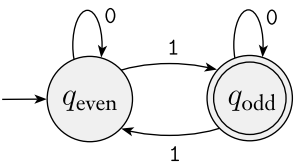


FIGURE 1.20
Adding the start and accept states

EXAMPLE 1.21

This example shows how to design a finite automaton E_2 to recognize the regular language of all strings that contain the string 001 as a substring. For example, 0010, 1001, 001, and 11111110011111 are all in the language, but 11 and 0000 are not. How would you recognize this language if you were pretending to be E_2? As symbols come in, you would initially skip over all 1s. If you come to a 0, then you note that you may have just seen the first of the three symbols in the pattern 001 you are seeking. If at this point you see a 1, there were too few 0s, so you go back to skipping over 1s. But if you see a 0 at that point, you should remember that you have just seen two symbols of the pattern. Now you simply need to continue scanning until you see a 1. If you find it, remember that you succeeded in finding the pattern and continue reading the input string until you get to the end.

So there are four possibilities: You

1. haven't just seen any symbols of the pattern,
2. have just seen a 0,
3. have just seen 00, or
4. have seen the entire pattern 001.

Assign the states q, q_0, q_{00}, and q_{001} to these possibilities. You can assign the transitions by observing that from q reading a 1 you stay in q, but reading a 0 you move to q_0. In q_0 reading a 1 you return to q, but reading a 0 you move to q_{00}. In q_{00} reading a 1 you move to q_{001}, but reading a 0 leaves you in q_{00}. Finally, in q_{001} reading a 0 or a 1 leaves you in q_{001}. The start state is q, and the only accept state is q_{001}, as shown in Figure 1.22.

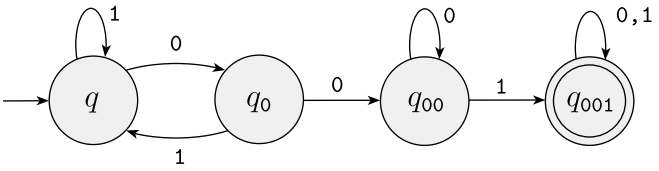


FIGURE 1.22
Accepts strings containing 001

THE REGULAR OPERATIONS

In the preceding two sections, we introduced and defined finite automata and regular languages. We now begin to investigate their properties. Doing so will help develop a toolbox of techniques for designing automata to recognize particular languages. The toolbox also will include ways of proving that certain other languages are nonregular (i.e., beyond the capability of finite automata).

In arithmetic, the basic objects are numbers and the tools are operations for manipulating them, such as $+$ and $\times$. In the theory of computation, the objects are languages and the tools include operations specifically designed for manipulating them. We define three operations on languages, called the ***regular operations***, and use them to study properties of the regular languages.

DEFINITION 1.23

Let A and B be languages. We define the regular operations ***union***, ***concatenation***, and ***star*** as follows:

- **Union**: $A \cup B = \{x | \ x \in A \text{ or } x \in B\}$.
- **Concatenation**: $A \circ B = \{xy | \ x \in A \text{ and } y \in B\}$.
- **Star**: $A^* = \{x_1 x_2 \ldots x_k | \ k \geq 0 \text{ and each } x_i \in A\}$.

You are already familiar with the union operation. It simply takes all the strings in both A and B and lumps them together into one language.

The concatenation operation is a little trickier. It attaches a string from A in front of a string from B in all possible ways to get the strings in the new language.

The star operation is a bit different from the other two because it applies to a single language rather than to two different languages. That is, the star operation is a ***unary operation*** instead of a ***binary operation***. It works by attaching any number of strings in A together to get a string in the new language. Because

"any number" includes 0 as a possibility, the empty string ε is always a member of A^*, no matter what A is.

EXAMPLE 1.24

Let the alphabet Σ be the standard 26 letters $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. If $A = \{\mathtt{good}, \mathtt{bad}\}$ and $B = \{\mathtt{boy}, \mathtt{girl}\}$, then

$A \cup B = \{\mathtt{good}, \mathtt{bad}, \mathtt{boy}, \mathtt{girl}\}$,

$A \circ B = \{\mathtt{goodboy}, \mathtt{goodgirl}, \mathtt{badboy}, \mathtt{badgirl}\}$, and

$A^* = \{\varepsilon, \mathtt{good}, \mathtt{bad}, \mathtt{goodgood}, \mathtt{goodbad}, \mathtt{badgood}, \mathtt{badbad}, \mathtt{goodgoodgood}, \mathtt{goodgoodbad}, \mathtt{goodbadgood}, \mathtt{goodbadbad}, \ldots\}$.

Let $\mathcal{N} = \{1, 2, 3, \ldots\}$ be the set of natural numbers. When we say that $\mathcal{N}$ is *closed under multiplication*, we mean that for any x and y in $\mathcal{N}$, the product $x \times y$ also is in $\mathcal{N}$. In contrast, $\mathcal{N}$ is not closed under division, as 1 and 2 are in $\mathcal{N}$ but $1/2$ is not. Generally speaking, a collection of objects is ***closed*** under some operation if applying that operation to members of the collection returns an object still in the collection. We show that the collection of regular languages is closed under all three of the regular operations. In Section 1.3, we show that these are useful tools for manipulating regular languages and understanding the power of finite automata. We begin with the union operation.

THEOREM 1.25

The class of regular languages is closed under the union operation.

In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

PROOF IDEA We have regular languages A_1 and A_2 and want to show that $A_1 \cup A_2$ also is regular. Because A_1 and A_2 are regular, we know that some finite automaton M_1 recognizes A_1 and some finite automaton M_2 recognizes A_2. To prove that $A_1 \cup A_2$ is regular, we demonstrate a finite automaton, call it M, that recognizes $A_1 \cup A_2$.

This is a proof by construction. We construct M from M_1 and M_2. Machine M must accept its input exactly when either M_1 or M_2 would accept it in order to recognize the union language. It works by *simulating* both M_1 and M_2 and accepting if either of the simulations accept.

How can we make machine M simulate M_1 and M_2? Perhaps it first simulates M_1 on the input and then simulates M_2 on the input. But we must be careful here! Once the symbols of the input have been read and used to simulate M_1, we can't "rewind the input tape" to try the simulation on M_2. We need another approach.

Pretend that you are M. As the input symbols arrive one by one, you simulate both M_1 and M_2 simultaneously. That way, only one pass through the input is necessary. But can you keep track of both simulations with finite memory? All you need to remember is the state that each machine would be in if it had read up to this point in the input. Therefore, you need to remember a pair of states. How many possible pairs are there? If M_1 has k_1 states and M_2 has k_2 states, the number of pairs of states, one from M_1 and the other from M_2, is the product $k_1 \times k_2$. This product will be the number of states in M, one for each pair. The transitions of M go from pair to pair, updating the current state for both M_1 and M_2. The accept states of M are those pairs wherein either M_1 or M_2 is in an accept state.

PROOF

Let M_1 recognize A_1, where $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, and
M_2 recognize A_2, where $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.

Construct M to recognize $A_1 \cup A_2$, where $M = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = \{(r_1, r_2) | \, r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$.
   This set is the ***Cartesian product*** of sets Q_1 and Q_2 and is written $Q_1 \times Q_2$. It is the set of all pairs of states, the first from Q_1 and the second from Q_2.
2. Σ, the alphabet, is the same as in M_1 and M_2. In this theorem and in all subsequent similar theorems, we assume for simplicity that both M_1 and M_2 have the same input alphabet Σ. The theorem remains true if they have different alphabets, Σ_1 and Σ_2. We would then modify the proof to let $\Sigma = \Sigma_1 \cup \Sigma_2$.
3. δ, the transition function, is defined as follows. For each $(r_1, r_2) \in Q$ and each $a \in \Sigma$, let
   $$\delta\big((r_1, r_2), a\big) = \big(\delta_1(r_1, a), \delta_2(r_2, a)\big).$$
   Hence δ gets a state of M (which actually is a pair of states from M_1 and M_2), together with an input symbol, and returns M's next state.
4. q_0 is the pair (q_1, q_2).
5. F is the set of pairs in which either member is an accept state of M_1 or M_2. We can write it as
   $$F = \{(r_1, r_2) | \, r_1 \in F_1 \text{ or } r_2 \in F_2\}.$$
   This expression is the same as $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$. (Note that it is *not* the same as $F = F_1 \times F_2$. What would that give us instead?[3])

[3] This expression would define M's accept states to be those for which *both* members of the pair are accept states. In this case, M would accept a string only if both M_1 *and* M_2 accept it, so the resulting language would be the *intersection* and not the union. In fact, this result proves that the class of regular languages is closed under intersection.

This concludes the construction of the finite automaton M that recognizes the union of A_1 and A_2. This construction is fairly simple, and thus its correctness is evident from the strategy described in the proof idea. More complicated constructions require additional discussion to prove correctness. A formal correctness proof for a construction of this type usually proceeds by induction. For an example of a construction proved correct, see the proof of Theorem 1.54. Most of the constructions that you will encounter in this course are fairly simple and so do not require a formal correctness proof.

We have just shown that the union of two regular languages is regular, thereby proving that the class of regular languages is closed under the union operation. We now turn to the concatenation operation and attempt to show that the class of regular languages is closed under that operation, too.

THEOREM 1.26

The class of regular languages is closed under the concatenation operation.

In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.

To prove this theorem, let's try something along the lines of the proof of the union case. As before, we can start with finite automata M_1 and M_2 recognizing the regular languages A_1 and A_2. But now, instead of constructing automaton M to accept its input if either M_1 or M_2 accept, it must accept if its input can be broken into two pieces, where M_1 accepts the first piece and M_2 accepts the second piece. The problem is that M doesn't know where to break its input (i.e., where the first part ends and the second begins). To solve this problem, we introduce a new technique called nondeterminism.

1.2 NONDETERMINISM

Nondeterminism is a useful concept that has had great impact on the theory of computation. So far in our discussion, every step of a computation follows in a unique way from the preceding step. When the machine is in a given state and reads the next input symbol, we know what the next state will be—it is determined. We call this ***deterministic*** computation. In a ***nondeterministic*** machine, several choices may exist for the next state at any point.

Nondeterminism is a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton. As Figure 1.27 shows, nondeterministic finite automata may have additional features.

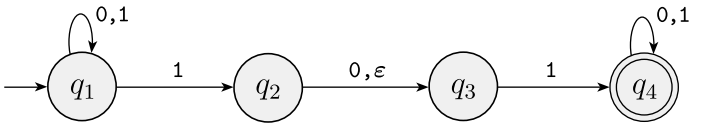


FIGURE 1.27
The nondeterministic finite automaton N_1

The difference between a deterministic finite automaton, abbreviated DFA, and a nondeterministic finite automaton, abbreviated NFA, is immediately apparent. First, every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. The NFA shown in Figure 1.27 violates that rule. State q_1 has one exiting arrow for 0, but it has two for 1; q_2 has one arrow for 0, but it has none for 1. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol.

Second, in a DFA, labels on the transition arrows are symbols from the alphabet. This NFA has an arrow with the label ε. In general, an NFA may have arrows labeled with members of the alphabet or ε. Zero, one, or many arrows may exit from each state with the label ε.

How does an NFA compute? Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed. For example, say that we are in state q_1 in NFA N_1 and that the next input symbol is a 1. After reading that symbol, the machine splits into multiple copies of itself and follows *all* the possibilities in parallel. Each copy of the machine takes one of the possible ways to proceed and continues as before. If there are subsequent choices, the machine splits again. If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it. Finally, if *any one* of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

If a state with an ε symbol on an exiting arrow is encountered, something similar happens. Without reading any input, the machine splits into multiple copies, one following each of the exiting ε-labeled arrows and one staying at the current state. Then the machine proceeds nondeterministically as before.

Nondeterminism may be viewed as a kind of parallel computation wherein multiple independent "processes" or "threads" can be running concurrently. When the NFA splits to follow several choices, that corresponds to a process "forking" into several children, each proceeding separately. If at least one of these processes accepts, then the entire computation accepts.

Another way to think of a nondeterministic computation is as a tree of possibilities. The root of the tree corresponds to the start of the computation. Every branching point in the tree corresponds to a point in the computation at which the machine has multiple choices. The machine accepts if at least one of the computation branches ends in an accept state, as shown in Figure 1.28.

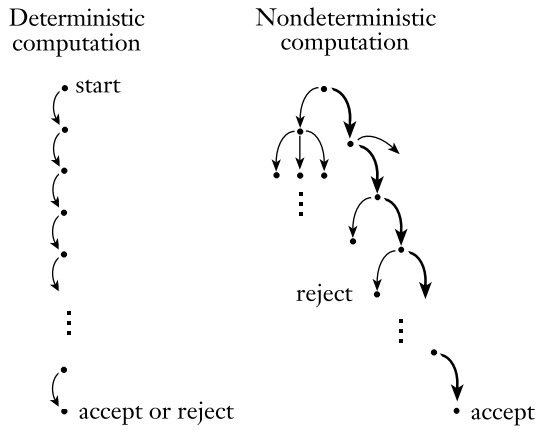


FIGURE 1.28
Deterministic and nondeterministic computations with an accepting branch

Let's consider some sample runs of the NFA N_1 shown in Figure 1.27. The computation of N_1 on input `010110` is depicted in the following figure.

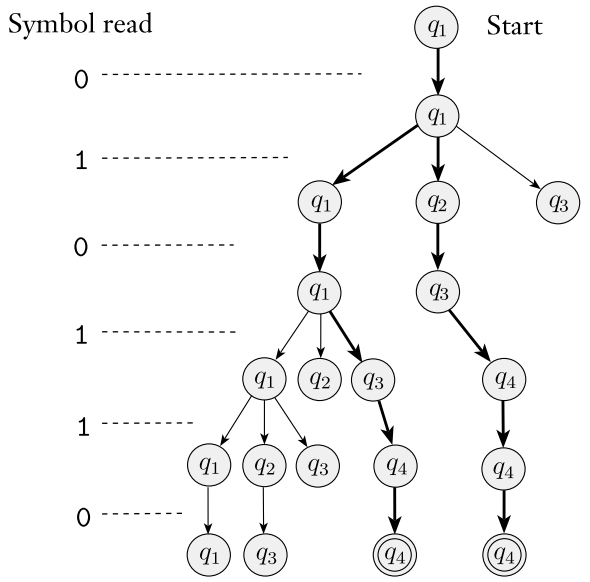


FIGURE 1.29
The computation of N_1 on input `010110`

On input 010110, start in the start state q_1 and read the first symbol 0. From q_1 there is only one place to go on a 0—namely, back to q_1—so remain there. Next, read the second symbol 1. In q_1 on a 1 there are two choices: either stay in q_1 or move to q_2. Nondeterministically, the machine splits in two to follow each choice. Keep track of the possibilities by placing a finger on each state where a machine could be. So you now have fingers on states q_1 and q_2. An ε arrow exits state q_2 so the machine splits again; keep one finger on q_2, and move the other to q_3. You now have fingers on q_1, q_2, and q_3.

When the third symbol 0 is read, take each finger in turn. Keep the finger on q_1 in place, move the finger on q_2 to q_3, and remove the finger that has been on q_3. That last finger had no 0 arrow to follow and corresponds to a process that simply "dies." At this point, you have fingers on states q_1 and q_3.

When the fourth symbol 1 is read, split the finger on q_1 into fingers on states q_1 and q_2, then further split the finger on q_2 to follow the ε arrow to q_3, and move the finger that was on q_3 to q_4. You now have a finger on each of the four states.

When the fifth symbol 1 is read, the fingers on q_1 and q_3 result in fingers on states q_1, q_2, q_3, and q_4, as you saw with the fourth symbol. The finger on state q_2 is removed. The finger that was on q_4 stays on q_4. Now you have two fingers on q_4, so remove one because you only need to remember that q_4 is a possible state at this point, not that it is possible for multiple reasons.

When the sixth and final symbol 0 is read, keep the finger on q_1 in place, move the one on q_2 to q_3, remove the one that was on q_3, and leave the one on q_4 in place. You are now at the end of the string, and you accept if some finger is on an accept state. You have fingers on states q_1, q_3, and q_4; and as q_4 is an accept state, N_1 accepts this string.

What does N_1 do on input 010? Start with a finger on q_1. After reading the 0, you still have a finger only on q_1; but after the 1 there are fingers on q_1, q_2, and q_3 (don't forget the ε arrow). After the third symbol 0, remove the finger on q_3, move the finger on q_2 to q_3, and leave the finger on q_1 where it is. At this point you are at the end of the input; and as no finger is on an accept state, N_1 rejects this input.

By continuing to experiment in this way, you will see that N_1 accepts all strings that contain either 101 or 11 as a substring.

Nondeterministic finite automata are useful in several respects. As we will show, every NFA can be converted into an equivalent DFA, and constructing NFAs is sometimes easier than directly constructing DFAs. An NFA may be much smaller than its deterministic counterpart, or its functioning may be easier to understand. Nondeterminism in finite automata is also a good introduction to nondeterminism in more powerful computational models because finite automata are especially easy to understand. Now we turn to several examples of NFAs.

EXAMPLE 1.30

Let A be the language consisting of all strings over {0,1} containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not). The following four-state NFA N_2 recognizes A.

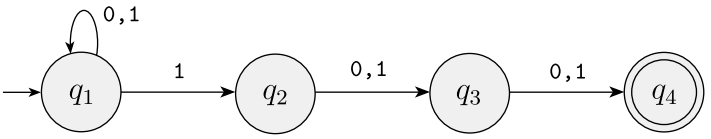


FIGURE 1.31
The NFA N_2 recognizing A

One good way to view the computation of this NFA is to say that it stays in the start state q_1 until it "guesses" that it is three places from the end. At that point, if the input symbol is a 1, it branches to state q_2 and uses q_3 and q_4 to "check" on whether its guess was correct.

As mentioned, every NFA can be converted into an equivalent DFA; but sometimes that DFA may have many more states. The smallest DFA for A contains eight states. Furthermore, understanding the functioning of the NFA is much easier, as you may see by examining the following figure for the DFA.

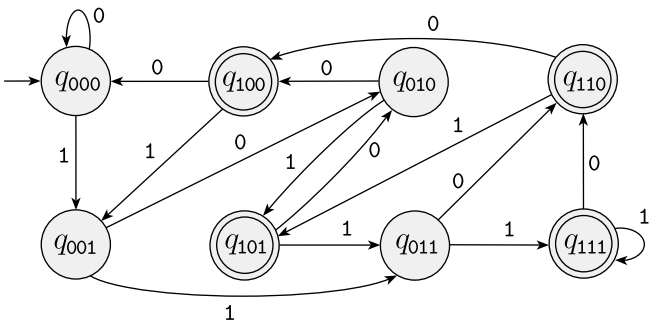


FIGURE 1.32
A DFA recognizing A

Suppose that we added ε to the labels on the arrows going from q_2 to q_3 and from q_3 to q_4 in machine N_2 in Figure 1.31. So both arrows would then have the label $0, 1, \varepsilon$ instead of just $0, 1$. What language would N_2 recognize with this modification? Try modifying the DFA in Figure 1.32 to recognize that language.

EXAMPLE 1.33

The following NFA N_3 has an input alphabet {0} consisting of a single symbol. An alphabet containing only one symbol is called a ***unary alphabet***.

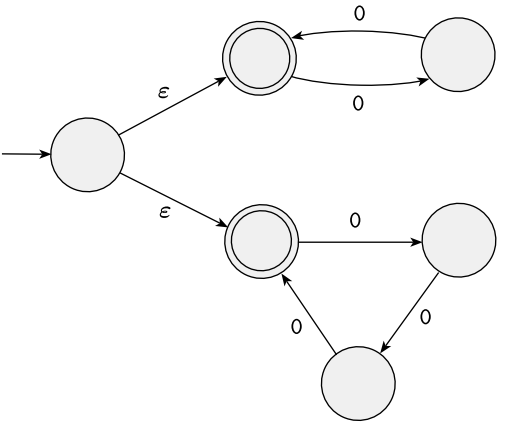


FIGURE 1.34
The NFA N_3

This machine demonstrates the convenience of having ε arrows. It accepts all strings of the form 0^k where k is a multiple of 2 or 3. (Remember that the superscript denotes repetition, not numerical exponentiation.) For example, N_3 accepts the strings ε, 00, 000, 0000, and 000000, but not 0 or 00000.

Think of the machine operating by initially guessing whether to test for a multiple of 2 or a multiple of 3 by branching into either the top loop or the bottom loop and then checking whether its guess was correct. Of course, we could replace this machine by one that doesn't have ε arrows or even any nondeterminism at all, but the machine shown is the easiest one to understand for this language. ■

EXAMPLE 1.35

We give another example of an NFA in Figure 1.36. Practice with it to satisfy yourself that it accepts the strings ε, a, baba, and baa, but that it doesn't accept the strings b, bb, and babba. Later we use this machine to illustrate the procedure for converting NFAs to DFAs.

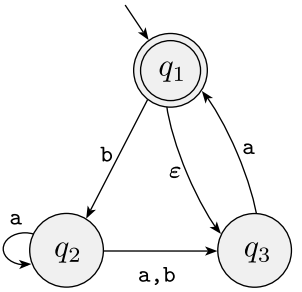


FIGURE 1.36
The NFA N_4

FORMAL DEFINITION OF A NONDETERMINISTIC FINITE AUTOMATON

The formal definition of a nondeterministic finite automaton is similar to that of a deterministic finite automaton. Both have states, an input alphabet, a transition function, a start state, and a collection of accept states. However, they differ in one essential way: in the type of transition function. In a DFA, the transition function takes a state and an input symbol and produces the next state. In an NFA, the transition function takes a state and an input symbol *or the empty string* and produces *the set of possible next states*. In order to write the formal definition, we need to set up some additional notation. For any set Q we write $\mathcal{P}(Q)$ to be the collection of all subsets of Q. Here $\mathcal{P}(Q)$ is called the ***power set*** of Q. For any alphabet Σ we write Σ_ε to be $\Sigma \cup \{\varepsilon\}$. Now we can write the formal description of the type of the transition function in an NFA as $\delta\colon Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$.

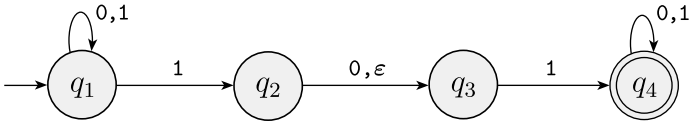
DEFINITION 1.37

A ***nondeterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta\colon Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

EXAMPLE 1.38

Recall the NFA N_1:



The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0,1\}$,
3. δ is given as

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$,

4. q_1 is the start state, and
5. $F = \{q_4\}$.

The formal definition of computation for an NFA is similar to that for a DFA. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and w a string over the alphabet Σ. Then we say that N ***accepts*** w if we can write w as $w = y_1 y_2 \cdots y_m$, where each y_i is a member of Σ_ε and a sequence of states $r_0, r_1, \ldots, r_m$ exists in Q with three conditions:

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, \ldots, m-1$, and
3. $r_m \in F$.

Condition 1 says that the machine starts out in the start state. Condition 2 says that state r_{i+1} is one of the allowable next states when N is in state r_i and reading y_{i+1}. Observe that $\delta(r_i, y_{i+1})$ is the *set* of allowable next states and so we say that r_{i+1} is a member of that set. Finally, condition 3 says that the machine accepts its input if the last state is an accept state.

EQUIVALENCE OF NFAS AND DFAS

Deterministic and nondeterministic finite automata recognize the same class of languages. Such equivalence is both surprising and useful. It is surprising because NFAs appear to have more power than DFAs, so we might expect that NFAs recognize more languages. It is useful because describing an NFA for a given language sometimes is much easier than describing a DFA for that language.

Say that two machines are ***equivalent*** if they recognize the same language.

THEOREM 1.39

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

PROOF IDEA If a language is recognized by an NFA, then we must show the existence of a DFA that also recognizes it. The idea is to convert the NFA into an equivalent DFA that simulates the NFA.

Recall the "reader as automaton" strategy for designing finite automata. How would you simulate the NFA if you were pretending to be a DFA? What do you need to keep track of as the input string is processed? In the examples of NFAs, you kept track of the various branches of the computation by placing a finger on each state that could be active at given points in the input. You updated the simulation by moving, adding, and removing fingers according to the way the NFA operates. All you needed to keep track of was the set of states having fingers on them.

If k is the number of states of the NFA, it has 2^k subsets of states. Each subset corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^k states. Now we need to figure out which will be the start state and accept states of the DFA, and what will be its transition function. We can discuss this more easily after setting up some formal notation.

PROOF Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA recognizing some language A. We construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ recognizing A. Before doing the full construction, let's first consider the easier case wherein N has no ε arrows. Later we take the ε arrows into account.

1. $Q' = \mathcal{P}(Q)$.
   Every state of M is a set of states of N. Recall that $\mathcal{P}(Q)$ is the set of subsets of Q.
2. For $R \in Q'$ and $a \in \Sigma$, let $\delta'(R, a) = \{q \in Q | \ q \in \delta(r, a) \text{ for some } r \in R\}$.
   If R is a state of M, it is also a set of states of N. When M reads a symbol a in state R, it shows where a takes each state in R. Because each state may go to a set of states, we take the union of all these sets. Another way to write this expression is
   $$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).$$[4]
3. $q_0' = \{q_0\}$.
   M starts in the state corresponding to the collection containing just the start state of N.
4. $F' = \{R \in Q' | \ R \text{ contains an accept state of } N\}$.
   The machine M accepts if one of the possible states that N could be in at this point is an accept state.

[4]The notation $\bigcup_{r \in R} \delta(r, a)$ means: the union of the sets $\delta(r, a)$ for each possible r in R.

Now we need to consider the ε arrows. To do so, we set up an extra bit of notation. For any state R of M, we define $E(R)$ to be the collection of states that can be reached from members of R by going only along ε arrows, including the members of R themselves. Formally, for $R \subseteq Q$ let

$$E(R) = \{q |\ q \text{ can be reached from } R \text{ by traveling along 0 or more } \varepsilon \text{ arrows}\}.$$

Then we modify the transition function of M to place additional fingers on all states that can be reached by going along ε arrows after every step. Replacing $\delta(r, a)$ by $E(\delta(r, a))$ achieves this effect. Thus

$$\delta'(R, a) = \{q \in Q |\ q \in E(\delta(r, a)) \text{ for some } r \in R\}.$$

Additionally, we need to modify the start state of M to move the fingers initially to all possible states that can be reached from the start state of N along the ε arrows. Changing $q_0{}'$ to be $E(\{q_0\})$ achieves this effect. We have now completed the construction of the DFA M that simulates the NFA N.

The construction of M obviously works correctly. At every step in the computation of M on an input, it clearly enters a state that corresponds to the subset of states that N could be in at that point. Thus our proof is complete.

Theorem 1.39 states that every NFA can be converted into an equivalent DFA. Thus nondeterministic finite automata give an alternative way of characterizing the regular languages. We state this fact as a corollary of Theorem 1.39.

COROLLARY 1.40

A language is regular if and only if some nondeterministic finite automaton recognizes it.

One direction of the "if and only if" condition states that a language is regular if some NFA recognizes it. Theorem 1.39 shows that any NFA can be converted into an equivalent DFA. Consequently, if an NFA recognizes some language, so does some DFA, and hence the language is regular. The other direction of the "if and only if" condition states that a language is regular only if some NFA recognizes it. That is, if a language is regular, some NFA must be recognizing it. Obviously, this condition is true because a regular language has a DFA recognizing it and any DFA is also an NFA.

EXAMPLE 1.41

Let's illustrate the procedure we gave in the proof of Theorem 1.39 for converting an NFA to a DFA by using the machine N_4 that appears in Example 1.35. For clarity, we have relabeled the states of N_4 to be $\{1, 2, 3\}$. Thus in the formal description of $N_4 = (Q, \{\mathtt{a},\mathtt{b}\}, \delta, 1, \{1\})$, the set of states Q is $\{1, 2, 3\}$ as shown in Figure 1.42.

To construct a DFA D that is equivalent to N_4, we first determine D's states. N_4 has three states, $\{1, 2, 3\}$, so we construct D with eight states, one for each subset of N_4's states. We label each of D's states with the corresponding subset. Thus D's state set is

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

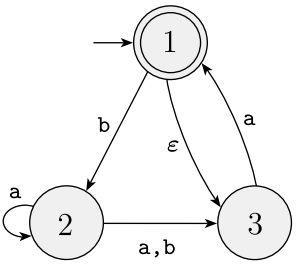


FIGURE 1.42
The NFA N_4

Next, we determine the start and accept states of D. The start state is $E(\{1\})$, the set of states that are reachable from 1 by traveling along ε arrows, plus 1 itself. An ε arrow goes from 1 to 3, so $E(\{1\}) = \{1, 3\}$. The new accept states are those containing N_4's accept state; thus $\{\{1\}, \{1,2\}, \{1,3\}, \{1,2,3\}\}$.

Finally, we determine D's transition function. Each of D's states goes to one place on input `a` and one place on input `b`. We illustrate the process of determining the placement of D's transition arrows with a few examples.

In D, state $\{2\}$ goes to $\{2,3\}$ on input `a` because in N_4, state 2 goes to both 2 and 3 on input `a` and we can't go farther from 2 or 3 along ε arrows. State $\{2\}$ goes to state $\{3\}$ on input `b` because in N_4, state 2 goes only to state 3 on input `b` and we can't go farther from 3 along ε arrows.

State $\{1\}$ goes to $\emptyset$ on `a` because no `a` arrows exit it. It goes to $\{2\}$ on `b`. Note that the procedure in Theorem 1.39 specifies that we follow the ε arrows *after* each input symbol is read. An alternative procedure based on following the ε arrows before reading each input symbol works equally well, but that method is not illustrated in this example.

State $\{3\}$ goes to $\{1,3\}$ on `a` because in N_4, state 3 goes to 1 on `a` and 1 in turn goes to 3 with an ε arrow. State $\{3\}$ on `b` goes to $\emptyset$.

State $\{1,2\}$ on `a` goes to $\{2,3\}$ because 1 points at no states with `a` arrows, 2 points at both 2 and 3 with `a` arrows, and neither points anywhere with ε arrows. State $\{1,2\}$ on `b` goes to $\{2,3\}$. Continuing in this way, we obtain the diagram for D in Figure 1.43.

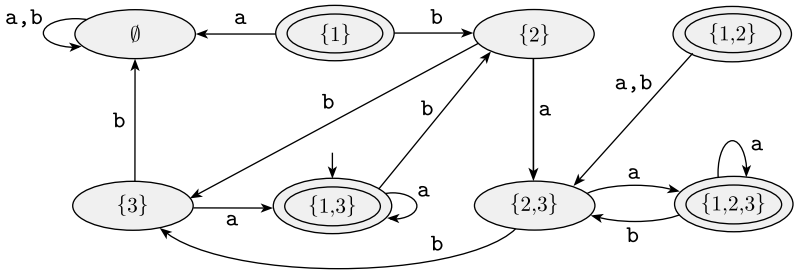


FIGURE 1.43
A DFA D that is equivalent to the NFA N_4

We may simplify this machine by observing that no arrows point at states $\{1\}$ and $\{1, 2\}$, so they may be removed without affecting the performance of the machine. Doing so yields the following figure.

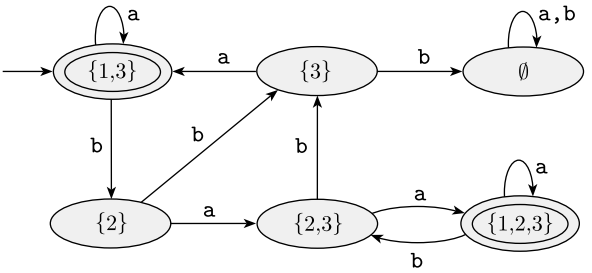


FIGURE 1.44
DFA D after removing unnecessary states

CLOSURE UNDER THE REGULAR OPERATIONS

Now we return to the closure of the class of regular languages under the regular operations that we began in Section 1.1. Our aim is to prove that the union, concatenation, and star of regular languages are still regular. We abandoned the original attempt to do so when dealing with the concatenation operation was too complicated. The use of nondeterminism makes the proofs much easier.

First, let's consider again closure under union. Earlier we proved closure under union by simulating deterministically both machines simultaneously via a Cartesian product construction. We now give a new proof to illustrate the

technique of nondeterminism. Reviewing the first proof, appearing on page 45, may be worthwhile to see how much easier and more intuitive the new proof is.

THEOREM 1.45

The class of regular languages is closed under the union operation.

PROOF IDEA We have regular languages A_1 and A_2 and want to prove that $A_1 \cup A_2$ is regular. The idea is to take two NFAs, N_1 and N_2 for A_1 and A_2, and combine them into one new NFA, N.

Machine N must accept its input if either N_1 or N_2 accepts this input. The new machine has a new start state that branches to the start states of the old machines with ε arrows. In this way, the new machine nondeterministically guesses which of the two machines accepts the input. If one of them accepts the input, N will accept it, too.

We represent this construction in the following figure. On the left, we indicate the start and accept states of machines N_1 and N_2 with large circles and some additional states with small circles. On the right, we show how to combine N_1 and N_2 into N by adding additional transition arrows.

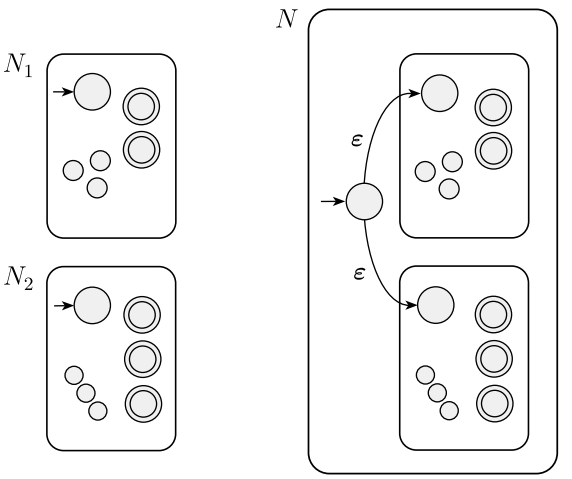


FIGURE 1.46
Construction of an NFA N to recognize $A_1 \cup A_2$

PROOF

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1, and
$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2.

Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \cup A_2$.

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.
   The states of N are all the states of N_1 and N_2, with the addition of a new start state q_0.
2. The state q_0 is the start state of N.
3. The set of accept states $F = F_1 \cup F_2$.
   The accept states of N are all the accept states of N_1 and N_2. That way, N accepts if either N_1 accepts or N_2 accepts.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon. \end{cases}$$

Now we can prove closure under concatenation. Recall that earlier, without nondeterminism, completing the proof would have been difficult.

THEOREM 1.47

The class of regular languages is closed under the concatenation operation.

PROOF IDEA We have regular languages A_1 and A_2 and want to prove that $A_1 \circ A_2$ is regular. The idea is to take two NFAs, N_1 and N_2 for A_1 and A_2, and combine them into a new NFA N as we did for the case of union, but this time in a different way, as shown in Figure 1.48.

Assign N's start state to be the start state of N_1. The accept states of N_1 have additional ε arrows that nondeterministically allow branching to N_2 whenever N_1 is in an accept state, signifying that it has found an initial piece of the input that constitutes a string in A_1. The accept states of N are the accept states of N_2 only. Therefore, it accepts when the input can be split into two parts, the first accepted by N_1 and the second by N_2. We can think of N as nondeterministically guessing where to make the split.

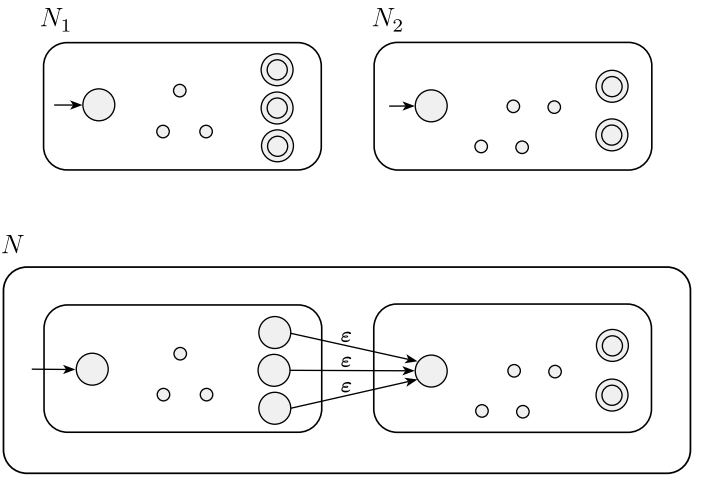


FIGURE 1.48
Construction of N to recognize $A_1 \circ A_2$

PROOF

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1, and
$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2.

Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognize $A_1 \circ A_2$.

1. $Q = Q_1 \cup Q_2$.
   The states of N are all the states of N_1 and N_2.
2. The state q_1 is the same as the start state of N_1.
3. The accept states F_2 are the same as the accept states of N_2.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ and } a = \varepsilon \\ \delta_2(q,a) & q \in Q_2. \end{cases}$$

THEOREM 1.49

The class of regular languages is closed under the star operation.

PROOF IDEA We have a regular language A_1 and want to prove that A_1^* also is regular. We take an NFA N_1 for A_1 and modify it to recognize A_1^*, as shown in the following figure. The resulting NFA N will accept its input whenever it can be broken into several pieces and N_1 accepts each piece.

We can construct N like N_1 with additional ε arrows returning to the start state from the accept states. This way, when processing gets to the end of a piece that N_1 accepts, the machine N has the option of jumping back to the start state to try to read another piece that N_1 accepts. In addition, we must modify N so that it accepts ε, which always is a member of A_1^*. One (slightly bad) idea is simply to add the start state to the set of accept states. This approach certainly adds ε to the recognized language, but it may also add other, undesired strings. Exercise 1.15 asks for an example of the failure of this idea. The way to fix it is to add a new start state, which also is an accept state, and which has an ε arrow to the old start state. This solution has the desired effect of adding ε to the language without adding anything else.

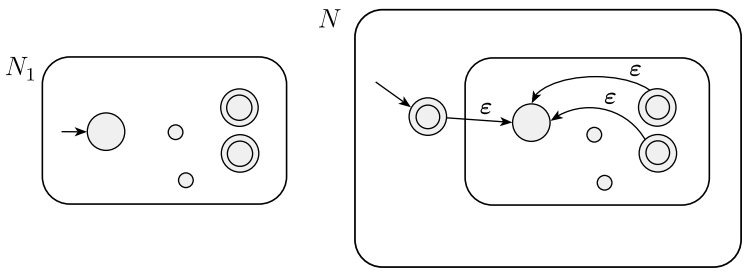


FIGURE 1.50
Construction of N to recognize A^*

PROOF Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1.
Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize A_1^*.

1. $Q = \{q_0\} \cup Q_1$.
   The states of N are the states of N_1 plus a new start state.
2. The state q_0 is the new start state.
3. $F = \{q_0\} \cup F_1$.
   The accept states are the old accept states plus the new start state.

4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon. \end{cases}$$

1.3 REGULAR EXPRESSIONS

In arithmetic, we can use the operations $+$ and $\times$ to build up expressions such as

$$(5+3) \times 4\,.$$

Similarly, we can use the regular operations to build up expressions describing languages, which are called ***regular expressions***. An example is:

$$(\mathtt{0} \cup \mathtt{1})\mathtt{0}^*.$$

The value of the arithmetic expression is the number 32. The value of a regular expression is a language. In this case, the value is the language consisting of all strings starting with a 0 or a 1 followed by any number of 0s. We get this result by dissecting the expression into its parts. First, the symbols 0 and 1 are shorthand for the sets $\{0\}$ and $\{1\}$. So $(0 \cup 1)$ means $(\{0\} \cup \{1\})$. The value of this part is the language $\{0,1\}$. The part 0^* means $\{0\}^*$, and its value is the language consisting of all strings containing any number of 0s. Second, like the $\times$ symbol in algebra, the concatenation symbol $\circ$ often is implicit in regular expressions. Thus $(0 \cup 1)0^*$ actually is shorthand for $(0 \cup 1) \circ 0^*$. The concatenation attaches the strings from the two parts to obtain the value of the entire expression.

Regular expressions have an important role in computer science applications. In applications involving text, users may want to search for strings that satisfy certain patterns. Regular expressions provide a powerful method for describing such patterns. Utilities such as awk and grep in UNIX, modern programming languages such as Perl, and text editors all provide mechanisms for the description of patterns by using regular expressions.

EXAMPLE 1.51

Another example of a regular expression is

$$(\mathtt{0} \cup \mathtt{1})^*.$$

It starts with the language $(\mathtt{0} \cup \mathtt{1})$ and applies the $*$ operation. The value of this expression is the language consisting of all possible strings of 0s and 1s. If $\Sigma = \{\mathtt{0},\mathtt{1}\}$, we can write Σ as shorthand for the regular expression $(\mathtt{0} \cup \mathtt{1})$. More generally, if Σ is any alphabet, the regular expression Σ describes the language consisting of all strings of length 1 over this alphabet, and Σ^* describes the language consisting of all strings over that alphabet. Similarly, $\Sigma^*\mathtt{1}$ is the language that contains all strings that end in a 1. The language $(\mathtt{0}\Sigma^*) \cup (\Sigma^*\mathtt{1})$ consists of all strings that start with a 0 or end with a 1. ■

In arithmetic, we say that $\times$ has precedence over $+$ to mean that when there is a choice, we do the $\times$ operation first. Thus in $2+3\times4$, the 3×4 is done before the addition. To have the addition done first, we must add parentheses to obtain $(2 + 3) \times 4$. In regular expressions, the star operation is done first, followed by concatenation, and finally union, unless parentheses change the usual order.

FORMAL DEFINITION OF A REGULAR EXPRESSION

DEFINITION 1.52

Say that R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

In items 1 and 2, the regular expressions a and ε represent the languages $\{a\}$ and $\{\varepsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the star of the language R_1, respectively.

Don't confuse the regular expressions ε and $\emptyset$. The expression ε represents the language containing a single string—namely, the empty string—whereas $\emptyset$ represents the language that doesn't contain any strings.

Seemingly, we are in danger of defining the notion of a regular expression in terms of itself. If true, we would have a ***circular definition***, which would be invalid. However, R_1 and R_2 always are smaller than R. Thus we actually are defining regular expressions in terms of smaller regular expressions and thereby avoiding circularity. A definition of this type is called an ***inductive definition***.

Parentheses in an expression may be omitted. If they are, evaluation is done in the precedence order: star, then concatenation, then union.

For convenience, we let R^+ be shorthand for RR^*. In other words, whereas R^* has all strings that are 0 or more concatenations of strings from R, the language R^+ has all strings that are 1 or more concatenations of strings from R. So $R^+ \cup \varepsilon = R^*$. In addition, we let R^k be shorthand for the concatenation of k R's with each other.

When we want to distinguish between a regular expression R and the language that it describes, we write $L(R)$ to be the language of R.

EXAMPLE 1.53

In the following instances, we assume that the alphabet Σ is $\{0,1\}$.

1. $0^*10^* = \{w \mid w \text{ contains a single } 1\}$.
2. $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}$.
3. $\Sigma^*001\Sigma^* = \{w \mid w \text{ contains the string } 001 \text{ as a substring}\}$.
4. $1^*(01^+)^* = \{w \mid \text{every } 0 \text{ in } w \text{ is followed by at least one } 1\}$.
5. $(\Sigma\Sigma)^* = \{w \mid w \text{ is a string of even length}\}$.[5]
6. $(\Sigma\Sigma\Sigma)^* = \{w \mid \text{the length of } w \text{ is a multiple of } 3\}$.
7. $01 \cup 10 = \{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w \text{ starts and ends with the same symbol}\}$.
9. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.
   The expression $0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$, so the concatenation operation adds either 0 or ε before every string in 1^*.
10. $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.
11. $1^*\emptyset = \emptyset$.
    Concatenating the empty set to any set yields the empty set.
12. $\emptyset^* = \{\varepsilon\}$.
    The star operation puts together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, giving only the empty string.

[5] The ***length*** of a string is the number of symbols that it contains.

If we let R be any regular expression, we have the following identities. They are good tests of whether you understand the definition.

$R \cup \emptyset = R$.
Adding the empty language to any other language will not change it.

$R \circ \varepsilon = R$.
Joining the empty string to any string will not change it.

However, exchanging $\emptyset$ and ε in the preceding identities may cause the equalities to fail.

$R \cup \varepsilon$ may not equal R.
For example, if $R = \mathtt{0}$, then $L(R) = \{\mathtt{0}\}$ but $L(R \cup \varepsilon) = \{\mathtt{0}, \varepsilon\}$.

$R \circ \emptyset$ may not equal R.
For example, if $R = \mathtt{0}$, then $L(R) = \{\mathtt{0}\}$ but $L(R \circ \emptyset) = \emptyset$.

Regular expressions are useful tools in the design of compilers for programming languages. Elemental objects in a programming language, called ***tokens***, such as the variable names and constants, may be described with regular expressions. For example, a numerical constant that may include a fractional part and/or a sign may be described as a member of the language

$$\left(\mathtt{+} \cup \mathtt{-} \cup \varepsilon\right)\left(D^{\mathtt{+}} \cup D^{\mathtt{+}} . D^{*} \cup D^{*} . D^{\mathtt{+}}\right)$$

where $D = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}\}$ is the alphabet of decimal digits. Examples of generated strings are: `72`, `3.14159`, `+7.`, and `-.01`.

Once the syntax of a programming language has been described with a regular expression in terms of its tokens, automatic systems can generate the ***lexical analyzer***, the part of a compiler that initially processes the input program.

EQUIVALENCE WITH FINITE AUTOMATA

Regular expressions and finite automata are equivalent in their descriptive power. This fact is surprising because finite automata and regular expressions superficially appear to be rather different. However, any regular expression can be converted into a finite automaton that recognizes the language it describes, and vice versa. Recall that a regular language is one that is recognized by some finite automaton.

THEOREM 1.54

A language is regular if and only if some regular expression describes it.

This theorem has two directions. We state and prove each direction as a separate lemma.

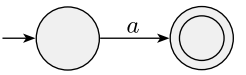
LEMMA 1.55

If a language is described by a regular expression, then it is regular.

PROOF IDEA Say that we have a regular expression R describing some language A. We show how to convert R into an NFA recognizing A. By Corollary 1.40, if an NFA recognizes A then A is regular.

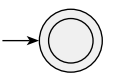
PROOF Let's convert R into an NFA N. We consider the six cases in the formal definition of regular expressions.

1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.

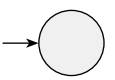


   Note that this machine fits the definition of an NFA but not that of a DFA because it has some states with no exiting arrow for each possible input symbol. Of course, we could have presented an equivalent DFA here; but an NFA is all we need for now, and it is easier to describe.

   Formally, $N = \big(\{q_1, q_2\},\ \Sigma,\ \delta,\ q_1,\ \{q_2\}\big)$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.
2. $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$, and the following NFA recognizes $L(R)$.



   Formally, $N = \big(\{q_1\}, \Sigma, \delta, q_1, \{q_1\}\big)$, where $\delta(r, b) = \emptyset$ for any r and b.
3. $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$.



   Formally, $N = \big(\{q\}, \Sigma, \delta, q, \emptyset\big)$, where $\delta(r, b) = \emptyset$ for any r and b.
4. $R = R_1 \cup R_2$.
5. $R = R_1 \circ R_2$.
6. $R = R_1^*$.

For the last three cases, we use the constructions given in the proofs that the class of regular languages is closed under the regular operations. In other words, we construct the NFA for R from the NFAs for R_1 and R_2 (or just R_1 in case 6) and the appropriate closure construction.

That ends the first part of the proof of Theorem 1.54, giving the easier direction of the if and only if condition. Before going on to the other direction, let's consider some examples whereby we use this procedure to convert a regular expression to an NFA.

EXAMPLE 1.56

We convert the regular expression $(\mathtt{ab} \cup \mathtt{a})^*$ to an NFA in a sequence of stages. We build up from the smallest subexpressions to larger subexpressions until we have an NFA for the original expression, as shown in the following diagram. Note that this procedure generally doesn't give the NFA with the fewest states. In this example, the procedure gives an NFA with eight states, but the smallest equivalent NFA has only two states. Can you find it?

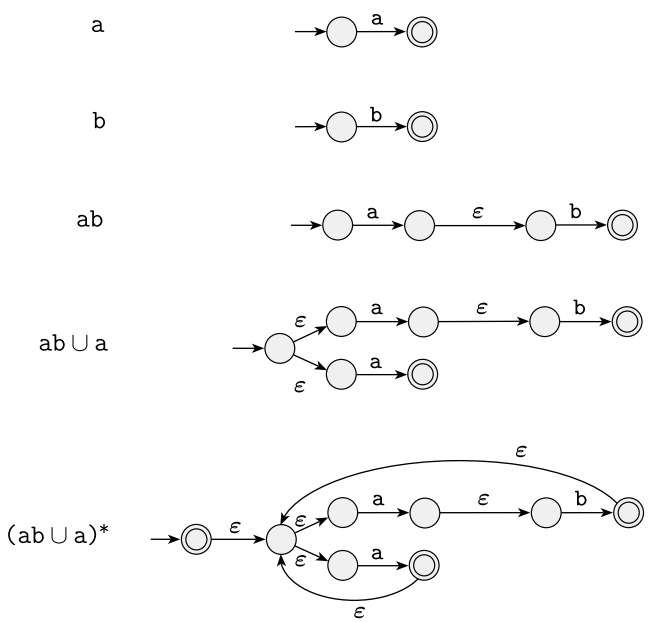


FIGURE 1.57
Building an NFA from the regular expression $(\mathtt{ab} \cup \mathtt{a})^*$

EXAMPLE 1.58

In Figure 1.59, we convert the regular expression $(\mathtt{a} \cup \mathtt{b})^*\mathtt{aba}$ to an NFA. A few of the minor steps are not shown.

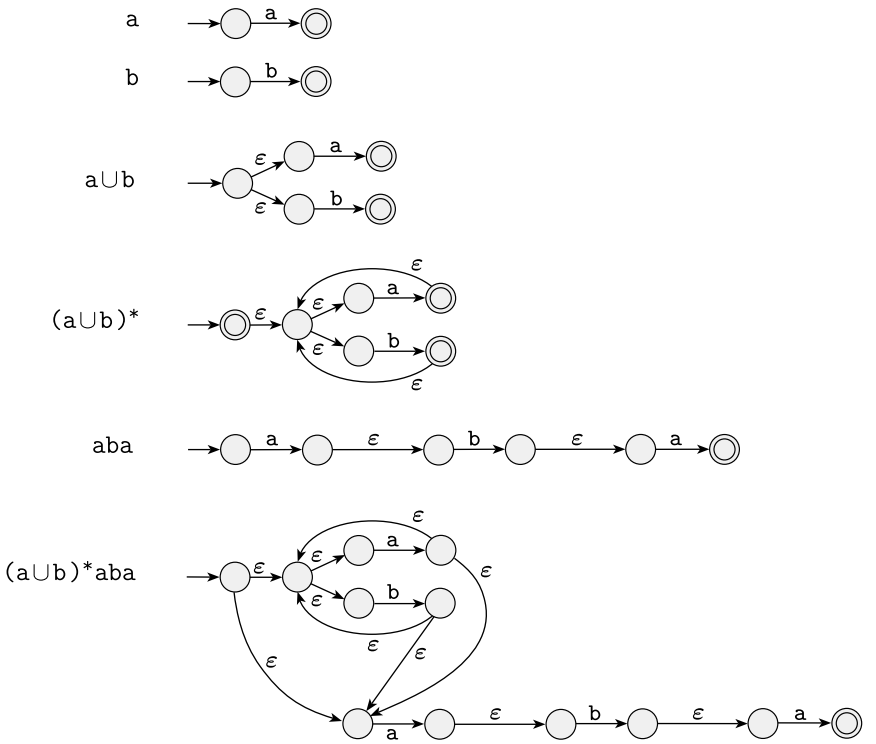


FIGURE 1.59
Building an NFA from the regular expression $(\mathtt{a} \cup \mathtt{b})^*\mathtt{aba}$

Now let's turn to the other direction of the proof of Theorem 1.54.

LEMMA 1.60

If a language is regular, then it is described by a regular expression.

PROOF IDEA We need to show that if a language A is regular, a regular expression describes it. Because A is regular, it is accepted by a DFA. We describe a procedure for converting DFAs into equivalent regular expressions.

We break this procedure into two parts, using a new type of finite automaton called a ***generalized nondeterministic finite automaton***, GNFA. First we show how to convert DFAs into GNFAs, and then GNFAs into regular expressions.

Generalized nondeterministic finite automata are simply nondeterministic finite automata wherein the transition arrows may have any regular expressions as labels, instead of only members of the alphabet or ε. The GNFA reads blocks of symbols from the input, not necessarily just one symbol at a time as in an ordinary NFA. The GNFA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constitute a string described by the regular expression on that arrow. A GNFA is nondeterministic and so may have several different ways to process the same input string. It accepts its input if its processing can cause the GNFA to be in an accept state at the end of the input. The following figure presents an example of a GNFA.

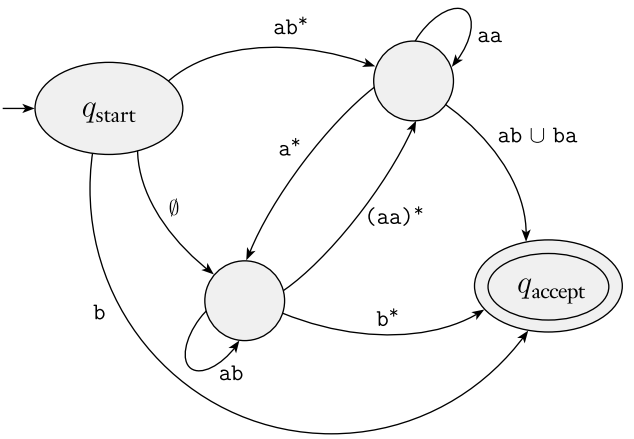


FIGURE 1.61
A generalized nondeterministic finite automaton

For convenience, we require that GNFAs always have a special form that meets the following conditions.

- The start state has transition arrows going to every other state but no arrows coming in from any other state.
- There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
- Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.

We can easily convert a DFA into a GNFA in the special form. We simply add a new start state with an ε arrow to the old start state and a new accept state with ε arrows from the old accept states. If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), we replace each with a single arrow whose label is the union of the previous labels. Finally, we add arrows labeled $\emptyset$ between states that had no arrows. This last step won't change the language recognized because a transition labeled with $\emptyset$ can never be used. From here on we assume that all GNFAs are in the special form.

Now we show how to convert a GNFA into a regular expression. Say that the GNFA has k states. Then, because a GNFA must have a start and an accept state and they must be different from each other, we know that $k \geq 2$. If $k > 2$, we construct an equivalent GNFA with $k - 1$ states. This step can be repeated on the new GNFA until it is reduced to two states. If $k = 2$, the GNFA has a single arrow that goes from the start state to the accept state. The label of this arrow is the equivalent regular expression. For example, the stages in converting a DFA with three states to an equivalent regular expression are shown in the following figure.

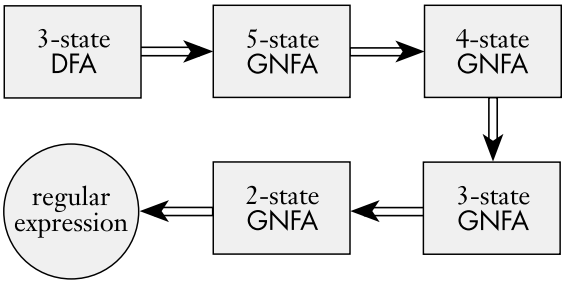


FIGURE 1.62
Typical stages in converting a DFA to a regular expression

The crucial step is constructing an equivalent GNFA with one fewer state when $k > 2$. We do so by selecting a state, ripping it out of the machine, and repairing the remainder so that the same language is still recognized. Any state will do, provided that it is not the start or accept state. We are guaranteed that such a state will exist because $k > 2$. Let's call the removed state q_{rip}.

After removing q_{rip} we repair the machine by altering the regular expressions that label each of the remaining arrows. The new labels compensate for the absence of q_{rip} by adding back the lost computations. The new label going from a state q_i to a state q_j is a regular expression that describes all strings that would

take the machine from q_i to q_j either directly or via q_{rip}. We illustrate this approach in Figure 1.63.

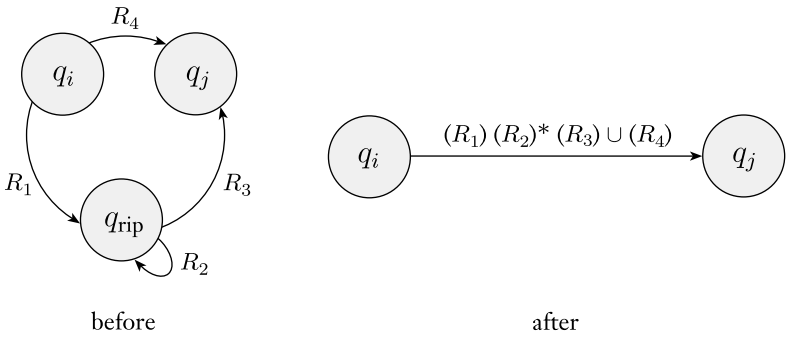


FIGURE 1.63
Constructing an equivalent GNFA with one fewer state

In the old machine, if

1. q_i goes to q_{rip} with an arrow labeled R_1,
2. q_{rip} goes to itself with an arrow labeled R_2,
3. q_{rip} goes to q_j with an arrow labeled R_3, and
4. q_i goes to q_j with an arrow labeled R_4,

then in the new machine, the arrow from q_i to q_j gets the label

$$(R_1)(R_2)^*(R_3) \cup (R_4).$$

We make this change for each arrow going from any state q_i to any state q_j, including the case where $q_i = q_j$. The new machine recognizes the original language.

PROOF Let's now carry out this idea formally. First, to facilitate the proof, we formally define the new type of automaton introduced. A GNFA is similar to a nondeterministic finite automaton except for the transition function, which has the form

$$\delta\colon \big(Q - \{q_{\text{accept}}\}\big) \times \big(Q - \{q_{\text{start}}\}\big) \longrightarrow \mathcal{R}.$$

The symbol $\mathcal{R}$ is the collection of all regular expressions over the alphabet Σ, and q_{start} and q_{accept} are the start and accept states. If $\delta(q_i, q_j) = R$, the arrow from state q_i to state q_j has the regular expression R as its label. The domain of the transition function is $\big(Q - \{q_{\text{accept}}\}\big) \times \big(Q - \{q_{\text{start}}\}\big)$ because an arrow connects every state to every other state, except that no arrows are coming from q_{accept} or going to q_{start}.

DEFINITION 1.64

A ***generalized nondeterministic finite automaton*** is a 5-tuple, $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, where

1. Q is the finite set of states,
2. Σ is the input alphabet,
3. $\delta\colon \big(Q - \{q_{\text{accept}}\}\big) \times \big(Q - \{q_{\text{start}}\}\big) \longrightarrow \mathcal{R}$ is the transition function,
4. q_{start} is the start state, and
5. q_{accept} is the accept state.

A GNFA accepts a string w in Σ^* if $w = w_1 w_2 \cdots w_k$, where each w_i is in Σ^* and a sequence of states $q_0, q_1, \ldots, q_k$ exists such that

1. $q_0 = q_{\text{start}}$ is the start state,
2. $q_k = q_{\text{accept}}$ is the accept state, and
3. for each i, we have $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$; in other words, R_i is the expression on the arrow from q_{i-1} to q_i.

Returning to the proof of Lemma 1.60, we let M be the DFA for language A. Then we convert M to a GNFA G by adding a new start state and a new accept state and additional transition arrows as necessary. We use the procedure CONVERT(G), which takes a GNFA and returns an equivalent regular expression. This procedure uses ***recursion***, which means that it calls itself. An infinite loop is avoided because the procedure calls itself only to process a GNFA that has one fewer state. The case where the GNFA has two states is handled without recursion.

CONVERT(G):

1. Let k be the number of states of G.
2. If $k = 2$, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression R. Return the expression R.
3. If $k > 2$, we select any state $q_{\text{rip}} \in Q$ different from q_{start} and q_{accept} and let G' be the GNFA $(Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$, where

$$Q' = Q - \{q_{\text{rip}}\},$$

and for any $q_i \in Q' - \{q_{\text{accept}}\}$ and any $q_j \in Q' - \{q_{\text{start}}\}$, let

$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4),$$

for $R_1 = \delta(q_i, q_{\text{rip}})$, $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R_3 = \delta(q_{\text{rip}}, q_j)$, and $R_4 = \delta(q_i, q_j)$.

4. Compute CONVERT(G') and return this value.

Next we prove that CONVERT returns a correct value.

CLAIM 1.65

For any GNFA G, CONVERT(G) is equivalent to G.

We prove this claim by induction on k, the number of states of the GNFA.

Basis: Prove the claim true for $k = 2$ states. If G has only two states, it can have only a single arrow, which goes from the start state to the accept state. The regular expression label on this arrow describes all the strings that allow G to get to the accept state. Hence this expression is equivalent to G.

Induction step: Assume that the claim is true for $k - 1$ states and use this assumption to prove that the claim is true for k states. First we show that G and G' recognize the same language. Suppose that G accepts an input w. Then in an accepting branch of the computation, G enters a sequence of states:

$$q_{\text{start}}, q_1, q_2, q_3, \ldots, q_{\text{accept}}.$$

If none of them is the removed state q_{rip}, clearly G' also accepts w. The reason is that each of the new regular expressions labeling the arrows of G' contains the old regular expression as part of a union.

If q_{rip} does appear, removing each run of consecutive q_{rip} states forms an accepting computation for G'. The states q_i and q_j bracketing a run have a new regular expression on the arrow between them that describes all strings taking q_i to q_j via q_{rip} on G. So G' accepts w.

Conversely, suppose that G' accepts an input w. As each arrow between any two states q_i and q_j in G' describes the collection of strings taking q_i to q_j in G, either directly or via q_{rip}, G must also accept w. Thus G and G' are equivalent.

The induction hypothesis states that when the algorithm calls itself recursively on input G', the result is a regular expression that is equivalent to G' because G' has $k - 1$ states. Hence this regular expression also is equivalent to G, and the algorithm is proved correct.

This concludes the proof of Claim 1.65, Lemma 1.60, and Theorem 1.54.

EXAMPLE 1.66

In this example, we use the preceding algorithm to convert a DFA into a regular expression. We begin with the two-state DFA in Figure 1.67(a).

In Figure 1.67(b), we make a four-state GNFA by adding a new start state and a new accept state, called s and a instead of q_{start} and q_{accept} so that we can draw them conveniently. To avoid cluttering up the figure, we do not draw the arrows

labeled $\emptyset$, even though they are present. Note that we replace the label $\mathtt{a}, \mathtt{b}$ on the self-loop at state 2 on the DFA with the label $\mathtt{a} \cup \mathtt{b}$ at the corresponding point on the GNFA. We do so because the DFA's label represents two transitions, one for $\mathtt{a}$ and the other for $\mathtt{b}$, whereas the GNFA may have only a single transition going from 2 to itself.

In Figure 1.67(c), we remove state 2 and update the remaining arrow labels. In this case, the only label that changes is the one from 1 to a. In part (b) it was $\emptyset$, but in part (c) it is $\mathtt{b}(\mathtt{a} \cup \mathtt{b})^*$. We obtain this result by following step 3 of the CONVERT procedure. State q_i is state 1, state q_j is a, and q_{rip} is 2, so $R_1 = \mathtt{b}$, $R_2 = \mathtt{a} \cup \mathtt{b}$, $R_3 = \varepsilon$, and $R_4 = \emptyset$. Therefore, the new label on the arrow from 1 to a is $(\mathtt{b})(\mathtt{a} \cup \mathtt{b})^*(\varepsilon) \cup \emptyset$. We simplify this regular expression to $\mathtt{b}(\mathtt{a} \cup \mathtt{b})^*$.

In Figure 1.67(d), we remove state 1 from part (c) and follow the same procedure. Because only the start and accept states remain, the label on the arrow joining them is the regular expression that is equivalent to the original DFA.

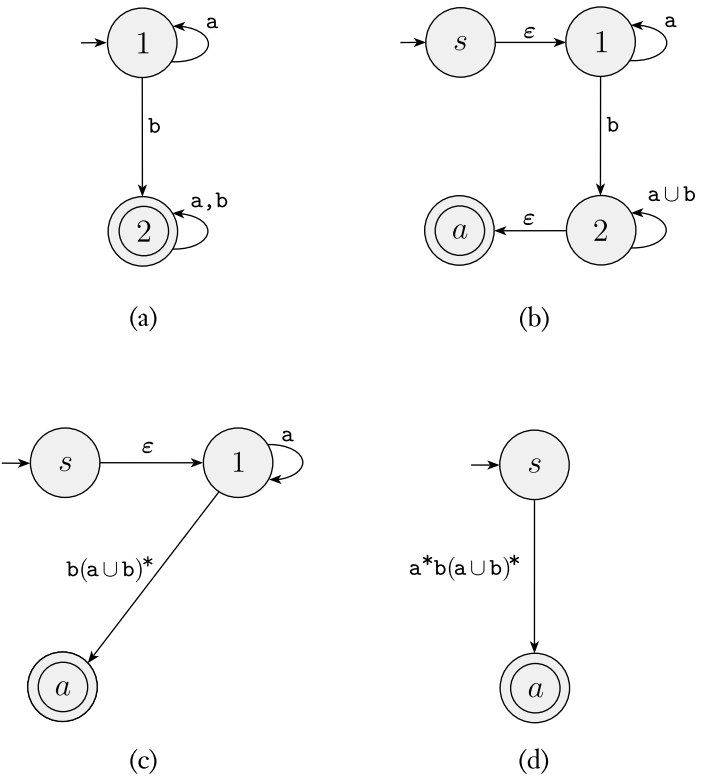


FIGURE 1.67
Converting a two-state DFA to an equivalent regular expression

EXAMPLE 1.68

In this example, we begin with a three-state DFA. The steps in the conversion are shown in the following figure.

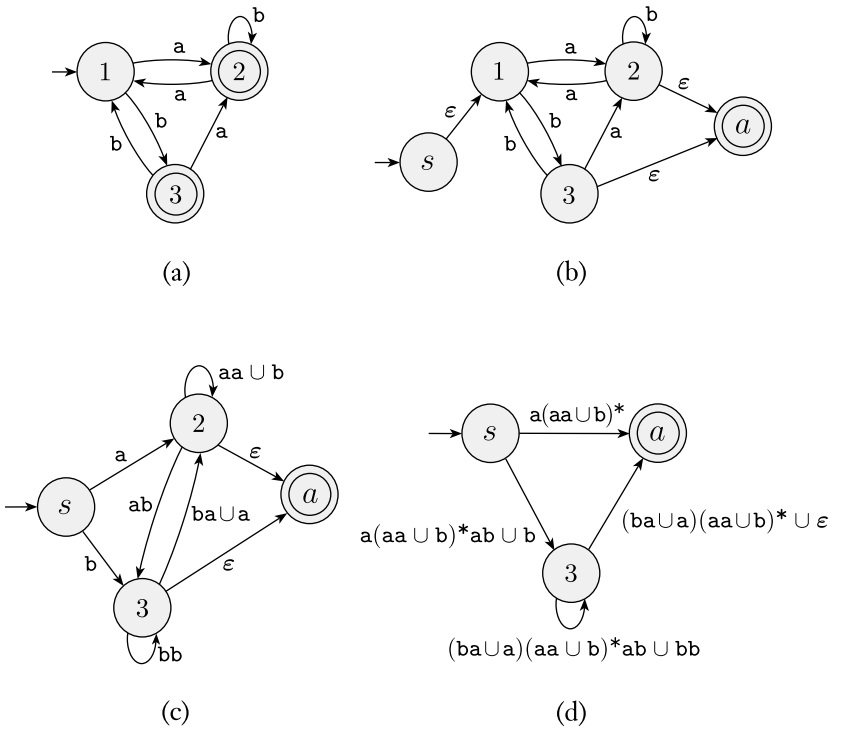


FIGURE 1.69
Converting a three-state DFA to an equivalent regular expression

1.4 NONREGULAR LANGUAGES

To understand the power of finite automata, you must also understand their limitations. In this section, we show how to prove that certain languages cannot be recognized by any finite automaton.

Let's take the language $B = \{0^n1^n | \ n \geq 0\}$. If we attempt to find a DFA that recognizes B, we discover that the machine seems to need to remember how many 0s have been seen so far as it reads the input. Because the number of 0s isn't limited, the machine will have to keep track of an unlimited number of possibilities. But it cannot do so with any finite number of states.

Next, we present a method for proving that languages such as B are not regular. Doesn't the argument already given prove nonregularity because the number of 0s is unlimited? It does not. Just because the language appears to require unbounded memory doesn't mean that it is necessarily so. It does happen to be true for the language B; but other languages seem to require an unlimited number of possibilities, yet actually they are regular. For example, consider two languages over the alphabet $\Sigma = \{0,1\}$:

$C = \{w | \ w$ has an equal number of 0s and 1s$\}$, and

$D = \{w | \ w$ has an equal number of occurrences of 01 and 10 as substrings$\}$.

At first glance, a recognizing machine appears to need to count in each case, and therefore neither language appears to be regular. As expected, C is not regular, but surprisingly D is regular![6] Thus our intuition can sometimes lead us astray, which is why we need mathematical proofs for certainty. In this section, we show how to prove that certain languages are not regular.

THE PUMPING LEMMA FOR REGULAR LANGUAGES

Our technique for proving nonregularity stems from a theorem about regular languages, traditionally called the ***pumping lemma***. This theorem states that all regular languages have a special property. If we can show that a language does not have this property, we are guaranteed that it is not regular. The property states that all strings in the language can be "pumped" if they are at least as long as a certain special value, called the ***pumping length***. That means each such string contains a section that can be repeated any number of times with the resulting string remaining in the language.

[6] See Problem 1.48.

THEOREM 1.70

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Recall the notation where $|s|$ represents the length of string s, y^i means that i copies of y are concatenated together, and y^0 equals ε.

When s is divided into xyz, either x or z may be ε, but condition 2 says that $y \neq \varepsilon$. Observe that without condition 2 the theorem would be trivially true. Condition 3 states that the pieces x and y together have length at most p. It is an extra technical condition that we occasionally find useful when proving certain languages to be nonregular. See Example 1.74 for an application of condition 3.

PROOF IDEA Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA that recognizes A. We assign the pumping length p to be the number of states of M. We show that any string s in A of length at least p may be broken into the three pieces xyz, satisfying our three conditions. What if no strings in A are of length at least p? Then our task is even easier because the theorem becomes *vacuously* true: Obviously the three conditions hold for all strings of length at least p if there aren't any such strings.

If s in A has length at least p, consider the sequence of states that M goes through when computing with input s. It starts with q_1 the start state, then goes to, say, q_3, then, say, q_{20}, then q_9, and so on, until it reaches the end of s in state q_{13}. With s in A, we know that M accepts s, so q_{13} is an accept state.

If we let n be the length of s, the sequence of states $q_1, q_3, q_{20}, q_9, \ldots, q_{13}$ has length $n + 1$. Because n is at least p, we know that $n + 1$ is greater than p, the number of states of M. Therefore, the sequence must contain a repeated state. This result is an example of the ***pigeonhole principle***, a fancy name for the rather obvious fact that if p pigeons are placed into fewer than p holes, some hole has to have more than one pigeon in it.

The following figure shows the string s and the sequence of states that M goes through when processing s. State q_9 is the one that repeats.

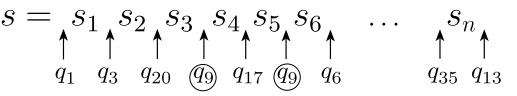


FIGURE 1.71
Example showing state q_9 repeating when M reads s

We now divide s into the three pieces x, y, and z. Piece x is the part of s appearing before q_9, piece y is the part between the two appearances of q_9, and

piece z is the remaining part of s, coming after the second occurrence of q_9. So x takes M from the state q_1 to q_9, y takes M from q_9 back to q_9, and z takes M from q_9 to the accept state q_{13}, as shown in the following figure.

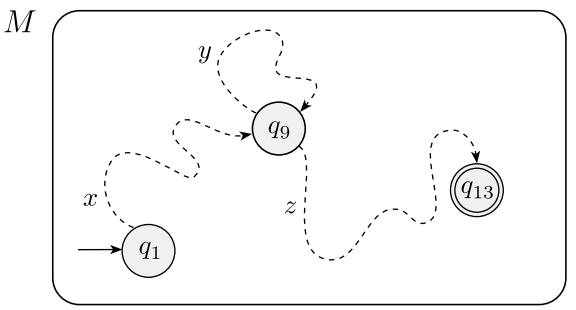


FIGURE 1.72
Example showing how the strings x, y, and z affect M

Let's see why this division of s satisfies the three conditions. Suppose that we run M on input $xyyz$. We know that x takes M from q_1 to q_9, and then the first y takes it from q_9 back to q_9, as does the second y, and then z takes it to q_{13}. With q_{13} being an accept state, M accepts input $xyyz$. Similarly, it will accept xy^iz for any $i > 0$. For the case $i = 0$, $xy^iz = xz$, which is accepted for similar reasons. That establishes condition 1.

Checking condition 2, we see that $|y| > 0$, as it was the part of s that occurred between two different occurrences of state q_9.

In order to get condition 3, we make sure that q_9 is the first repetition in the sequence. By the pigeonhole principle, the first $p+1$ states in the sequence must contain a repetition. Therefore, $|xy| \leq p$.

PROOF Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA recognizing A and p be the number of states of M.

Let $s = s_1 s_2 \cdots s_n$ be a string in A of length n, where $n \geq p$. Let $r_1, \ldots, r_{n+1}$ be the sequence of states that M enters while processing s, so $r_{i+1} = \delta(r_i, s_i)$ for $1 \leq i \leq n$. This sequence has length $n + 1$, which is at least $p + 1$. Among the first $p + 1$ elements in the sequence, two must be the same state, by the pigeonhole principle. We call the first of these r_j and the second r_l. Because r_l occurs among the first $p+1$ places in a sequence starting at r_1, we have $l \leq p+1$. Now let $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{l-1}$, and $z = s_l \cdots s_n$.

As x takes M from r_1 to r_j, y takes M from r_j to r_j, and z takes M from r_j to r_{n+1}, which is an accept state, M must accept xy^iz for $i \geq 0$. We know that $j \neq l$, so $|y| > 0$; and $l \leq p+1$, so $|xy| \leq p$. Thus we have satisfied all conditions of the pumping lemma.

To use the pumping lemma to prove that a language B is not regular, first assume that B is regular in order to obtain a contradiction. Then use the pumping lemma to guarantee the existence of a pumping length p such that all strings of length p or greater in B can be pumped. Next, find a string s in B that has length p or greater but that cannot be pumped. Finally, demonstrate that s cannot be pumped by considering all ways of dividing s into x, y, and z (taking condition 3 of the pumping lemma into account if convenient) and, for each such division, finding a value i where $xy^iz \notin B$. This final step often involves grouping the various ways of dividing s into several cases and analyzing them individually. The existence of s contradicts the pumping lemma if B were regular. Hence B cannot be regular.

Finding s sometimes takes a bit of creative thinking. You may need to hunt through several candidates for s before you discover one that works. Try members of B that seem to exhibit the "essence" of B's nonregularity. We further discuss the task of finding s in some of the following examples.

EXAMPLE 1.73

Let B be the language $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$. We use the pumping lemma to prove that B is not regular. The proof is by contradiction.

Assume to the contrary that B is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string $\mathtt{0}^p\mathtt{1}^p$. Because s is a member of B and s has length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in B. We consider three cases to show that this result is impossible.

1. The string y consists only of 0s. In this case, the string $xyyz$ has more 0s than 1s and so is not a member of B, violating condition 1 of the pumping lemma. This case is a contradiction.
2. The string y consists only of 1s. This case also gives a contradiction.
3. The string y consists of both 0s and 1s. In this case, the string $xyyz$ may have the same number of 0s and 1s, but they will be out of order with some 1s before 0s. Hence it is not a member of B, which is a contradiction.

Thus a contradiction is unavoidable if we make the assumption that B is regular, so B is not regular. Note that we can simplify this argument by applying condition 3 of the pumping lemma to eliminate cases 2 and 3.

In this example, finding the string s was easy because any string in B of length p or more would work. In the next two examples, some choices for s do not work so additional care is required. ■

EXAMPLE 1.74

Let $C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$. We use the pumping lemma to prove that C is not regular. The proof is by contradiction.

Assume to the contrary that C is regular. Let p be the pumping length given by the pumping lemma. As in Example 1.73, let s be the string $\mathtt{0}^p\mathtt{1}^p$. With s being a member of C and having length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in C. We would like to show that this outcome is impossible. But wait, it *is* possible! If we let x and z be the empty string and y be the string $\mathtt{0}^p\mathtt{1}^p$, then xy^iz always has an equal number of 0s and 1s and hence is in C. So it *seems* that s can be pumped.

Here condition 3 in the pumping lemma is useful. It stipulates that when pumping s, it must be divided so that $|xy| \leq p$. That restriction on the way that s may be divided makes it easier to show that the string $s = \mathtt{0}^p\mathtt{1}^p$ we selected cannot be pumped. If $|xy| \leq p$, then y must consist only of 0s, so $xyyz \notin C$. Therefore, s cannot be pumped. That gives us the desired contradiction.

Selecting the string s in this example required more care than in Example 1.73. If we had chosen $s = (\mathtt{01})^p$ instead, we would have run into trouble because we need a string that *cannot* be pumped and that string *can* be pumped, even taking condition 3 into account. Can you see how to pump it? One way to do so sets $x = \varepsilon$, $y = \mathtt{01}$, and $z = (\mathtt{01})^{p-1}$. Then $xy^iz \in C$ for every value of i. If you fail on your first attempt to find a string that cannot be pumped, don't despair. Try another one!

An alternative method of proving that C is nonregular follows from our knowledge that B is nonregular. If C were regular, $C \cap \mathtt{0}^*\mathtt{1}^*$ also would be regular. The reasons are that the language $\mathtt{0}^*\mathtt{1}^*$ is regular and that the class of regular languages is closed under intersection, which we proved in footnote 3 (page 46). But $C \cap \mathtt{0}^*\mathtt{1}^*$ equals B, and we know that B is nonregular from Example 1.73. ■

EXAMPLE 1.75

Let $F = \{ww \mid w \in \{\mathtt{0},\mathtt{1}\}^*\}$. We show that F is nonregular, using the pumping lemma.

Assume to the contrary that F is regular. Let p be the pumping length given by the pumping lemma. Let s be the string $\mathtt{0}^p\mathtt{1}\mathtt{0}^p\mathtt{1}$. Because s is a member of F and s has length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, satisfying the three conditions of the lemma. We show that this outcome is impossible.

Condition 3 is once again crucial because without it we could pump s if we let x and z be the empty string. With condition 3 the proof follows because y must consist only of 0s, so $xyyz \notin F$.

Observe that we chose $s = \mathtt{0}^p\mathtt{1}\mathtt{0}^p\mathtt{1}$ to be a string that exhibits the "essence" of the nonregularity of F, as opposed to, say, the string $\mathtt{0}^p\mathtt{0}^p$. Even though $\mathtt{0}^p\mathtt{0}^p$ is a member of F, it fails to demonstrate a contradiction because it can be pumped. ■

EXAMPLE 1.76

Here we demonstrate a nonregular unary language. Let $D = \{1^{n^2} \mid n \geq 0\}$. In other words, D contains all strings of 1s whose length is a perfect square. We use the pumping lemma to prove that D is not regular. The proof is by contradiction.

Assume to the contrary that D is regular. Let p be the pumping length given by the pumping lemma. Let s be the string 1^{p^2}. Because s is a member of D and s has length at least p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in D. As in the preceding examples, we show that this outcome is impossible. Doing so in this case requires a little thought about the sequence of perfect squares:

$$0, 1, 4, 9, 16, 25, 36, 49, \ldots$$

Note the growing gap between successive members of this sequence. Large members of this sequence cannot be near each other.

Now consider the two strings xyz and xy^2z. These strings differ from each other by a single repetition of y, and consequently their lengths differ by the length of y. By condition 3 of the pumping lemma, $|xy| \leq p$ and thus $|y| \leq p$. We have $|xyz| = p^2$ and so $|xy^2z| \leq p^2 + p$. But $p^2 + p < p^2 + 2p + 1 = (p+1)^2$. Moreover, condition 2 implies that y is not the empty string and so $|xy^2z| > p^2$. Therefore, the length of xy^2z lies strictly between the consecutive perfect squares p^2 and $(p+1)^2$. Hence this length cannot be a perfect square itself. So we arrive at the contradiction $xy^2z \notin D$ and conclude that D is not regular. ∎

EXAMPLE 1.77

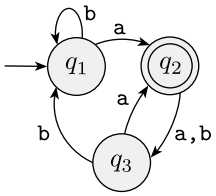
Sometimes "pumping down" is useful when we apply the pumping lemma. We use the pumping lemma to show that $E = \{0^i1^j \mid i > j\}$ is not regular. The proof is by contradiction.

Assume that E is regular. Let p be the pumping length for E given by the pumping lemma. Let $s = 0^{p+1}1^p$. Then s can be split into xyz, satisfying the conditions of the pumping lemma. By condition 3, y consists only of 0s. Let's examine the string $xyyz$ to see whether it can be in E. Adding an extra copy of y increases the number of 0s. But, E contains all strings in 0^*1^* that have more 0s than 1s, so increasing the number of 0s will still give a string in E. No contradiction occurs. We need to try something else.

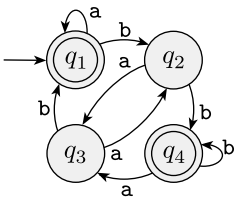
The pumping lemma states that $xy^iz \in E$ even when $i = 0$, so let's consider the string $xy^0z = xz$. Removing string y decreases the number of 0s in s. Recall that s has just one more 0 than 1. Therefore, xz cannot have more 0s than 1s, so it cannot be a member of E. Thus we obtain a contradiction. ∎

EXERCISES

[A]**1.1** The following are the state diagrams of two DFAs, M_1 and M_2. Answer the following questions about each of these machines.



M_1



M_2

a. What is the start state?
b. What is the set of accept states?
c. What sequence of states does the machine go through on input aabb?
d. Does the machine accept the string aabb?
e. Does the machine accept the string ε?

[A]**1.2** Give the formal description of the machines M_1 and M_2 pictured in Exercise 1.1.

1.3 The formal description of a DFA M is $\big(\{q_1, q_2, q_3, q_4, q_5\}, \{\mathtt{u}, \mathtt{d}\}, \delta, q_3, \{q_3\}\big)$, where δ is given by the following table. Give the state diagram of this machine.

	u	d
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_2	q_4
q_4	q_3	q_5
q_5	q_4	q_5

1.4 Each of the following languages is the intersection of two simpler languages. In each part, construct DFAs for the simpler languages, then combine them using the construction discussed in footnote 3 (page 46) to give the state diagram of a DFA for the language given. In all parts, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

a. $\{w \mid w$ has at least three a's and at least two b's$\}$
[A]**b.** $\{w \mid w$ has exactly two a's and at least two b's$\}$
c. $\{w \mid w$ has an even number of a's and one or two b's$\}$
[A]**d.** $\{w \mid w$ has an even number of a's and each a is followed by at least one b$\}$
e. $\{w \mid w$ starts with an a and has at most one b$\}$
f. $\{w \mid w$ has an odd number of a's and ends with a b$\}$
g. $\{w \mid w$ has even length and an odd number of a's$\}$

1.5 Each of the following languages is the complement of a simpler language. In each part, construct a DFA for the simpler language, then use it to give the state diagram of a DFA for the language given. In all parts, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

[A] **a.** $\{w \mid w$ does not contain the substring $\mathtt{ab}\}$

[A] **b.** $\{w \mid w$ does not contain the substring $\mathtt{baba}\}$

c. $\{w \mid w$ contains neither the substrings $\mathtt{ab}$ nor $\mathtt{ba}\}$

d. $\{w \mid w$ is any string not in $\mathtt{a}^*\mathtt{b}^*\}$

e. $\{w \mid w$ is any string not in $(\mathtt{ab}^+)^*\}$

f. $\{w \mid w$ is any string not in $\mathtt{a}^* \cup \mathtt{b}^*\}$

g. $\{w \mid w$ is any string that doesn't contain exactly two $\mathtt{a}$'s$\}$

h. $\{w \mid w$ is any string except $\mathtt{a}$ and $\mathtt{b}\}$

1.6 Give state diagrams of DFAs recognizing the following languages. In all parts, the alphabet is $\{\mathtt{0},\mathtt{1}\}$.

a. $\{w \mid w$ begins with a $\mathtt{1}$ and ends with a $\mathtt{0}\}$

b. $\{w \mid w$ contains at least three $\mathtt{1}$s$\}$

c. $\{w \mid w$ contains the substring $\mathtt{0101}$ (i.e., $w = x\mathtt{0101}y$ for some x and y)$\}$

d. $\{w \mid w$ has length at least 3 and its third symbol is a $\mathtt{0}\}$

e. $\{w \mid w$ starts with $\mathtt{0}$ and has odd length, or starts with $\mathtt{1}$ and has even length$\}$

f. $\{w \mid w$ doesn't contain the substring $\mathtt{110}\}$

g. $\{w \mid$ the length of w is at most $5\}$

h. $\{w \mid w$ is any string except $\mathtt{11}$ and $\mathtt{111}\}$

i. $\{w \mid$ every odd position of w is a $\mathtt{1}\}$

j. $\{w \mid w$ contains at least two $\mathtt{0}$s and at most one $\mathtt{1}\}$

k. $\{\varepsilon, \mathtt{0}\}$

l. $\{w \mid w$ contains an even number of $\mathtt{0}$s, or contains exactly two $\mathtt{1}$s$\}$

m. The empty set

n. All strings except the empty string

1.7 Give state diagrams of NFAs with the specified number of states recognizing each of the following languages. In all parts, the alphabet is $\{\mathtt{0},\mathtt{1}\}$.

[A] **a.** The language $\{w \mid w$ ends with $\mathtt{00}\}$ with three states

b. The language of Exercise 1.6c with five states

c. The language of Exercise 1.6l with six states

d. The language $\{\mathtt{0}\}$ with two states

e. The language $\mathtt{0}^*\mathtt{1}^*\mathtt{0}^+$ with three states

[A] **f.** The language $\mathtt{1}^*(\mathtt{001}^+)^*$ with three states

g. The language $\{\varepsilon\}$ with one state

h. The language $\mathtt{0}^*$ with one state

1.8 Use the construction in the proof of Theorem 1.45 to give the state diagrams of NFAs recognizing the union of the languages described in

a. Exercises 1.6a and 1.6b.

b. Exercises 1.6c and 1.6f.

1.9 Use the construction in the proof of Theorem 1.47 to give the state diagrams of NFAs recognizing the concatenation of the languages described in

- **a.** Exercises 1.6g and 1.6i.
- **b.** Exercises 1.6b and 1.6m.

1.10 Use the construction in the proof of Theorem 1.49 to give the state diagrams of NFAs recognizing the star of the languages described in

- **a.** Exercise 1.6b.
- **b.** Exercise 1.6j.
- **c.** Exercise 1.6m.

[A]**1.11** Prove that every NFA can be converted to an equivalent one that has a single accept state.

1.12 Let $D = \{w \mid w$ contains an even number of `a`'s and an odd number of `b`'s and does not contain the substring `ab`$\}$. Give a DFA with five states that recognizes D and a regular expression that generates D. (Suggestion: Describe D more simply.)

1.13 Let F be the language of all strings over $\{$`0`,`1`$\}$ that do not contain a pair of `1`s that are separated by an odd number of symbols. Give the state diagram of a DFA with five states that recognizes F. (You may find it helpful first to find a 4-state NFA for the complement of F.)

1.14

- **a.** Show that if M is a DFA that recognizes language B, swapping the accept and nonaccept states in M yields a new DFA recognizing the complement of B. Conclude that the class of regular languages is closed under complement.
- **b.** Show by giving an example that if M is an NFA that recognizes language C, swapping the accept and nonaccept states in M doesn't necessarily yield a new NFA that recognizes the complement of C. Is the class of languages recognized by NFAs closed under complement? Explain your answer.

1.15 Give a counterexample to show that the following construction fails to prove Theorem 1.49, the closure of the class of regular languages under the star operation.[7] Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1. Construct $N = (Q_1, \Sigma, \delta, q_1, F)$ as follows. N is supposed to recognize A_1^*.

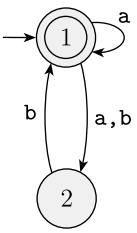
- **a.** The states of N are the states of N_1.
- **b.** The start state of N is the same as the start state of N_1.
- **c.** $F = \{q_1\} \cup F_1$.
  The accept states F are the old accept states plus its start state.
- **d.** Define δ so that for any $q \in Q_1$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \notin F_1 \text{ or } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \varepsilon. \end{cases}$$

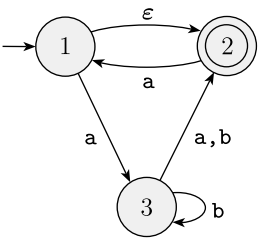
(Suggestion: Show this construction graphically, as in Figure 1.50.)

[7] In other words, you must present a finite automaton, N_1, for which the constructed automaton N does not recognize the star of N_1's language.

1.16 Use the construction given in Theorem 1.39 to convert the following two nondeterministic finite automata to equivalent deterministic finite automata.



(a)



(b)

1.17
a. Give an NFA recognizing the language $(\mathtt{01} \cup \mathtt{001} \cup \mathtt{010})^*$.
b. Convert this NFA to an equivalent DFA. Give only the portion of the DFA that is reachable from the start state.

1.18 Give regular expressions generating the languages of Exercise 1.6.

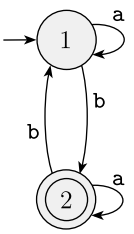
1.19 Use the procedure described in Lemma 1.55 to convert the following regular expressions to nondeterministic finite automata.

a. $(\mathtt{0} \cup \mathtt{1})^*\mathtt{000}(\mathtt{0} \cup \mathtt{1})^*$
b. $(((\mathtt{00})^*(\mathtt{11})) \cup \mathtt{01})^*$
c. $\emptyset^*$

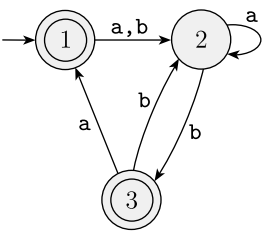
1.20 For each of the following languages, give two strings that are members and two strings that are *not* members—a total of four strings for each part. Assume the alphabet $\Sigma = \{\mathtt{a},\mathtt{b}\}$ in all parts.

a. $\mathtt{a}^*\mathtt{b}^*$
b. $\mathtt{a}(\mathtt{ba})^*\mathtt{b}$
c. $\mathtt{a}^* \cup \mathtt{b}^*$
d. $(\mathtt{aaa})^*$
e. $\Sigma^*\mathtt{a}\Sigma^*\mathtt{b}\Sigma^*\mathtt{a}\Sigma^*$
f. $\mathtt{aba} \cup \mathtt{bab}$
g. $(\varepsilon \cup \mathtt{a})\mathtt{b}$
h. $(\mathtt{a} \cup \mathtt{ba} \cup \mathtt{bb})\Sigma^*$

1.21 Use the procedure described in Lemma 1.60 to convert the following finite automata to regular expressions.



(a)



(b)

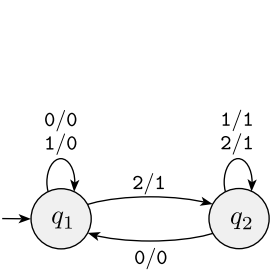
1.22 In certain programming languages, comments appear between delimiters such as `/#` and `#/`. Let C be the language of all valid delimited comment strings. A member of C must begin with `/#` and end with `#/` but have no intervening `#/`. For simplicity, assume that the alphabet for C is $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{/}, \texttt{\#}\}$.

a. Give a DFA that recognizes C.

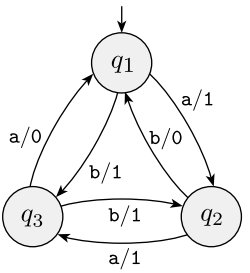
b. Give a regular expression that generates C.

A**1.23** Let B be any language over the alphabet Σ. Prove that $B = B^+$ iff $BB \subseteq B$.

1.24 A ***finite state transducer*** (FST) is a type of deterministic finite automaton whose output is a string and not just *accept* or *reject*. The following are state diagrams of finite state transducers T_1 and T_2.



T_1



T_2

Each transition of an FST is labeled with two symbols, one designating the input symbol for that transition and the other designating the output symbol. The two symbols are written with a slash, /, separating them. In T_1, the transition from q_1 to q_2 has input symbol `2` and output symbol `1`. Some transitions may have multiple input–output pairs, such as the transition in T_1 from q_1 to itself. When an FST computes on an input string w, it takes the input symbols $w_1 \cdots w_n$ one by one and, starting at the start state, follows the transitions by matching the input labels with the sequence of symbols $w_1 \cdots w_n = w$. Every time it goes along a transition, it outputs the corresponding output symbol. For example, on input `2212011`, machine T_1 enters the sequence of states $q_1, q_2, q_2, q_2, q_2, q_1, q_1, q_1$ and produces output `1111000`. On input `abbb`, T_2 outputs `1011`. Give the sequence of states entered and the output produced in each of the following parts.

a. T_1 on input `011`

b. T_1 on input `211`

c. T_1 on input `121`

d. T_1 on input `0202`

e. T_2 on input `b`

f. T_2 on input `bbab`

g. T_2 on input `bbbbbb`

h. T_2 on input ε

1.25 Read the informal definition of the finite state transducer given in Exercise 1.24. Give a formal definition of this model, following the pattern in Definition 1.5 (page 35). Assume that an FST has an input alphabet Σ and an output alphabet Γ but not a set of accept states. Include a formal definition of the computation of an FST. (Hint: An FST is a 5-tuple. Its transition function is of the form $\delta\colon Q \times \Sigma \longrightarrow Q \times \Gamma$.)

1.26 Using the solution you gave to Exercise 1.25, give a formal description of the machines T_1 and T_2 depicted in Exercise 1.24.

1.27 Read the informal definition of the finite state transducer given in Exercise 1.24. Give the state diagram of an FST with the following behavior. Its input and output alphabets are {0,1}. Its output string is identical to the input string on the even positions but inverted on the odd positions. For example, on input 0000111 it should output 1010010.

1.28 Convert the following regular expressions to NFAs using the procedure given in Theorem 1.54. In all parts, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

a. $\mathtt{a}(\mathtt{abb})^* \cup \mathtt{b}$

b. $\mathtt{a}^+ \cup (\mathtt{ab})^+$

c. $(\mathtt{a} \cup \mathtt{b}^+)\mathtt{a}^+\mathtt{b}^+$

1.29 Use the pumping lemma to show that the following languages are not regular.

A a. $A_1 = \{\mathtt{0}^n\mathtt{1}^n\mathtt{2}^n \mid n \geq 0\}$

b. $A_2 = \{www \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$

A c. $A_3 = \{\mathtt{a}^{2^n} \mid n \geq 0\}$ (Here, $\mathtt{a}^{2^n}$ means a string of 2^n a's.)

1.30 Describe the error in the following "proof" that $\mathtt{0}^*\mathtt{1}^*$ is not a regular language. (An error must exist because $\mathtt{0}^*\mathtt{1}^*$ *is* regular.) The proof is by contradiction. Assume that $\mathtt{0}^*\mathtt{1}^*$ is regular. Let p be the pumping length for $\mathtt{0}^*\mathtt{1}^*$ given by the pumping lemma. Choose s to be the string $\mathtt{0}^p\mathtt{1}^p$. You know that s is a member of $\mathtt{0}^*\mathtt{1}^*$, but Example 1.73 shows that s cannot be pumped. Thus you have a contradiction. So $\mathtt{0}^*\mathtt{1}^*$ is not regular.

PROBLEMS

1.31 For any string $w = w_1w_2 \cdots w_n$, the ***reverse*** of w, written $w^{\mathcal{R}}$, is the string w in reverse order, $w_n \cdots w_2w_1$. For any language A, let $A^{\mathcal{R}} = \{w^{\mathcal{R}} \mid w \in A\}$. Show that if A is regular, so is $A^{\mathcal{R}}$.

1.32 Let

$$\Sigma_3 = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}.$$

Σ_3 contains all size 3 columns of 0s and 1s. A string of symbols in Σ_3 gives three rows of 0s and 1s. Consider each row to be a binary number and let

$$B = \{w \in \Sigma_3^* \mid \text{the bottom row of } w \text{ is the sum of the top two rows}\}.$$

For example,

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \in B, \quad \text{but} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \notin B.$$

Show that B is regular. (Hint: Working with $B^{\mathcal{R}}$ is easier. You may assume the result claimed in Problem 1.31.)

1.33 Let

$$\Sigma_2 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}.$$

Here, Σ_2 contains all columns of 0s and 1s of height two. A string of symbols in Σ_2 gives two rows of 0s and 1s. Consider each row to be a binary number and let

$$C = \{w \in \Sigma_2^* | \text{ the bottom row of } w \text{ is three times the top row}\}.$$

For example, $\begin{bmatrix} 0 \\ 0 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix} \in C$, but $\begin{bmatrix} 0 \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} \notin C$. Show that C is regular. (You may assume the result claimed in Problem 1.31.)

1.34 Let Σ_2 be the same as in Problem 1.33. Consider each row to be a binary number and let

$$D = \{w \in \Sigma_2^* | \text{ the top row of } w \text{ is a larger number than is the bottom row}\}.$$

For example, $\begin{bmatrix} 0 \\ 0 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix} \in D$, but $\begin{bmatrix} 0 \\ 0 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix} \notin D$. Show that D is regular.

1.35 Let Σ_2 be the same as in Problem 1.33. Consider the top and bottom rows to be strings of 0s and 1s, and let

$$E = \{w \in \Sigma_2^* | \text{ the bottom row of } w \text{ is the reverse of the top row of } w\}.$$

Show that E is not regular.

1.36 Let $B_n = \{\mathtt{a}^k | \ k \text{ is a multiple of } n\}$. Show that for each $n \geq 1$, the language B_n is regular.

1.37 Let $C_n = \{x | \ x \text{ is a binary number that is a multiple of } n\}$. Show that for each $n \geq 1$, the language C_n is regular.

1.38 An ***all-***NFA M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ that accepts $x \in \Sigma^*$ if *every* possible state that M could be in after reading input x is a state from F. Note, in contrast, that an ordinary NFA accepts a string if *some* state among these possible states is an accept state. Prove that all-NFAs recognize the class of regular languages.

1.39 The construction in Theorem 1.54 shows that every GNFA is equivalent to a GNFA with only two states. We can show that an opposite phenomenon occurs for DFAs. Prove that for every $k > 1$, a language $A_k \subseteq \{0,1\}^*$ exists that is recognized by a DFA with k states but not by one with only $k - 1$ states.

1.40 Recall that string x is a ***prefix*** of string y if a string z exists where $xz = y$, and that x is a ***proper prefix*** of y if in addition $x \neq y$. In each of the following parts, we define an operation on a language A. Show that the class of regular languages is closed under that operation.

a. $NOPREFIX(A) = \{w \in A | \text{ no proper prefix of } w \text{ is a member of } A\}$.

b. $NOEXTEND(A) = \{w \in A | \ w \text{ is not the proper prefix of any string in } A\}$.

1.41 For languages A and B, let the ***perfect shuffle*** of A and B be the language

$$\{w | \ w = a_1 b_1 \cdots a_k b_k, \text{ where } a_1 \cdots a_k \in A \text{ and } b_1 \cdots b_k \in B, \text{ each } a_i, b_i \in \Sigma\}.$$

Show that the class of regular languages is closed under perfect shuffle.

1.42 For languages A and B, let the ***shuffle*** of A and B be the language

$$\{w | \ w = a_1 b_1 \cdots a_k b_k, \text{ where } a_1 \cdots a_k \in A \text{ and } b_1 \cdots b_k \in B, \text{ each } a_i, b_i \in \Sigma^*\}.$$

Show that the class of regular languages is closed under shuffle.

1.43 Let A be any language. Define *DROP-OUT*(A) to be the language containing all strings that can be obtained by removing one symbol from a string in A. Thus, *DROP-OUT*$(A) = \{xz \mid xyz \in A \text{ where } x, z \in \Sigma^*, y \in \Sigma\}$. Show that the class of regular languages is closed under the *DROP-OUT* operation. Give both a proof by picture and a more formal proof by construction as in Theorem 1.47.

A**1.44** Let B and C be languages over $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. Define

$$B \stackrel{1}{\leftarrow} C = \{w \in B \mid \text{ for some } y \in C, \text{ strings } w \text{ and } y \text{ contain equal numbers of } \mathtt{1}\mathrm{s}\}.$$

Show that the class of regular languages is closed under the $\stackrel{1}{\leftarrow}$ operation.

***1.45** Let $A/B = \{w \mid wx \in A \text{ for some } x \in B\}$. Show that if A is regular and B is any language, then A/B is regular.

1.46 Prove that the following languages are not regular. You may use the pumping lemma and the closure of the class of regular languages under union, intersection, and complement.

a. $\{\mathtt{0}^n\mathtt{1}^m\mathtt{0}^n \mid m, n \geq 0\}$

A**b.** $\{\mathtt{0}^m\mathtt{1}^n \mid m \neq n\}$

c. $\{w \mid w \in \{\mathtt{0},\mathtt{1}\}^* \text{ is not a palindrome}\}$[8]

***d.** $\{wtw \mid w, t \in \{\mathtt{0},\mathtt{1}\}^+\}$

1.47 Let $\Sigma = \{\mathtt{1}, \mathtt{\#}\}$ and let

$$Y = \{w \mid w = x_1\mathtt{\#}x_2\mathtt{\#}\cdots\mathtt{\#}x_k \text{ for } k \geq 0, \text{ each } x_i \in \mathtt{1}^*, \text{ and } x_i \neq x_j \text{ for } i \neq j\}.$$

Prove that Y is not regular.

1.48 Let $\Sigma = \{\mathtt{0},\mathtt{1}\}$ and let

$$D = \{w \mid w \text{ contains an equal number of occurrences of the substrings } \mathtt{01} \text{ and } \mathtt{10}\}.$$

Thus $\mathtt{101} \in D$ because $\mathtt{101}$ contains a single $\mathtt{01}$ and a single $\mathtt{10}$, but $\mathtt{1010} \notin D$ because $\mathtt{1010}$ contains two $\mathtt{10}$s and one $\mathtt{01}$. Show that D is a regular language.

1.49 **a.** Let $B = \{\mathtt{1}^k y \mid y \in \{\mathtt{0},\mathtt{1}\}^* \text{ and } y \text{ contains at least } k \text{ } \mathtt{1}\mathrm{s}, \text{ for } k \geq 1\}$. Show that B is a regular language.

b. Let $C = \{\mathtt{1}^k y \mid y \in \{\mathtt{0},\mathtt{1}\}^* \text{ and } y \text{ contains at most } k \text{ } \mathtt{1}\mathrm{s}, \text{ for } k \geq 1\}$. Show that C isn't a regular language.

A**1.50** Read the informal definition of the finite state transducer given in Exercise 1.24. Prove that no FST can output $w^{\mathcal{R}}$ for every input w if the input and output alphabets are $\{\mathtt{0},\mathtt{1}\}$.

1.51 Let x and y be strings and let L be any language. We say that x and y are ***distinguishable by L*** if some string z exists whereby exactly one of the strings xz and yz is a member of L; otherwise, for every string z, we have $xz \in L$ whenever $yz \in L$ and we say that x and y are ***indistinguishable by L***. If x and y are indistinguishable by L, we write $x \equiv_L y$. Show that $\equiv_L$ is an equivalence relation.

[8] A ***palindrome*** is a string that reads the same forward and backward.

A* **1.52** **Myhill–Nerode theorem.** Refer to Problem 1.51. Let L be a language and let X be a set of strings. Say that X is ***pairwise distinguishable by L*** if every two distinct strings in X are distinguishable by L. Define the ***index of L*** to be the maximum number of elements in any set that is pairwise distinguishable by L. The index of L may be finite or infinite.

- **a.** Show that if L is recognized by a DFA with k states, L has index at most k.
- **b.** Show that if the index of L is a finite number k, it is recognized by a DFA with k states.
- **c.** Conclude that L is regular iff it has finite index. Moreover, its index is the size of the smallest DFA recognizing it.

1.53 Let $\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{+}, \mathtt{=}\}$ and

$$ADD = \{x\mathtt{=}y\mathtt{+}z \mid x, y, z \text{ are binary integers, and } x \text{ is the sum of } y \text{ and } z\}.$$

Show that ADD is not regular.

1.54 Consider the language $F = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j, k \geq 0 \text{ and if } i = 1 \text{ then } j = k\}$.

- **a.** Show that F is not regular.
- **b.** Show that F acts like a regular language in the pumping lemma. In other words, give a pumping length p and demonstrate that F satisfies the three conditions of the pumping lemma for this value of p.
- **c.** Explain why parts (a) and (b) do not contradict the pumping lemma.

1.55 The pumping lemma says that every regular language has a pumping length p, such that every string in the language can be pumped if it has length p or more. If p is a pumping length for language A, so is any length $p' \geq p$. The ***minimum pumping length*** for A is the smallest p that is a pumping length for A. For example, if $A = \mathtt{01}^*$, the minimum pumping length is 2. The reason is that the string $s = \mathtt{0}$ is in A and has length 1 yet s cannot be pumped; but any string in A of length 2 or more contains a $\mathtt{1}$ and hence can be pumped by dividing it so that $x = \mathtt{0}$, $y = \mathtt{1}$, and z is the rest. For each of the following languages, give the minimum pumping length and justify your answer.

A **a.** $\mathtt{0001}^*$
A **b.** $\mathtt{0}^*\mathtt{1}^*$
c. $\mathtt{001} \cup \mathtt{0}^*\mathtt{1}^*$
A **d.** $\mathtt{0}^*\mathtt{1}^+\mathtt{0}^+\mathtt{1}^* \cup \mathtt{10}^*\mathtt{1}$
e. $(\mathtt{01})^*$
f. ε
g. $\mathtt{1}^*\mathtt{01}^*\mathtt{01}^*$
h. $\mathtt{10}(\mathtt{11}^*\mathtt{0})^*\mathtt{0}$
i. $\mathtt{1011}$
j. Σ^*

* **1.56** If A is a set of natural numbers and k is a natural number greater than 1, let

$$B_k(A) = \{w \mid w \text{ is the representation in base } k \text{ of some number in } A\}.$$

Here, we do not allow leading $\mathtt{0}$s in the representation of a number. For example, $B_2(\{3, 5\}) = \{\mathtt{11}, \mathtt{101}\}$ and $B_3(\{3, 5\}) = \{\mathtt{10}, \mathtt{12}\}$. Give an example of a set A for which $B_2(A)$ is regular but $B_3(A)$ is not regular. Prove that your example works.

***1.57** If A is any language, let $A_{\frac{1}{2}-}$ be the set of all first halves of strings in A so that

$$A_{\frac{1}{2}-} = \{x \mid \text{ for some } y,\ |x| = |y| \text{ and } xy \in A\}.$$

Show that if A is regular, then so is $A_{\frac{1}{2}-}$.

***1.58** If A is any language, let $A_{\frac{1}{3}-\frac{1}{3}}$ be the set of all strings in A with their middle thirds removed so that

$$A_{\frac{1}{3}-\frac{1}{3}} = \{xz \mid \text{ for some } y,\ |x| = |y| = |z| \text{ and } xyz \in A\}.$$

Show that if A is regular, then $A_{\frac{1}{3}-\frac{1}{3}}$ is not necessarily regular.

1.59** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let h be a state of M called its "home". A ***synchronizing sequence for M and h is a string $s \in \Sigma^*$ where $\delta(q, s) = h$ for every $q \in Q$. (Here we have extended δ to strings, so that $\delta(q, s)$ equals the state where M ends up when M starts at state q and reads input s.) Say that M is ***synchronizable*** if it has a synchronizing sequence for some state h. Prove that if M is a k-state synchronizable DFA, then it has a synchronizing sequence of length at most k^3. Can you improve upon this bound?

1.60 Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. For each $k \geq 1$, let C_k be the language consisting of all strings that contain an `a` exactly k places from the right-hand end. Thus $C_k = \Sigma^*\mathtt{a}\Sigma^{k-1}$. Describe an NFA with $k + 1$ states that recognizes C_k in terms of both a state diagram and a formal description.

1.61 Consider the languages C_k defined in Problem 1.60. Prove that for each k, no DFA can recognize C_k with fewer than 2^k states.

1.62 Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. For each $k \geq 1$, let D_k be the language consisting of all strings that have at least one `a` among the last k symbols. Thus $D_k = \Sigma^*\mathtt{a}(\Sigma \cup \varepsilon)^{k-1}$. Describe a DFA with at most $k+1$ states that recognizes D_k in terms of both a state diagram and a formal description.

***1.63**

a. Let A be an infinite regular language. Prove that A can be split into two infinite disjoint regular subsets.

b. Let B and D be two languages. Write $B \Subset D$ if $B \subseteq D$ and D contains infinitely many strings that are not in B. Show that if B and D are two regular languages where $B \Subset D$, then we can find a regular language C where $B \Subset C \Subset D$.

1.64 Let N be an NFA with k states that recognizes some language A.

a. Show that if A is nonempty, A contains some string of length at most k.

b. Show, by giving an example, that part (a) is not necessarily true if you replace both A's by $\overline{A}$.

c. Show that if $\overline{A}$ is nonempty, $\overline{A}$ contains some string of length at most 2^k.

d. Show that the bound given in part (c) is nearly tight; that is, for each k, demonstrate an NFA recognizing a language A_k where $\overline{A_k}$ is nonempty and where $\overline{A_k}$'s shortest member strings are of length exponential in k. Come as close to the bound in (c) as you can.

*1.65 Prove that for each $n > 0$, a language B_n exists where

a. B_n is recognizable by an NFA that has n states, and

b. if $B_n = A_1 \cup \cdots \cup A_k$, for regular languages A_i, then at least one of the A_i requires a DFA with exponentially many states.

1.66 A ***homomorphism*** is a function $f: \Sigma \longrightarrow \Gamma^*$ from one alphabet to strings over another alphabet. We can extend f to operate on strings by defining $f(w) = f(w_1)f(w_2)\cdots f(w_n)$, where $w = w_1 w_2 \cdots w_n$ and each $w_i \in \Sigma$. We further extend f to operate on languages by defining $f(A) = \{f(w) | \ w \in A\}$, for any language A.

a. Show, by giving a formal construction, that the class of regular languages is closed under homomorphism. In other words, given a DFA M that recognizes B and a homomorphism f, construct a finite automaton M' that recognizes $f(B)$. Consider the machine M' that you constructed. Is it a DFA in every case?

b. Show, by giving an example, that the class of non-regular languages is not closed under homomorphism.

*1.67 Let the ***rotational closure*** of language A be $RC(A) = \{yx | \ xy \in A\}$.

a. Show that for any language A, we have $RC(A) = RC(RC(A))$.

b. Show that the class of regular languages is closed under rotational closure.

*1.68 In the traditional method for cutting a deck of playing cards, the deck is arbitrarily split two parts, which are exchanged before reassembling the deck. In a more complex cut, called Scarne's cut, the deck is broken into three parts and the middle part in placed first in the reassembly. We'll take Scarne's cut as the inspiration for an operation on languages. For a language A, let $CUT(A) = \{yxz | \ xyz \in A\}$.

a. Exhibit a language B for which $CUT(B) \neq CUT(CUT(B))$.

b. Show that the class of regular languages is closed under CUT.

1.69 Let $\Sigma = \{0,1\}$. Let $WW_k = \{ww | \ w \in \Sigma^*$ and w is of length $k\}$.

a. Show that for each k, no DFA can recognize WW_k with fewer than 2^k states.

b. Describe a much smaller NFA for $\overline{WW_k}$, the complement of WW_k.

1.70 We define the ***avoids*** operation for languages A and B to be

$$A \textit{ avoids } B = \{w | \ w \in A \text{ and } w \text{ doesn't contain any string in } B \text{ as a substring}\}.$$

Prove that the class of regular languages is closed under the *avoids* operation.

1.71 Let $\Sigma = \{0,1\}$.

a. Let $A = \{0^k u 0^k | \ k \geq 1$ and $u \in \Sigma^*\}$. Show that A is regular.

b. Let $B = \{0^k 1 u 0^k | \ k \geq 1$ and $u \in \Sigma^*\}$. Show that B is not regular.

1.72 Let M_1 and M_2 be DFAs that have k_1 and k_2 states, respectively, and then let $U = L(M_1) \cup L(M_2)$.

a. Show that if $U \neq \emptyset$, then U contains some string s, where $|s| < \max(k_1, k_2)$.

b. Show that if $U \neq \Sigma^*$, then U excludes some string s, where $|s| < k_1 k_2$.

1.73 Let $\Sigma = \{0,1,\#\}$. Let $C = \{x\#x^{\mathcal{R}}\#x | \ x \in \{0,1\}^*\}$. Show that $\overline{C}$ is a CFL.

SELECTED SOLUTIONS

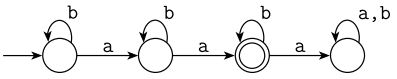
1.1 For M_1: **(a)** q_1; **(b)** $\{q_2\}$; **(c)** q_1, q_2, q_3, q_1, q_1; **(d)** No; **(e)** No
For M_2: **(a)** q_1; **(b)** $\{q_1, q_4\}$; **(c)** q_1, q_1, q_1, q_2, q_4; **(d)** Yes; **(e)** Yes

1.2 $M_1 = (\{q_1, q_2, q_3\}, \{\mathtt{a}, \mathtt{b}\}, \delta_1, q_1, \{q_2\})$.
$M_2 = (\{q_1, q_2, q_3, q_4\}, \{\mathtt{a}, \mathtt{b}\}, \delta_2, q_1, \{q_1, q_4\})$.
The transition functions are

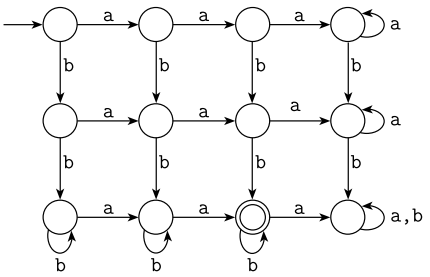
δ_1	a	b
q_1	q_2	q_1
q_2	q_3	q_3
q_3	q_2	q_1

δ_2	a	b
q_1	q_1	q_2
q_2	q_3	q_4
q_3	q_2	q_1
q_4	q_3	q_4.

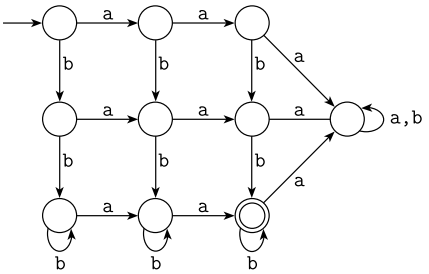
1.4 **(b)** The following are DFAs for the two languages $\{w|\ w$ has exactly two a's$\}$ and $\{w|\ w$ has at least two b's$\}$.



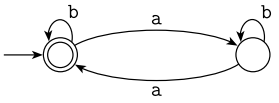
Combining them using the intersection construction gives the following DFA.

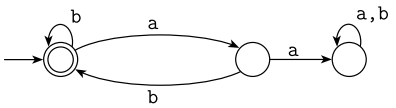


Though the problem doesn't request you to simplify the DFA, certain states can be combined to give the following DFA.

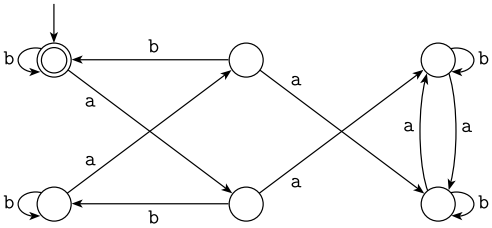


(d) These are DFAs for the two languages $\{w|\ w$ has an even number of a's$\}$ and $\{w|$ each a in w is followed by at least one b$\}$.

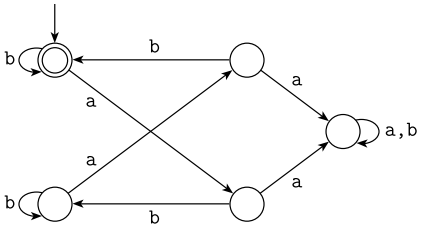




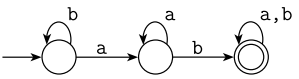
Combining them using the intersection construction gives the following DFA.

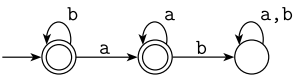


Though the problem doesn't request you to simplify the DFA, certain states can be combined to give the following DFA.

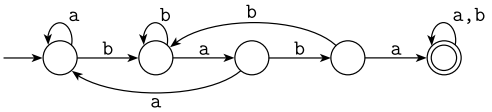


1.5 **(a)** The left-hand DFA recognizes $\{w|\ w$ contains `ab`$\}$. The right-hand DFA recognizes its complement, $\{w|\ w$ doesn't contain `ab`$\}$.

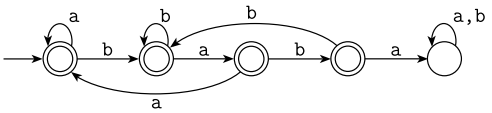




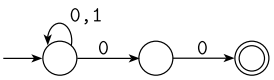
(b) This DFA recognizes $\{w|\ w$ contains `baba`$\}$.



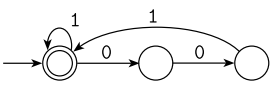
This DFA recognizes $\{w|\ w$ does not contain `baba`$\}$.



1.7 **(a)**



(f)



1.11 Let $N = (Q, \Sigma, \delta, q_0, F)$ be any NFA. Construct an NFA N' with a single accept state that recognizes the same language as N. Informally, N' is exactly like N except it has ε-transitions from the states corresponding to the accept states of N, to a new accept state, q_{accept}. State q_{accept} has no emerging transitions. More formally, $N' = (Q \cup \{q_{\text{accept}}\}, \Sigma, \delta', q_0, \{q_{\text{accept}}\})$, where for each $q \in Q$ and $a \in \Sigma_\varepsilon$

$$\delta'(q,a) = \begin{cases} \delta(q,a) & \text{if } a \neq \varepsilon \text{ or } q \notin F \\ \delta(q,a) \cup \{q_{\text{accept}}\} & \text{if } a = \varepsilon \text{ and } q \in F \end{cases}$$

and $\delta'(q_{\text{accept}}, a) = \emptyset$ for each $a \in \Sigma_\varepsilon$.

1.23 We prove both directions of the "iff."
($\rightarrow$) Assume that $B = B^+$ and show that $BB \subseteq B$.
For every language $BB \subseteq B^+$ holds, so if $B = B^+$, then $BB \subseteq B$.
($\leftarrow$) Assume that $BB \subseteq B$ and show that $B = B^+$.
For every language $B \subseteq B^+$, so we need to show only $B^+ \subseteq B$. If $w \in B^+$, then $w = x_1 x_2 \cdots x_k$ where each $x_i \in B$ and $k \geq 1$. Because $x_1, x_2 \in B$ and $BB \subseteq B$, we have $x_1 x_2 \in B$. Similarly, because $x_1 x_2$ is in B and x_3 is in B, we have $x_1 x_2 x_3 \in B$. Continuing in this way, $x_1 \cdots x_k \in B$. Hence $w \in B$, and so we may conclude that $B^+ \subseteq B$.

The latter argument may be written formally as the following proof by induction.
Assume that $BB \subseteq B$.
Claim: For each $k \geq 1$, if $x_1, \ldots, x_k \in B$, then $x_1 \cdots x_k \in B$.
Basis: Prove for $k = 1$. This statement is obviously true.
Induction step: For each $k \geq 1$, assume that the claim is true for k and prove it to be true for $k + 1$.
If $x_1, \ldots, x_k, x_{k+1} \in B$, then by the induction assumption, $x_1 \cdots x_k \in B$. Therefore, $x_1 \cdots x_k x_{k+1} \in BB$, but $BB \subseteq B$, so $x_1 \cdots x_{k+1} \in B$. That proves the induction step and the claim. The claim implies that if $BB \subseteq B$, then $B^+ \subseteq B$.

1.29 **(a)** Assume that $A_1 = \{0^n 1^n 2^n \mid n \geq 0\}$ is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string $0^p 1^p 2^p$. Because s is a member of A_1 and s is longer than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in A_1. Consider two possibilities:

1. The string y consists only of 0s, only of 1s, or only of 2s. In these cases, the string $xyyz$ will not have equal numbers of 0s, 1s, and 2s. Hence $xyyz$ is not a member of A_1, a contradiction.
2. The string y consists of more than one kind of symbol. In this case, $xyyz$ will have the 0s, 1s, or 2s out of order. Hence $xyyz$ is not a member of A_1, a contradiction.

Either way we arrive at a contradiction. Therefore, A_1 is not regular.

(c) Assume that $A_3 = \{a^{2^n} \mid n \geq 0\}$ is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string a^{2^p}. Because s is a member of A_3 and s is longer than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, satisfying the three conditions of the pumping lemma.

The third condition tells us that $|xy| \leq p$. Furthermore, $p < 2^p$ and so $|y| < 2^p$. Therefore, $|xyyz| = |xyz| + |y| < 2^p + 2^p = 2^{p+1}$. The second condition requires $|y| > 0$ so $2^p < |xyyz| < 2^{p+1}$. The length of $xyyz$ cannot be a power of 2. Hence $xyyz$ is not a member of A_3, a contradiction. Therefore, A_3 is not regular.

1.40 **(a)** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing A, where A is some regular language. Construct $M' = (Q', \Sigma, \delta', q_0', F')$ recognizing $NOPREFIX(A)$ as follows:

1. $Q' = Q$.
2. For $r \in Q'$ and $a \in \Sigma$, define $\delta'(r, a) = \begin{cases} \{\delta(r, a)\} & \text{if } r \notin F \\ \emptyset & \text{if } r \in F. \end{cases}$
3. $q_0' = q_0$.
4. $F' = F$.

1.44 Let $M_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$ and $M_C = (Q_C, \Sigma, \delta_C, q_C, F_C)$ be DFAs recognizing B and C, respectively. Construct NFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes $B \stackrel{1}{\leftarrow} C$ as follows. To decide whether its input w is in $B \stackrel{1}{\leftarrow} C$, the machine M checks that $w \in B$, and in parallel nondeterministically guesses a string y that contains the same number of 1s as contained in w and checks that $y \in C$.

1. $Q = Q_B \times Q_C$.
2. For $(q, r) \in Q$ and $a \in \Sigma_\varepsilon$, define

$$\delta((q,r),a) = \begin{cases} \{(\delta_B(q,0), r)\} & \text{if } a = 0 \\ \{(\delta_B(q,1), \delta_C(r,1))\} & \text{if } a = 1 \\ \{(q, \delta_C(r,0))\} & \text{if } a = \varepsilon. \end{cases}$$

3. $q_0 = (q_B, q_C)$.
4. $F = F_B \times F_C$.

1.46 **(b)** Let $B = \{0^m 1^n \mid m \neq n\}$. Observe that $\overline{B} \cap 0^*1^* = \{0^k 1^k \mid k \geq 0\}$. If B were regular, then $\overline{B}$ would be regular and so would $\overline{B} \cap 0^*1^*$. But we already know that $\{0^k 1^k \mid k \geq 0\}$ isn't regular, so B cannot be regular.

Alternatively, we can prove B to be nonregular by using the pumping lemma directly, though doing so is trickier. Assume that $B = \{0^m 1^n \mid m \neq n\}$ is regular. Let p be the pumping length given by the pumping lemma. Observe that $p!$ is divisible by all integers from 1 to p, where $p! = p(p-1)(p-2)\cdots 1$. The string $s = 0^p 1^{p+p!} \in B$, and $|s| \geq p$. Thus the pumping lemma implies that s can be divided as xyz with $x = 0^a$, $y = 0^b$, and $z = 0^c 1^{p+p!}$, where $b \geq 1$ and $a+b+c = p$. Let s' be the string $xy^{i+1}z$, where $i = p!/b$. Then $y^i = 0^{p!}$ so $y^{i+1} = 0^{b+p!}$, and so $s' = 0^{a+b+c+p!}1^{p+p!}$. That gives $s' = 0^{p+p!}1^{p+p!} \notin B$, a contradiction.

1.50 Assume to the contrary that some FST T outputs $w^{\mathcal{R}}$ on input w. Consider the input strings 00 and 01. On input 00, T must output 00, and on input 01, T must output 10. In both cases, the first input bit is a 0 but the first output bits differ. Operating in this way is impossible for an FST because it produces its first output bit before it reads its second input. Hence no such FST can exist.

1.52 **(a)** We prove this assertion by contradiction. Let M be a k-state DFA that recognizes L. Suppose for a contradiction that L has index greater than k. That means some set X with more than k elements is pairwise distinguishable by L. Because M has k states, the pigeonhole principle implies that X contains two distinct strings x and y, where $\delta(q_0, x) = \delta(q_0, y)$. Here $\delta(q_0, x)$ is the state that M is in after starting in the start state q_0 and reading input string x. Then, for any string $z \in \Sigma^*$, $\delta(q_0, xz) = \delta(q_0, yz)$. Therefore, either both xz and yz are in L or neither are in L. But then x and y aren't distinguishable by L, contradicting our assumption that X is pairwise distinguishable by L.

(b) Let $X = \{s_1, \ldots, s_k\}$ be pairwise distinguishable by L. We construct DFA $M = (Q, \Sigma, \delta, q_0, F)$ with k states recognizing L. Let $Q = \{q_1, \ldots, q_k\}$, and define $\delta(q_i, a)$ to be q_j, where $s_j \equiv_L s_i a$ (the relation $\equiv_L$ is defined in Problem 1.51). Note that $s_j \equiv_L s_i a$ for some $s_j \in X$; otherwise, $X \cup s_i a$ would have $k+1$ elements and would be pairwise distinguishable by L, which would contradict the assumption that L has index k. Let $F = \{q_i \mid s_i \in L\}$. Let the start state q_0 be the q_i such that $s_i \equiv_L \varepsilon$. M is constructed so that for any state q_i, $\{s \mid \delta(q_0, s) = q_i\} = \{s \mid s \equiv_L s_i\}$. Hence M recognizes L.

(c) Suppose that L is regular and let k be the number of states in a DFA recognizing L. Then from part (a), L has index at most k. Conversely, if L has index k, then by part (b) it is recognized by a DFA with k states and thus is regular. To show that the index of L is the size of the smallest DFA accepting it, suppose that L's index is *exactly* k. Then, by part (b), there is a k-state DFA accepting L. That is the smallest such DFA because if it were any smaller, then we could show by part (a) that the index of L is less than k.

1.55 **(a)** The minimum pumping length is 4. The string 000 is in the language but cannot be pumped, so 3 is not a pumping length for this language. If s has length 4 or more, it contains 1s. By dividing s into xyz, where x is 000 and y is the first 1 and z is everything afterward, we satisfy the pumping lemma's three conditions.

(b) The minimum pumping length is 1. The pumping length cannot be 0 because the string ε is in the language and it cannot be pumped. Every nonempty string in the language can be divided into xyz, where x, y, and z are ε, the first character, and the remainder, respectively. This division satisfies the three conditions.

(d) The minimum pumping length is 3. The pumping length cannot be 2 because the string 11 is in the language and it cannot be pumped. Let s be a string in the language of length at least 3. If s is generated by $0^*1^+0^+1^*$ and s begins either 0 or 11, write $s = xyz$ where $x = \varepsilon$, y is the first symbol, and z is the remainder of s. If s is generated by $0^*1^+0^+1^*$ and s begins 10, write $s = xyz$ where $x = 10$, y is the next symbol, and z is the remainder of s. Breaking s up in this way shows that it can be pumped. If s is generated by 10^*1, we can write it as xyz where $x = 1$, $y = 0$, and z is the remainder of s. This division gives a way to pump s.

2

CONTEXT-FREE LANGUAGES

In Chapter 1 we introduced two different, though equivalent, methods of describing languages: *finite automata* and *regular expressions*. We showed that many languages can be described in this way but that some simple languages, such as $\{0^n1^n | \, n \geq 0\}$, cannot.

In this chapter we present ***context-free grammars***, a more powerful method of describing languages. Such grammars can describe certain features that have a recursive structure, which makes them useful in a variety of applications.

Context-free grammars were first used in the study of human languages. One way of understanding the relationship of terms such as *noun*, *verb*, and *preposition* and their respective phrases leads to a natural recursion because noun phrases may appear inside verb phrases and vice versa. Context-free grammars help us organize and understand these relationships.

An important application of context-free grammars occurs in the specification and compilation of programming languages. A grammar for a programming language often appears as a reference for people trying to learn the language syntax. Designers of compilers and interpreters for programming languages often start by obtaining a grammar for the language. Most compilers and interpreters contain a component called a ***parser*** that extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution. A number of methodologies facilitate the construction of a parser once a context-free grammar is available. Some tools even automatically generate the parser from the grammar.

The collection of languages associated with context-free grammars are called the ***context-free languages***. They include all the regular languages and many additional languages. In this chapter, we give a formal definition of context-free grammars and study the properties of context-free languages. We also introduce ***pushdown automata***, a class of machines recognizing the context-free languages. Pushdown automata are useful because they allow us to gain additional insight into the power of context-free grammars.

2.1 CONTEXT-FREE GRAMMARS

The following is an example of a context-free grammar, which we call G_1.

$$\begin{aligned} A &\to \mathtt{0}A\mathtt{1} \\ A &\to B \\ B &\to \mathtt{\#} \end{aligned}$$

A grammar consists of a collection of ***substitution rules***, also called ***productions***. Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a ***variable***. The string consists of variables and other symbols called ***terminals***. The variable symbols often are represented by capital letters. The terminals are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols. One variable is designated as the ***start variable***. It usually occurs on the left-hand side of the topmost rule. For example, grammar G_1 contains three rules. G_1's variables are A and B, where A is the start variable. Its terminals are `0`, `1`, and `#`.

You use a grammar to describe a language by generating each string of that language in the following manner.

1. Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise.
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.
3. Repeat step 2 until no variables remain.

For example, grammar G_1 generates the string `000#111`. The sequence of substitutions to obtain a string is called a ***derivation***. A derivation of string `000#111` in grammar G_1 is

$$A \Rightarrow \mathtt{0}A\mathtt{1} \Rightarrow \mathtt{00}A\mathtt{11} \Rightarrow \mathtt{000}A\mathtt{111} \Rightarrow \mathtt{000}B\mathtt{111} \Rightarrow \mathtt{000\#111}.$$

You may also represent the same information pictorially with a ***parse tree***. An example of a parse tree is shown in Figure 2.1.

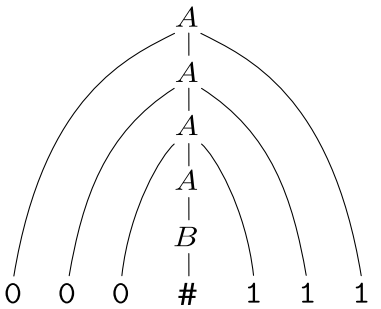


FIGURE 2.1
Parse tree for 000#111 in grammar G_1

All strings generated in this way constitute the ***language of the grammar***. We write $L(G_1)$ for the language of grammar G_1. Some experimentation with the grammar G_1 shows us that $L(G_1)$ is $\{\mathtt{0}^n\mathtt{\#1}^n \mid n \geq 0\}$. Any language that can be generated by some context-free grammar is called a ***context-free language*** (CFL). For convenience when presenting a context-free grammar, we abbreviate several rules with the same left-hand variable, such as $A \to \mathtt{0}A\mathtt{1}$ and $A \to B$, into a single line $A \to \mathtt{0}A\mathtt{1} \mid B$, using the symbol " | " as an "or".

The following is a second example of a context-free grammar, called G_2, which describes a fragment of the English language.

$$
\begin{aligned}
\langle \text{SENTENCE} \rangle &\to \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
\langle \text{NOUN-PHRASE} \rangle &\to \langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{VERB-PHRASE} \rangle &\to \langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{PREP-PHRASE} \rangle &\to \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \\
\langle \text{CMPLX-NOUN} \rangle &\to \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\
\langle \text{CMPLX-VERB} \rangle &\to \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \\
\langle \text{ARTICLE} \rangle &\to \mathtt{a} \mid \mathtt{the} \\
\langle \text{NOUN} \rangle &\to \mathtt{boy} \mid \mathtt{girl} \mid \mathtt{flower} \\
\langle \text{VERB} \rangle &\to \mathtt{touches} \mid \mathtt{likes} \mid \mathtt{sees} \\
\langle \text{PREP} \rangle &\to \mathtt{with}
\end{aligned}
$$

Grammar G_2 has 10 variables (the capitalized grammatical terms written inside brackets); 27 terminals (the standard English alphabet plus a space character); and 18 rules. Strings in $L(G_2)$ include:

```
a boy sees
the boy sees a flower
a girl with a flower likes the boy
```

Each of these strings has a derivation in grammar G_2. The following is a derivation of the first string on this list.

$$\begin{aligned}\langle \text{SENTENCE} \rangle &\Rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\ &\Rightarrow \langle \text{CMPLX-NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\ &\Rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\ &\Rightarrow \texttt{a}\ \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\ &\Rightarrow \texttt{a boy}\ \langle \text{VERB-PHRASE} \rangle \\ &\Rightarrow \texttt{a boy}\ \langle \text{CMPLX-VERB} \rangle \\ &\Rightarrow \texttt{a boy}\ \langle \text{VERB} \rangle \\ &\Rightarrow \texttt{a boy sees}\end{aligned}$$

FORMAL DEFINITION OF A CONTEXT-FREE GRAMMAR

Let's formalize our notion of a context-free grammar (CFG).

DEFINITION 2.2

A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

If u, v, and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv ***yields*** uwv, written $uAv \Rightarrow uwv$. Say that u ***derives*** v, written $u \stackrel{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

The ***language of the grammar*** is $\{w \in \Sigma^* | \ S \stackrel{*}{\Rightarrow} w\}$.

In grammar G_1, $V = \{A, B\}$, $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{\#}\}$, $S = A$, and R is the collection of the three rules appearing on page 102. In grammar G_2,

$$\begin{aligned}V = \{&\langle \text{SENTENCE} \rangle, \langle \text{NOUN-PHRASE} \rangle, \langle \text{VERB-PHRASE} \rangle, \\ &\langle \text{PREP-PHRASE} \rangle, \langle \text{CMPLX-NOUN} \rangle, \langle \text{CMPLX-VERB} \rangle, \\ &\langle \text{ARTICLE} \rangle, \langle \text{NOUN} \rangle, \langle \text{VERB} \rangle, \langle \text{PREP} \rangle\},\end{aligned}$$

and $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z}, \text{“ ”}\}$. The symbol “ ” is the blank symbol, placed invisibly after each word (a, boy, etc.), so the words won't run together.

Often we specify a grammar by writing down only its rules. We can identify the variables as the symbols that appear on the left-hand side of the rules and the terminals as the remaining symbols. By convention, the start variable is the variable on the left-hand side of the first rule.

EXAMPLES OF CONTEXT-FREE GRAMMARS

EXAMPLE 2.3

Consider grammar $G_3 = (\{S\}, \{\mathtt{a}, \mathtt{b}\}, R, S)$. The set of rules, R, is

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid SS \mid \varepsilon.$$

This grammar generates strings such as `abab`, `aaabbb`, and `aababb`. You can see more easily what this language is if you think of `a` as a left parenthesis "`(`" and `b` as a right parenthesis "`)`". Viewed in this way, $L(G_3)$ is the language of all strings of properly nested parentheses. Observe that the right-hand side of a rule may be the empty string ε.

EXAMPLE 2.4

Consider grammar $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$.

V is $\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$ and Σ is $\{\mathtt{a}, \mathtt{+}, \mathtt{x}, \mathtt{(}, \mathtt{)}\}$. The rules are

$$\begin{aligned} \langle \text{EXPR} \rangle &\rightarrow \langle \text{EXPR} \rangle \mathtt{+} \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\ \langle \text{TERM} \rangle &\rightarrow \langle \text{TERM} \rangle \mathtt{x} \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\ \langle \text{FACTOR} \rangle &\rightarrow \mathtt{(} \langle \text{EXPR} \rangle \mathtt{)} \mid \mathtt{a} \end{aligned}$$

The two strings `a+axa` and `(a+a)xa` can be generated with grammar G_4. The parse trees are shown in the following figure.

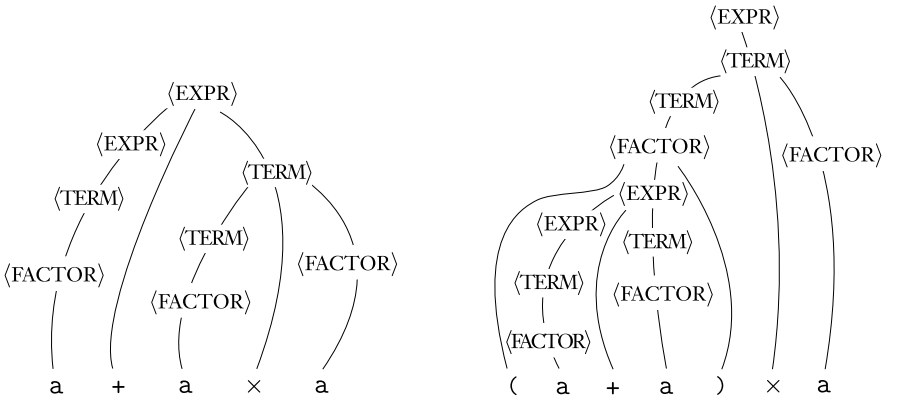


FIGURE 2.5
Parse trees for the strings `a+axa` and `(a+a)xa`

A compiler translates code written in a programming language into another form, usually one more suitable for execution. To do so, the compiler extracts

the meaning of the code to be compiled in a process called ***parsing***. One representation of this meaning is the parse tree for the code, in the context-free grammar for the programming language. We discuss an algorithm that parses context-free languages later in Theorem 7.16 and in Problem 7.45.

Grammar G_4 describes a fragment of a programming language concerned with arithmetic expressions. Observe how the parse trees in Figure 2.5 "group" the operations. The tree for `a+axa` groups the `x` operator and its operands (the second two `a`'s) as one operand of the `+` operator. In the tree for `(a+a)xa`, the grouping is reversed. These groupings fit the standard precedence of multiplication before addition and the use of parentheses to override the standard precedence. Grammar G_4 is designed to capture these precedence relations. ■

DESIGNING CONTEXT-FREE GRAMMARS

As with the design of finite automata, discussed in Section 1.1 (page 41), the design of context-free grammars requires creativity. Indeed, context-free grammars are even trickier to construct than finite automata because we are more accustomed to programming a machine for specific tasks than we are to describing languages with grammars. The following techniques are helpful, singly or in combination, when you're faced with the problem of constructing a CFG.

First, many CFLs are the union of simpler CFLs. If you must construct a CFG for a CFL that you can break into simpler pieces, do so and then construct individual grammars for each piece. These individual grammars can be easily merged into a grammar for the original language by combining their rules and then adding the new rule $S \to S_1 \mid S_2 \mid \cdots \mid S_k$, where the variables S_i are the start variables for the individual grammars. Solving several simpler problems is often easier than solving one complicated problem.

For example, to get a grammar for the language $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\} \cup \{\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$, first construct the grammar

$$S_1 \to \mathtt{0}S_1\mathtt{1} \mid \varepsilon$$

for the language $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ and the grammar

$$S_2 \to \mathtt{1}S_2\mathtt{0} \mid \varepsilon$$

for the language $\{\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$ and then add the rule $S \to S_1 \mid S_2$ to give the grammar

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to \mathtt{0}S_1\mathtt{1} \mid \varepsilon \\ S_2 &\to \mathtt{1}S_2\mathtt{0} \mid \varepsilon. \end{aligned}$$

Second, constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language. You can convert any DFA into an equivalent CFG as follows. Make a variable R_i for each state q_i of the DFA. Add the rule $R_i \to aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA. Add the rule $R_i \to \varepsilon$ if q_i is an accept state of the DFA. Make R_0 the start variable of the grammar, where q_0 is the start state of the machine. Verify on your own that the resulting CFG generates the same language that the DFA recognizes.

Third, certain context-free languages contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring. This situation occurs in the language $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s. You can construct a CFG to handle this situation by using a rule of the form $R \to uRv$, which generates strings wherein the portion containing the u's corresponds to the portion containing the v's.

Finally, in more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures. That situation occurs in the grammar that generates arithmetic expressions in Example 2.4. Any time the symbol a appears, an entire parenthesized expression might appear recursively instead. To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.

AMBIGUITY

Sometimes a grammar can generate the same string in several different ways. Such a string will have several different parse trees and thus several different meanings. This result may be undesirable for certain applications, such as programming languages, where a program should have a unique interpretation.

If a grammar generates the same string in several different ways, we say that the string is derived *ambiguously* in that grammar. If a grammar generates some string ambiguously, we say that the grammar is *ambiguous*.

For example, consider grammar G_5:

$$\langle \text{EXPR} \rangle \to \langle \text{EXPR} \rangle \texttt{+} \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \texttt{x} \langle \text{EXPR} \rangle \mid \texttt{(} \langle \text{EXPR} \rangle \texttt{)} \mid \texttt{a}$$

This grammar generates the string a+axa ambiguously. The following figure shows the two different parse trees.

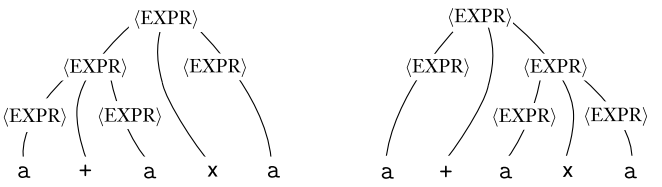


FIGURE 2.6
The two parse trees for the string a+axa in grammar G_5

This grammar doesn't capture the usual precedence relations and so may group the $+$ before the $\times$ or vice versa. In contrast, grammar G_4 generates exactly the same language, but every generated string has a unique parse tree. Hence G_4 is unambiguous, whereas G_5 is ambiguous.

Grammar G_2 (page 103) is another example of an ambiguous grammar. The sentence `the girl touches the boy with the flower` has two different derivations. In Exercise 2.8 you are asked to give the two parse trees and observe their correspondence with the two different ways to read that sentence.

Now we formalize the notion of ambiguity. When we say that a grammar generates a string ambiguously, we mean that the string has two different parse trees, not two different derivations. Two derivations may differ merely in the order in which they replace variables yet not in their overall structure. To concentrate on structure, we define a type of derivation that replaces variables in a fixed order. A derivation of a string w in a grammar G is a ***leftmost derivation*** if at every step the leftmost remaining variable is the one replaced. The derivation preceding Definition 2.2 (page 104) is a leftmost derivation.

DEFINITION 2.7

A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language. Some context-free languages, however, can be generated only by ambiguous grammars. Such languages are called ***inherently ambiguous***. Problem 2.29 asks you to prove that the language $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = j \text{ or } j = k\}$ is inherently ambiguous.

CHOMSKY NORMAL FORM

When working with context-free grammars, it is often convenient to have them in simplified form. One of the simplest and most useful forms is called the

Chomsky normal form. Chomsky normal form is useful in giving algorithms for working with context-free grammars, as we do in Chapters 4 and 7.

DEFINITION 2.8

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$$A \to BC$$
$$A \to a$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition, we permit the rule $S \to \varepsilon$, where S is the start variable.

THEOREM 2.9

Any context-free language is generated by a context-free grammar in Chomsky normal form.

PROOF IDEA We can convert any grammar G into Chomsky normal form. The conversion has several stages wherein rules that violate the conditions are replaced with equivalent ones that are satisfactory. First, we add a new start variable. Then, we eliminate all ***ε-rules*** of the form $A \to \varepsilon$. We also eliminate all ***unit rules*** of the form $A \to B$. In both cases we patch up the grammar to be sure that it still generates the same language. Finally, we convert the remaining rules into the proper form.

PROOF First, we add a new start variable S_0 and the rule $S_0 \to S$, where S was the original start variable. This change guarantees that the start variable doesn't occur on the right-hand side of a rule.

Second, we take care of all ε-rules. We remove an ε-rule $A \to \varepsilon$, where A is not the start variable. Then for each occurrence of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted. In other words, if $R \to uAv$ is a rule in which u and v are strings of variables and terminals, we add rule $R \to uv$. We do so for each *occurrence* of an A, so the rule $R \to uAvAw$ causes us to add $R \to uvAw$, $R \to uAvw$, and $R \to uvw$. If we have the rule $R \to A$, we add $R \to \varepsilon$ unless we had previously removed the rule $R \to \varepsilon$. We repeat these steps until we eliminate all ε-rules not involving the start variable.

Third, we handle all unit rules. We remove a unit rule $A \to B$. Then, whenever a rule $B \to u$ appears, we add the rule $A \to u$ unless this was a unit rule previously removed. As before, u is a string of variables and terminals. We repeat these steps until we eliminate all unit rules.

Finally, we convert all remaining rules into the proper form. We replace each rule $A \to u_1 u_2 \cdots u_k$, where $k \geq 3$ and each u_i is a variable or terminal symbol,

with the rules $A \to u_1A_1$, $A_1 \to u_2A_2$, $A_2 \to u_3A_3$, ..., and $A_{k-2} \to u_{k-1}u_k$. The A_i's are new variables. We replace any terminal u_i in the preceding rule(s) with the new variable U_i and add the rule $U_i \to u_i$.

EXAMPLE 2.10

Let G_6 be the following CFG and convert it to Chomsky normal form by using the conversion procedure just given. The series of grammars presented illustrates the steps in the conversion. Rules shown in bold have just been added. Rules shown in gray have just been removed.

1. The original CFG G_6 is shown on the left. The result of applying the first step to make a new start variable appears on the right.

$$S \to ASA \mid \mathtt{a}B$$
$$A \to B \mid S$$
$$B \to \mathtt{b} \mid \varepsilon$$

$$\boldsymbol{S_0 \to S}$$
$$S \to ASA \mid \mathtt{a}B$$
$$A \to B \mid S$$
$$B \to \mathtt{b} \mid \varepsilon$$

2. Remove ε-rules $B \to \varepsilon$, shown on the left, and $A \to \varepsilon$, shown on the right.

$$S_0 \to S$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a}$$
$$A \to B \mid S \mid \boldsymbol{\varepsilon}$$
$$B \to \mathtt{b} \mid \boldsymbol{\varepsilon}$$

$$S_0 \to S$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S}$$
$$A \to B \mid S \mid \boldsymbol{\varepsilon}$$
$$B \to \mathtt{b}$$

3a. Remove unit rules $S \to S$, shown on the left, and $S_0 \to S$, shown on the right.

$$S_0 \to S$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \mid S$$
$$A \to B \mid S$$
$$B \to \mathtt{b}$$

$$S_0 \to S \mid \boldsymbol{ASA} \mid \boldsymbol{\mathtt{a}B} \mid \boldsymbol{\mathtt{a}} \mid \boldsymbol{SA} \mid \boldsymbol{AS}$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$A \to B \mid S$$
$$B \to \mathtt{b}$$

3b. Remove unit rules $A \to B$ and $A \to S$.

$$S_0 \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$A \to B \mid S \mid \mathtt{b}$$
$$B \to \mathtt{b}$$

$$S_0 \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$A \to S \mid \mathtt{b} \mid \boldsymbol{ASA} \mid \boldsymbol{\mathtt{a}B} \mid \boldsymbol{\mathtt{a}} \mid \boldsymbol{SA} \mid \boldsymbol{AS}$$
$$B \to \mathtt{b}$$

4. Convert the remaining rules into the proper form by adding additional variables and rules. The final grammar in Chomsky normal form is equivalent to G_6. (Actually the procedure given in Theorem 2.9 produces several variables U_i and several rules $U_i \rightarrow \mathtt{a}$. We simplified the resulting grammar by using a single variable U and rule $U \rightarrow \mathtt{a}$.)

$$\begin{aligned} S_0 &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\ S &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\ A &\rightarrow \mathtt{b} \mid AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\ A_1 &\rightarrow SA \\ U &\rightarrow \mathtt{a} \\ B &\rightarrow \mathtt{b} \end{aligned}$$

2.2 PUSHDOWN AUTOMATA

In this section we introduce a new type of computational model called ***pushdown automata***. These automata are like nondeterministic finite automata but have an extra component called a ***stack***. The stack provides additional memory beyond the finite amount available in the control. The stack allows pushdown automata to recognize some nonregular languages.

Pushdown automata are equivalent in power to context-free grammars. This equivalence is useful because it gives us two options for proving that a language is context free. We can give either a context-free grammar generating it or a pushdown automaton recognizing it. Certain languages are more easily described in terms of generators, whereas others are more easily described by recognizers.

The following figure is a schematic representation of a finite automaton. The control represents the states and transition function, the tape contains the input string, and the arrow represents the input head, pointing at the next input symbol to be read.

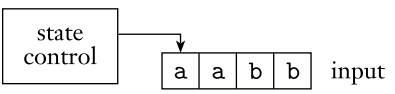


FIGURE 2.11
Schematic of a finite automaton