DISCRETE MATHEMATICS
WITH APPLICATIONS

FOURTH EDITION

SUSANNA S. EPP
DePaul University

BROOKS/COLE
CENGAGE Learning

Australia · Brazil · Japan · Korea · Mexico · Singapore · Spain · United Kingdom · United States

BROOKS/COLE
CENGAGE Learning

Cover Photo: *The stones are discrete objects placed one on top of another like a chain of careful reasoning. A person who decides to build such a tower aspires to the heights and enjoys playing with a challenging problem. Choosing the stones takes both a scientific and an aesthetic sense. Getting them to balance requires patient effort and careful thought. And the tower that results is beautiful. A perfect metaphor for discrete mathematics!*

Discrete Mathematics with Applications, Fourth Edition
Susanna S. Epp

Publisher: Richard Stratton
Senior Sponsoring Editor: Molly Taylor
Associate Editor: Daniel Seibert
Editorial Assistant: Shaylin Walsh
Associate Media Editor: Andrew Coppola
Senior Marketing Manager: Jennifer Pursley Jones
Marketing Communications Manager: Mary Anne Payumo
Marketing Coordinator: Erica O'Connell
Content Project Manager: Alison Eigel Zade
Senior Art Director: Jill Ort
Senior Print Buyer: Diane Gibbons
Right Acquisition Specialists: Timothy Sisler and Don Schlotman
Production Service: Elm Street Publishing Services
Photo Manager: Chris Althof, Bill Smith Group
Cover Designer: Hanh Luu
Cover Image: GettyImages.com (Collection: OJO Images, Photographer: Martin Barraud)
Compositor: Integra Software Services Pvt. Ltd.

Library of Congress Control Number: 2010927831

Student Edition:
ISBN-13: 978-0-495-39132-6
ISBN-10: 0-495-39132-8

Brooks/Cole
20 Channel Center Street
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at: **international.cengage.com/region.**

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your course and learning solutions, visit **www.cengage.com**

Purchase any of our products at your local college store or at our preferred online store **www.cengagebrain.com.**

Printed in Canada
2 3 4 5 6 7 14 13 12 11

CHAPTER 12

REGULAR EXPRESSIONS AND FINITE-STATE AUTOMATA

The theoretical foundations of computer science were derived from several disciplines: logic (the foundations of mathematics), electrical engineering (the design of switching circuits), brain research (models of neurons), and linguistics (the formal specification of languages).

As discussed briefly in Sections 6.4 and 7.4, the 1930s saw the development of mathematical treatments of basic questions concerning what can be proved in mathematics and what can be computed by means of a finite sequence of mechanized operations. Although the first digital computers were not built until the early 1940s, ten years earlier Alan Turing developed a simple abstract model of a machine, now called a Turing machine, by means of which he defined what it would mean for a function to be computable.

Around the same time, somewhat similar models of computation were developed by the American logicians Alonzo Church, Stephen C. Kleene, and Emil Post (who was born in Poland but came to the United States as a child), but Church and others showed these all to be equivalent. As a result, Church formulated a conjecture, now known as the **Church-Turing thesis,** asserting that the Turing machine is universal in the sense that anything that can ever be computed on a machine can be computed with a Turing machine. If this thesis is correct—which is widely believed—then all computers that have been or will ever be constructed are theoretically equivalent in what they can do, although they may differ widely in speed and storage capacity. For instance, quantum computers may have the capability to compute certain quantities enormously faster than classical computers. But Church's thesis implies that the theory of computation is likely to remain fundamentally the same, even though the enabling technology is subject to constant change.

In the early 1940s, Warren S. McCulloch and Walter Pitts, working at the Massachusetts Institute of Technology (M.I.T.), developed a model of how the neurons in the brain might work and how models of neurons could be combined to make "circuits" or "automata" capable of more complicated computations. To a certain extent, they were influenced by the results of Claude Shannon, who also worked at M.I.T. and had in the 1930s developed the foundations of a theory that implemented Boolean functions as switching circuits. In the 1950s, Kleene analyzed the work of McCulloch and Pitts and connected it with versions of the machine models introduced by Turing and others.

Another development of the 1950s was the introduction of high-level computer languages. During the same years, linguist Noam Chomsky's attempts to understand the underlying principles by means of which human beings generate speech led him to develop a theory of formal languages, which he defined using sets of abstract rules, called

grammars, of varying levels of complexity. It soon became apparent that Chomsky's theory was of great utility in the analysis and construction of computer languages. For computer science, the most useful of Chomsky's language classifications are also the two simplest: the *regular languages* and the *context-free languages*.

Regular languages, which are defined by *regular expressions*, are used extensively for matching patterns within text (as in word processing or Internet searches) and for lexical analysis in computer language compilers. They are part of sophisticated text editors and a number of UNIX* utilities, and they are also used in transforming XML† documents.

Through use of the Backus-Naur notation (introduced in Section 10.5), context-free languages are able to describe many of the more complex aspects of modern high-level computer languages, and they form the basis for the main part of compilers, which translate programs written in a high-level language into machine code suitable for execution.

A remarkable fact is that all of the subjects referred to previously are related. Each context-free grammar turns out to be equivalent to a type of automaton called a *push-down automaton*, and each regular expression turns out to be equivalent to a type of automaton called a *finite-state automaton*. In this chapter, we focus on the study of regular languages and finite-state automata, leaving the subject of context-free grammars and their equivalent automata to a later course in compiler construction or automata theory.

Note *Automata* is the plural of *automaton*.

12.1 Formal Languages and Regular Expressions

The mind has finite means but it makes unbounded use of them and in very specific and organized ways. That's the core problem of language that it became possible to face [by the mid-twentieth century]. — Noam Chomsky, circa 1998

Photo by Norman Lenburg, 1979. Courtesy University of Wisconsin-Madison Archives.

Noam Chomsky (born 1928)

An English sentence can be regarded as a string of words, and an English word can be regarded as a string of letters. Not every string of letters is a legitimate word, and not every string of words is a grammatical sentence. We could say that a word is legitimate if it can be found in an unabridged English dictionary and that a sentence is grammatical if it satisfies the rules in a standard English grammar book.

Computer languages are similar to English in that certain strings of characters are legitimate words of the language and certain strings of words can be put together according to certain rules to form syntactically correct programs. A compiler for a computer language analyzes the stream of characters in a program—first to recognize individual word and sentence units (this part of the compiler is called a lexical scanner), then to analyze the syntax, or grammar, of the sentences (this part is called a syntactic analyzer), and finally to translate the sentences into machine code (this part is called a code generator).

In computer science it has proved useful to look at languages from a very abstract point of view as strings of certain fundamental units and allow any finite set of symbols to be used as an alphabet. It is common to denote an alphabet by a capital Greek sigma: Σ. (This just happens to be the same symbol as the one used for summation, but the two concepts have no other connection.)

The definition of a *string of characters of an alphabet* Σ (or a *string over* Σ) is a generalization of the definition of string introduced earlier. A *formal language over an alphabet* is any set of strings of characters of the alphabet. These definitions are given formally on the next page.

*UNIX is an operating system that was developed in 1969 by Kenneth Thompson at Bell Laboratories. It was later rewritten in Dennis Ritchie's C language, which was also developed at Bell Laboratories.

†XML is a standard for defining markup languages used for Internet applications.

Alphabet Σ:	a finite set of characters
String over Σ:	(1) a finite sequence of elements (called **characters**) of Σ or (2) the null string ϵ
Length of a string over Σ:	The number of characters that made up the string, with the null string having length 0.
Formal language over Σ:	a set of strings over the alphabet

Note that the empty set satisfies the criteria for being a formal language. Allowing the empty set to be a formal language turns out to be convenient in certain technical situations.

Example 12.1.1 Examples of Formal Languages

Let the alphabet $\Sigma = \{a, b\}$.

a. Define a language L_1 over Σ to be the set of all strings that begin with the character a and have length at most three characters. Find L_1.

b. A **palindrome** is a string that looks the same if the order of its characters is reversed. For instance, *aba* and *baab* are palindromes. Define a language L_2 over Σ to be the set of all palindromes obtained using the characters of Σ. Write ten elements of L_2.

Solution

a. $L_1 = \{a, aa, ab, aaa, aab, aba, abb\}$

b. L_2 contains the following ten strings (among infinitely many others):

$$\epsilon, a, b, aa, bb, aaa, bab, abba, babaabab, abaabbbbbaaba$$ ■

Stephen C. Kleene (1909–1994)

• Notation

Let Σ be an alphabet. For each nonnegative integer n, let

Σ^n = the set of all strings over Σ that have length n,

Σ^+ = the set of all strings over Σ that have length at least 1, and

Σ^* = the set of all strings over Σ.

Note that Σ^n is essentially the Cartesian product of n copies of Σ. The language Σ^* is called the **Kleene closure of Σ**, in honor of Stephen C. Kleene (pronounced CLAY-knee). Σ^+ is the set of all strings over Σ except for ϵ and is called the **positive closure of Σ**.

Example 12.1.2 The Languages Σ^n, Σ^+, and Σ^*

Let $\Sigma = \{a, b\}$.

a. Find Σ^0, Σ^1, Σ^2, and Σ^3.

b. Let $A = \Sigma^0 \cup \Sigma^1$ and $B = \Sigma^2 \cup \Sigma^3$. Use words to describe A, B, and $A \cup B$.

c. Describe a systematic way of writing the elements of Σ^+. What change needs to be made to obtain the elements of Σ^*?

Solution

a. $\Sigma^0 = \{\epsilon\}$, $\Sigma^1 = \{a, b\}$, $\Sigma^2 = \{aa, ab, ba, bb\}$, and $\Sigma^3 = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$

b. A is the set of all strings over Σ of length at most 1.

B is the set of all strings over Σ of length 2 or 3.

$A \cup B$ is the set of all strings over Σ of length at most 3.

c. Elements of Σ^+ can be written systematically by writing all the strings of length 1, then all the strings of length 2, and so forth.

$$\Sigma^+: a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, \ldots$$

Of course the process of writing the strings in Σ^+ would continue forever, because Σ^+ is an infinite set. The only change that needs to be made to obtain Σ^* is to place the null string at the beginning of the list. ■

Example 12.1.3 Polish Notation: A Language Consisting of Postfix Expressions

An expression such as $a + b$, in which a binary operator such as $+$ sits between the two quantities on which it acts, is said to be written in **infix notation**. Alternative notations are called **prefix notation** (in which the binary operator precedes the quantities on which it acts) and **postfix notation** (in which the binary operator follows the quantities on which it acts). In prefix notation, $a + b$ is written $+ab$. In postfix notation, $a + b$ is written $ab +$.

Prefix and postfix notations were introduced in 1920 by the Polish mathematician Jan Łukasiewicz (pronounced Wu-cash-AY-vich). In his honor—and because some people have difficulty pronouncing his name—they are often referred to as **Polish notation** and **reverse Polish notation,** respectively. A great advantage of these notations is that they eliminate the need for parentheses in writing arithmetic expressions. For instance, in postfix (or reverse Polish) notation, the expression 8 4 + 6 / is evaluated from left to right as follows: Add 8 and 4 to obtain 12, and then divide 12 by 6 to obtain 2. As another example, if the expression $(a + b) \cdot c$ in infix notation is converted to postfix notation, the result is $ab + c \cdot$.

a. If the expression $ab \cdot cd \cdot +$ in postfix notation is converted to infix notation, what is the result?

b. Let $\Sigma = \{4, 1, +, -\}$, and let $L =$ the set of all strings over Σ obtained by writing either a 4 or a 1 first, then either a 4 or a 1, and finally either a + or a −. List all elements of L between braces, and evaluate the resulting expressions.

Solution

a. $a \cdot b + c \cdot d$

b. $L = \{4\,1+,\ 4\,1-,\ 1\,4+,\ 1\,4-,\ 4\,4+,\ 4\,4-,\ 1\,1+,\ 1\,1-\}$

$4\,1+ = 4 + 1 = 5, \quad 4\,1- = 4 - 1 = 3, \quad 1\,4+ = 1 + 4 = 5,$

$1\,4- = 1 - 4 = -3, \quad 4\,4+ = 4 + 4 = 8, \quad 4\,4- = 4 - 4 = 0,$

$1\,1+ = 1 + 1 = 2, \quad 1\,1- = 1 - 1 = 0$ ■

The following definition describes ways in which languages can be combined to form new languages.

• Definition

Let Σ be an alphabet. Given any strings x and y over Σ, the **concatenation of x and y** is the string obtained by writing all the characters of x followed by all the character of y. For any languages L and L' over Σ, three new languages can be defined as follows:

The **concatenation of L and L'**, denoted LL', is

$$LL' = \{xy \mid x \in L \text{ and } y \in L'\}.$$

The **union of L and L'**, denoted $L \cup L'$, is

$$L \cup L' = \{x \mid x \in L \text{ or } x \in L'\}.$$

The **Kleene closure of L**, denoted L^*, is

$$L^* = \{x \mid x \text{ is a concatenation of any finite number of strings in } L\}.$$

Note that ϵ is in L^* because it is regarded as a concatenation of zero strings in L.

Example 12.1.4 New Languages from Old

Let L_1 be the set of all strings consisting of an even number of a's (namely, ε, aa, $aaaa$, $aaaaaa$, ...), and let $L_2 = \{b, bb, bbb\}$. Find L_1L_2, $L_1 \cup L_2$, and $(L_1 \cup L_2)^*$. Note that the null string ϵ is in L_1 because 0 is an even number.

Solution

L_1L_2 = the set of all strings that consist of an even number of a's followed by b or by bb or by bbb.

$L_1 \cup L_2$ = the set that includes the strings b, bb, bbb and any strings consisting of an even number of a's.

$(L_1 \cup L_2)^*$ = the set of all strings of a's and b's in which every occurrence of a is in a block consisting of an even number of a's. ■

The Language Defined by a Regular Expression

One of the most useful ways to define a language is by means of a *regular expression*, a concept first introduced by Kleene. We give a recursive definition for generating the set of all regular expressions over an alphabet.

• Definition

Given an alphabet Σ, the following are **regular expressions over Σ:**

I. BASE: $\emptyset$, ϵ, and each individual symbol in Σ are regular expressions over Σ.

II. RECURSION: If r and s are regular expressions over Σ, then the following are also regular expressions over Σ:

(i) (rs) (ii) $(r \mid s)$ (iii) (r^*)

where rs denotes the concatenation of r and s, r^* denotes the concatenation of r with itself any finite number (including zero) of times, and $r \mid s$ denotes either one of the strings r or s. The regular expression r^* is called the **Kleene closure** of r.

III. RESTRICTION: Nothing is a regular expression over Σ except for objects defined in (I) and (II) above.

As an example, one regular expression over $\Sigma = \{a, b, c\}$ is

$$a \mid (b \mid c)^* \mid (ab)^*.$$

If the alphabet Σ happens to include symbols—such as (|) *—special provisions have to be made to avoid ambiguity. An *escape character*, usually a backslash, is added before the potentially ambiguous symbol. For instance, a left parenthesis would be written as \(and the backslash itself would be written as \\.

To eliminate parentheses, an order of precedence for the operations used to define regular expressions has been introduced. The highest is *, concatenation is next, and | is the lowest. It is also customary to eliminate the outer set of parentheses in a regular expression, because doing so does not produce ambiguity. Thus

$$(a((bc)^*)) = a(bc)^* \quad \text{and} \quad (a \mid (bc)) = a \mid bc.$$

Example 12.1.5 Order of Precedence for the Operations in a Regular Expression

a. Add parentheses to make the order of precedence clear in the following expression: $ab^* \mid b^*a$.

b. Use the convention about order of precedence to eliminate the parentheses in the following expression: $((a \mid ((b^*)c))(a^*))$.

Solution

a. $((a(b^*)) \mid ((b^*)a))$ b. $(a \mid b^*c)a^*$ ■

Given a finite alphabet, every regular expression r over the alphabet defines a formal language $L(r)$. The function L is defined recursively.

• Definition

For any finite alphabet Σ, the function L that associates a language to each regular expression over Σ is defined by (I) and (II) below. For each such regular expression r, $L(r)$ is called the **language defined by** r.

I. BASE: $L(\emptyset) = \emptyset$, $L(\epsilon) = \{\epsilon\}$, $L(a) = \{a\}$ for every a in Σ.

II. RECURSION: If $L(r)$ and $L(r')$ are the languages defined by the regular expressions r and r' over Σ, then

(i) $L(rr') = L(r)L(r')$ (ii) $L(r \mid r') = L(r) \cup L(r')$ (iii) $L(r^*) = (L(r))^*$

Note that any finite language can be defined by a regular expression. For instance, the language {cat, dog, bird} is defined by the regular expression (cat | dog | bird). An important example is the following.

Example 12.1.6 Using Set Notation to Describe the Language Defined by a Regular Expression

Let $\Sigma = \{a, b\}$, and consider the language defined by the regular expression $(a \mid b)^*$. Use set notation to find this language, and describe it in words.

Solution The language defined by $(a \mid b)^*$ is

$$\begin{aligned} L((a \mid b)^*) &= (L(a \mid b))^* \\ &= (L(a) \cup L(b))^* \\ &= (\{a\} \cup \{b\})^* \\ &= \{a, b\}^* && \text{by definition of operations on languages} \\ &= \text{the set of all strings of } a\text{'s and } b\text{'s} \\ &= \Sigma^*. \end{aligned}$$

■

Note that concatenating strings and taking unions of sets are both associative operations. Thus for any regular expressions r, s and t,

$$L((rs)t) = L(r(st)).$$

Moreover,

$$\begin{aligned} L((r \mid s) \mid t) &= (L(r \mid s)) \cup L(t) && \text{by definition of } \mid \\ &= (L(r) \cup L(s)) \cup L(t) && \text{by definition of } \mid \\ &= L(r) \cup (L(s) \cup L(t)) && \text{by the associativity of union for sets} \\ &= L(r) \cup (L(s \mid t)) && \text{by definition of } \mid \\ &= L(r \mid (s \mid t)) && \text{by definition of } \mid. \end{aligned}$$

Because of these relationships, it is customary to drop the parentheses in "associative" situations and write

$$rst = (rs)t = r(st)$$

and

$$r \mid s \mid t = (r \mid s) \mid t = r \mid (s \mid t).$$

As you become accustomed to working with regular expressions, you will find that you do not need to go through a formal derivation in order to determine the language defined by an expression.

Example 12.1.7 The Language Defined by a Regular Expression

Let $\Sigma = \{0, 1\}$. Use words to describe the languages defined by the following regular expressions over Σ.

a. $0^*1^* \mid 1^*0^*$ b. $0(0 \mid 1)^*$

Solution

a. The strings in this language consist either of a string of 0's followed by a string of 1's or of a string of 1's followed by a string of 0's. However, in either case the strings could be empty, which means that ε is also in the language.

b. The strings in this language have to start with a 0. The 0 may be followed by any finite number (including zero) of 0's and 1's in any order. Thus the language is the set of all strings of 0's and 1's that start with a 0. ■

Example 12.1.8 Individual Strings in the Language Defined by a Regular Expression

In each of (a) and (b), let $\Sigma = \{a, b\}$ and consider the language L over Σ defined by the given regular expression.

a. The regular expression is $a^*b(a \mid b)^*$. Write five strings that belong to L.

b. The regular expression is $a^* \mid (ab)^*$. Indicate which of the following strings belong to L:

$$a \quad b \quad aaaa \quad abba \quad ababab$$

Solution

a. The strings $b, ab, abbb, abaaa$, and $ababba$ are five strings from the infinitely many in L.

b. The following strings are the only ones listed that belong to L: $a, aaaa$, and $ababab$. The string b does not belong to L because it is neither a string of a's nor a string of possibly repeated ab's. The string $abba$ does not belong to L because any two b's that might occur in a string of L are separated by an a. ■

Example 12.1.9 A Regular Expression That Defines a Language

Let $\Sigma = \{0, 1\}$. Find regular expressions over Σ that define that following languages.

a. The language consisting of all strings of 0's and 1's that have even length and in which the 0's and 1's alternate.

b. The language consisting of all strings of 0's and 1's with an even number of 1's. Such strings are said to have *even parity*.

c. The language consisting of all strings of 0's and 1's that do not contain two consecutive 1's.

Solution

a. If a string in the language starts with a 1, the pattern 10 must continue for the length of the string. If it starts with 0, the pattern 01 must continue for the length of the string. Also, the null string satisfies the condition by default. Thus an answer is

$$(10)^* \mid (01)^*.$$

b. Basic strings with even parity are ϵ, 0, and 10^*1. Concatenation of strings with even parity also have even parity. Because such a string may start or end with a string of 0's, an answer is

$$(0 \mid 10^*1)^*.$$

c. Note that a string may end in a 1, but any other 1 must be followed immediately by a 0. Thus, it is enough to enforce the rule that a 1 must be followed by a 0, unless the 1 is at the end of the string. A regular expression satisfying these conditions is

$$(0 \mid 10)^*(\epsilon \mid 1).$$ ■

Note that a given language may be defined by more than one regular expression. For example, both

$$(a^* \mid b^*)^* \quad \text{and} \quad (a \mid b)^*$$

define the language consisting of the set of all strings of a's and b's.

Example 12.1.10 Deciding Whether Regular Expressions Define the Same Language

In (a) and (b), determine whether the given regular expressions define the same language. If they do, describe the language. If they do not, give an example of a string that is in one of the languages but not the other.

a. $(a \mid \epsilon)^*$ and a^* b. $0^* \mid 1^*$ and $(01)^*$

Solution

a. Note that because the null string ϵ has no characters, when it is concatenated with any other string x, the result is just x: for all strings x, $x\epsilon = \epsilon x = x$. Now $L((a \mid \epsilon)^*)$ is the set of strings formed using a and ϵ in any order, and so, because $a\epsilon = \epsilon a = a$, this is the same as the set of strings consisting of zero or more a's. Thus $L((a \mid \epsilon)^*) = L(a^*)$.

b. The two languages defined by the given regular expressions are not the same: 0101 is in the second language but not the first. ■

Practical Uses of Regular Expressions

Many applications of computers involve performing operations on pieces of text. For instance, word and text processing programs allow us to find certain words or phrases in a document and possibly replace them with others. A compiler for a computer language analyzes an incoming stream of characters to find groupings that represent aspects of the computer language such as keywords, constants, identifiers, and operators. And in bioinformatics, pattern matching and flexible searching techniques are used extensively to analyze the long sequences of the characters A, C, G, and T that occur in DNA.

Through their connection with finite-state automata, which we discuss in the next section, regular expressions provide an extremely useful way to describe a pattern in order to identify a string or a collection of strings within a piece of text. Regular expressions make it possible to replace a long, complicated set of if-then-else statements with code that is easy both to produce and to understand. Because of their convenience, regular expressions were introduced into a number of UNIX utilities, such as *grep* (short for ***g***lobally ***s***earch for ***r***egular ***e***xpression and ***p***rint) and *egrep* (*extended grep*), in text editors, such as *QED* (short for ***Q***uick ***ED***itor, the first text editor to use regular expressions), *vi* (short for ***v***isual ***i***nterface), *sed* (short for stream ***ed***itor and originally developed for UNIX but now used by many systems), and *Emacs* (short for ***E***ditor ***mac***ros), and in the lexical scanner component of a compiler. The computer language Perl has a particularly powerful implementation for regular expressions, which has become a de facto standard. The implementations used in Java and .NET are similar.

A number of shorthand notations have been developed to facilitate working with regular expressions in text processing. When characters in an alphabet or in a part of an alphabet are understood to occur in a standard order, the notation *[beginning character–ending character]* is commonly used to represent the regular expression that consists of a single character in the range from the beginning to the ending character. It is called a **character class.** Thus

$$[A - C] \quad \text{stands for} \quad (A \mid B \mid C)$$

and

$$[0 - 9] \quad \text{stands for} \quad (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9).$$

Character classes are also allowed to include more than one range of characters. For instance,

$$[A - C\, x - z] \quad \text{stands for} \quad (A \mid B \mid C \mid x \mid y \mid z)$$

As an example, consider the language defined by the regular expression

$$[A - Z\, a - z]([A - Z\, a - z] \mid [0 - 9])^*.$$

The following are some strings in the language:

Account Number, z23, *jsmith*109, *Draft2rev*.

In general, the language is the set of all strings that start with a letter followed by a sequence of digits or letters. This set is the same as the set of allowable identifiers in a number of computer languages.

Other commonly used shorthands are

$$[ABC] \text{ to stand for } (A \mid B \mid C)$$

and a single dot

. to stand for an arbitrary character.

Thus, for instance, if $\Sigma = \{A, B, C\}$, then

$$A.C \quad \text{stands for} \quad (AAC \mid ABC \mid ACC).$$

When the symbol ^ is placed at the beginning of a character class, it indicates that a character of the same type as those in the range of the class is to occur at that point in the string, except for one of the specific characters indicated after the ^ sign. For instance,

$$[\hat{}D - Z][0 - 9][0 - 9]^*$$

stands for any string starting with a letter of the alphabet different from D to Z, followed by any positive number of digits from 0 to 9. Examples are $B3097$, $C0046$, and so forth. If r is a regular expression, the notation $r+$ denotes the concatenation of r with itself any positive finite number of times. In symbols,

$$r+ = rr^*.$$

For example, $[A - Z]+$

represents any nonempty string of capital letters. If r is a regular expression, then

$$r? = (\epsilon \mid r).$$

That is, $r?$ denotes either zero occurrences or exactly one occurrence of r. Finally, if m and n are positive integers with $m \leq n$,

$r\{n\}$ denotes the concatenation of r with itself exactly n times,

and

$r\{m, n\}$ denotes the concatenation of r with itself anywhere from m through n times.

Thus a check to help determine whether a given string is a local telephone number in the United States is to see whether it has the form

$$[0 - 9][0 - 9][0 - 9] - [0 - 9][0 - 9][0 - 9][0 - 9],$$

or, equivalently, whether it has the form

$$[0 - 9]\{3\} - [0 - 9]\{4\}.$$

Example 12.1.11 A Regular Expression for a Date

People often write dates in a variety of formats. For instance, in the United States the following all represent the fifth of February of 2050:

2/5/2050 2-5-2050 02/05/2050 02-05-2050

Write a regular expression that would help check whether a given string might be a valid date written in one of these forms.

Note In most of the rest of the world these expressions represent the second of May of 2050.

Solution The language defined by the following regular expression consists of all strings that begin with one or two digits followed by either a hyphen or a slash, followed by either one or two digits, followed by either a hyphen or a slash, followed by four digits.

$$[0-9]\{1,2\}[-/][0-9]\{1,2\}[0-9]\{4\}$$

All valid dates of the given format are elements of the language defined by this expression, but the language also includes strings that are not valid dates. For instance, 09/54/1978 is in the language, but it is not a valid date because September does not have 54 days, and 38/12/2184 is not valid because there is no 38th month. It is possible to write a more complicated regular expression that could be used to check all aspects of the validity of a date (see exercise 40 at the end of the section), but the kind of simpler expression given above is nonetheless useful. For instance, it provides an easy way to notify a user of an interactive program that a certain kind of mistake was made and that information should be reentered. ■

Test Yourself

Answers to Test Yourself questions are located at the end of each section.

1. If x and y are strings, the concatenation of x and y is _____.
2. If L and L' are languages, the concatenation of L and L' is _____.
3. If L and L' are languages, the union of L and L' is _____.
4. If L is a language, the Kleene closure of L is _____.
5. The set of regular expressions over an alphabet Σ is defined recursively. The BASE for the definition is the statement that _____. The RECURSION for the definition specifies that if r and s are any regular expressions over Σ, then the following are also regular expressions in the set: _____, _____, and _____.
6. The function that associates a language to each regular expression over an alphabet Σ is defined recursively. The BASE for the definition is the statement that $L(\emptyset) =$ _____, $L(\epsilon) =$ _____, and $L(a) =$ _____ for every a in Σ. The RECURSION for the definition specifies that if $L(r)$ and $L(r')$ are the languages defined by the regular expression r and r' over Σ, then $L(rr') =$ _____, $L(r \mid r') =$ _____, and $L(r^*) =$ _____.
7. The notation [A - C] is an example of a _____ and denotes the regular expression _____.
8. Use of a single dot in a regular expression stands for _____.
9. The symbol ^, placed at the beginning of a character class, indicates _____.
10. If r is a regular expression, the notation $r+$ denotes _____.
11. If r is a regular expression, the notation $r?$ denotes _____.
12. If r is a regular expression, the notation $r\{n\}$ denotes _____ and the notation $r\{m, n\}$ denotes _____.

Exercise Set 12.1*

In 1 and 2 let $\Sigma = \{x, y\}$ be an alphabet.

1. a. Let L_1 be the language consisting of all strings over Σ that are palindromes and have length ≤ 4. List the elements of L_1 between braces.

b. Let L_2 be the language consisting of all strings over Σ that begin with an x and have length ≤ 3. List the elements of L_2.

2. a. Let L_3 be the language consisting of all strings over Σ of length ≤ 3 in which all the x's appear to the left of all the y's. List the elements of L_3 between braces.

b. List between braces the elements of Σ^4, the set of strings of length 4 over Σ.

c. Let $A = \Sigma^1 \cup \Sigma^2$ and $B = \Sigma^3 \cup \Sigma^4$. Describe A, B, and $A \cup B$ in words.

***H* 3.** a. If the expression $ab + cd + \cdot$ in postfix notation is converted to infix notation, what is the result?

b. Let $\Sigma = \{1, 2, *, /\}$ and let L be the set of all strings over Σ obtained by writing first a number (1 or 2), then a second number (1 or 2), which can be the same as the first one, and finally an operation (* or / where * indicates multiplication and / indicates division). Then L is a set of postfix, or reverse Polish, expressions. List all the elements of L between braces, and evaluate the resulting expressions.

*For exercises with blue numbers or letters, solutions are given in Appendix B. The symbol *H* indicates that only a hint or a partial solution is given. The symbol * signals that an exercise is more challenging than usual.

In 4–6, describe L_1L_2, $L_1 \cup L_2$, and $(L_1 \cup L_2)^*$ for the given languages L_1 and L_2.

4. L_1 is the set of all strings of a's and b's that start with an a and contain only that one a; L_2 is the set of all strings of a's and b's that contain an even number of a's.

5. L_1 is the set of all strings of a's, b's, and c's that contain no c's and have the same number of a's as b's; L_2 is the set of all strings of a's, b's, and c's that contain no a's or b's.

6. L_1 is the set of all strings of 0's and 1's that start with a 0, and L_2 is the set of all strings of 0's and 1's that end with a 0.

In 7–9, add parentheses to make the order of precedence clear in the given expressions.

7. $(a \mid b^*b)(a^* \mid ab)$

8. $0^*1 \mid 0(0^*1)^*$

9. $(x \mid yz^*)^*(yx \mid (yz)^*z)$

In 10–12 use the convention about order of precedence to eliminate the parentheses in the given regular expression.

10. $((a(b^*)) \mid (c(b^*)))\ ((ac) \mid (bc))$

11. $(1(1^*)) \mid ((1(0^*)) \mid ((1^*)1))$

12. $(xy)(((x^*)y)^*) \mid (((yx) \mid y)(y^*))$

In 13–15 use set notation to derive the language defined by the given regular expression. Assume $\Sigma = \{a, b, c\}$.

13. $\epsilon \mid ab$

14. $\emptyset \mid \epsilon$

15. $(a \mid b)c$

In 16–18 write five strings that belong to the language defined by the given regular expression.

16. $0^*1(0^*1^*)^*$

17. $b^* \mid b^*ab^*$

18. $x^*(yxxy \mid x)^*$

In 19–21 use words to describe the language defined by the given regular expression.

19. $b^*ab^*ab^*a$

20. $1(0 \mid 1)^*00$

21. $(x \mid y)y(x \mid y)^*$

In 22–24 indicate whether the given strings belong to the language defined by the given regular expression. Briefly justify your answers.

22. Expression: $(b \mid \epsilon)a(a \mid b)^*a(b \mid \epsilon)$, strings: $aaaba$, $baabb$

23. Expression: $(x^*y \mid zy^*)^*$, strings: $zyyxz$, $zyyzy$

24. Expression: $(01^*2)^*$, strings: 120, 01202

In 25–27 find a regular expression that defines the given language.

25. The language consisting of all strings of 0's and 1's with an odd number of 1's. (Such a string is said to have *odd parity*.)

26. The language consisting of all strings of a's and b's in which the third character from the end is a b.

27. The language consisting of strings of x's and y's in which the elements in every pair of x's are separated by at least one y.

Let r, s, and t be regular expressions over $\Sigma = \{a, b\}$. In 28–30 determine whether the two regular expressions define the same language. If they do, describe the language. If they do not, give an example of a string that is in one of the languages but not the other.

28. $(r \mid s)t$ and $rt \mid st$

29. $(rs)^*$ and r^*s^*

30. $(rs)^*$ and $((rs)^*)^*$

In 31–39 write a regular expression to define the given set of strings. Use the shorthand notations given in the section when ever convenient. In most cases, your expression will describe other strings in addition to the given ones, but try to make your answer fit the given strings as closely as possible within reasonable space limitations.

31. All words that are written in lower-case letters and start with the letters *pre* but do not consist of *pre* all by itself.

32. All words that are written in upper-case letters, and contain the letters *BIO* (as a unit) or *INFO* (as a unit).

33. All words that are written in lower-case letters, end in *ly*, and contain at least five letters.

34. All words that are written in lower-case letters and contain at least one of the vowels a, e, i, o, or u.

35. All words that are written in lower-case letters and contain exactly one of the vowels a, e, i, o, or u.

36. All words that are written in upper-case letters and do not start with one of the vowels A, E, I, O, or U but contain exactly two of these vowels next to each other.

37. All United States social security numbers (which consist of three digits, a hyphen, two digits, another hyphen, and finally four more digits), where the final four digits start with a 3 and end with a 6.

38. All telephone numbers that have three digits, then a hyphen, then three more digits, then a hyphen, and then four digits, where the first three digits are either 800 or 888 and the last four digits start and end with a 2.

39. All signed or unsigned numbers with or without a decimal point. A signed number has one of the prefixes + or −, and an unsigned number does not have a prefix. Represent the decimal point as \., to distinguish it from the single dot symbol for an arbitrary character.

H 40. Write a regular expression to perform a complete check to determine whether a given string represents a valid date from 1980 to 2079 written in one of the formats of Example 12.1.11. (During this period, leap years occur every four years starting in 1980.)

*41. Write a regular expression to define the set of strings of 0's and 1's with an even number of 0's and even number of 1's.

Answers for Test Yourself

1. the string obtained by writing all the characters of x followed by all the characters of y 2. $\{xy \mid x \in L \text{ and } y \in L'\}$ 3. $\{s \mid s \in L \text{ or } s \in L'\}$ 4. $\{t \mid t \text{ is a concatenation of any finite number of strings in } L\}$ 5. $\emptyset$, ϵ, and each individual symbol in Σ are regular expressions over Σ; (rs); $(r \mid s)$; (r^*) 6. $\emptyset$; $\{\epsilon\}$; $\{a\}$; $L(r)L(r')$; $L(r) \cup L(r')$; $(L(r))^*$ 7. character class; $(A \mid B \mid C)$ 8. an arbitrary character 9. a character of the same type as those in the range of the class is to occur at that point in the string except for one of the specific characters indicated after the ^ sign. 10. The concatenation of r with itself any positive finite number of times 11. $(\epsilon \mid r)$ 12. the concatenation of r with itself exactly n times; the concatenation of r with itself anywhere from m through n times

12.2 Finite-State Automata

The world of the future will be an ever more demanding struggle against the limitations of our intelligence, not a comfortable hammock in which we can lie down to be waited upon by our robot slaves. — Norbert Wiener, 1964

The kind of circuit discussed in Section 2.4 is called a *combinational circuit*. Such a circuit is characterized by the fact that its output is completely determined by its input/output table, or, in other words, by a Boolean function. Its output does not depend in any way on the history of previous inputs to the circuit. For this reason, a combinational circuit is said to have no memory.

Combinational circuits are very important in computer design, but they are not the only type of circuits used. Equally important are *sequential circuits*. For sequential circuits one cannot predict the output corresponding to a particular input unless one also knows something about the prior history of the circuit, or, more technically, unless one knows the state the circuit was in before receiving the input. The behavior of a sequential circuit is a function not only of the input to the circuit but also of the state the circuit is in when the input is received. A computer memory circuit is a type of sequential circuit.

A **finite-state automaton** (aw-TAHM-uh-tahn) is an idealized machine that embodies the essential idea of a sequential circuit. Each piece of input to a finite-state automaton leads to a change in the state of the automaton, which in turn affects how subsequent input is processed. Imagine, for example, the act of dialing a telephone number. Dialing 1–800 puts the telephone circuit in a state of readiness to receive the final seven digits of a toll-free call, whereas dialing 328 leads to a state of expectation for the four digits of a local call. Vending machines operate similarly. Just knowing that you put a quarter into a vending machine is not enough for you to be able to predict what the behavior of the machine will be. You also have to know the state the machine was in when the quarter was inserted. If 75¢ had already been deposited, you might get a beverage or some candy, but if the quarter was the first coin deposited, you would probably get nothing at all.

Example 12.2.1 A Simple Vending Machine

A simple vending machine dispenses bottles of juice that cost $1 each. The machine accepts quarters and half-dollars only and does not give change. As soon as the amount deposited equals or exceeds $1 the machine releases a bottle of juice. The next coin deposited starts the process over again. The operation of the machine is represented by the diagram of Figure 12.2.1.

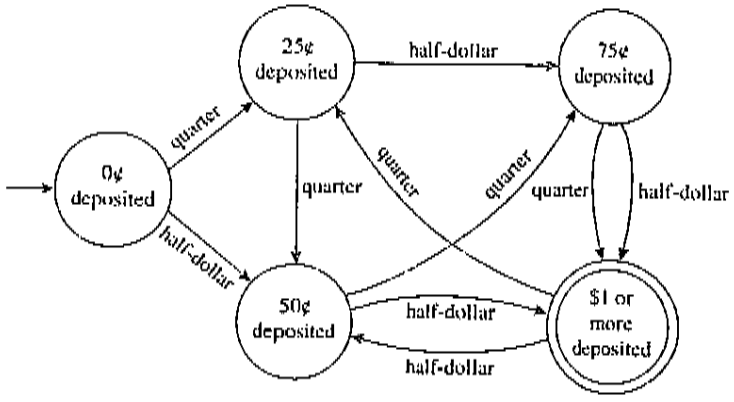


Figure 12.2.1 A Simple Vending Machine

Each circle represents a state of the machine: the state in which 0¢ has been deposited, 25¢, 50¢, 75¢, and $1 or more. The unlabeled arrow pointing to "0¢ deposited" indicates that this is the initial state of the machine. The double circle around "$1 or more deposited" indicates that a bottle of juice is released when the machine has reached this state. (It is called an *accepting state* of the machine because when the machine is in this state, it has accepted the input sequence of coins as payment for juice.) The arrows that link the states indicate what happens when a particular input is made to the machine in each of its various states. For instance, the arrow labeled "quarter" that goes from "0¢ deposited" to "25¢ deposited" indicates that when the machine is in the state "0¢ deposited" and a quarter is inserted, the machine goes to the state "25¢ deposited." The arrow labeled "half-dollar" that goes from "75¢ deposited" to "$1 or more deposited" indicates that when the machine is in the state "75¢ deposited" and a half-dollar is inserted, the machine goes to the state "$1 or more deposited" and juice is dispensed. (In this case the purchaser would pay $1.25 for the juice because the machine does not return change.) The arrow labeled "quarter" that goes from "$1 or more deposited" to "25¢ deposited" indicates that when the machine is in the state "$1 or more deposited" and a quater is inserted, the machine goes back to the state "25¢ deposited." (This corresponds to the fact that after the machine has dispensed a bottle of juice, it starts operation all over again.)

Equivalently, the operation of the vending machine can be represented by a *next-state table* as shown in Table 12.2.1.

Table 12.2.1 Next-State Table

			Input	
			Quarter	**Half-Dollar**
	→	0¢ deposited	25¢ deposited	50¢ deposited
		25¢ deposited	50¢ deposited	75¢ deposited
State		50¢ deposited	75¢ deposited	$1 or more deposited
		75¢ deposited	$1 or more deposited	$1 or more deposited
	◎	$1 or more deposited	25¢ deposited	50¢ deposited

The arrow pointing to "0¢ deposited" in the table indicates that the machine begins operation in this state. The double circle next to "$1 or more deposited" indicates that a bottle of juice is released when the machine has reached this state. Entries in the body of the table are interpreted in the obvious way. For instance, the entry in the third row of the column labeled *Half-Dollar* shows that when the machine is in state "50¢ deposited" and a half-dollar is deposited, it goes to state "$1 or more deposited."

Note that Table 12.2.1 conveys exactly the same information as the diagram of Figure 12.2.1. If the diagram is given, the table can be constructed; and if the table is given, the diagram can be drawn. ■

David Eugene Smith Collection, Rare Book and Manuscript Library, Columbia University

David Hilbert (1862–1943)

Time & Life Pictures/Getty Images

Alan M. Turing (1912–1954)

Observe that the vending machine described in Example 12.2.1 can be thought of as having a primitive memory: It "remembers" how much money has been deposited (within limits) by referring to the state it is in. This capability for storing information and acting upon it is what gives finite-state automata their tremendous power.

The most important finite-state automata are digital computers. Each computer consists of several subsystems: input devices, a processor, and output devices. A processor typically consists of a central processing unit and a finite number of memory locations. At any given time, the state of the processor is determined by the locations and values of all the bits stored within its memory. A computer that has n different locations for storing a single bit can therefore exist in 2^n different states. For a modern computer, n is many billions or even trillions, so the total number of states is enormous. But it *is* finite. Therefore, despite the complexity of a computer, just as for a vending machine, it is possible to predict the next state given knowledge of the current state and the input. Indeed, this is essentially what programmers try to do every time they write a program. Fortunately, modern, high-level computer languages provide a lot of help.

The basic theory of automata was developed to answer very theoretical questions about the foundations of mathematics posed by the great German mathematician David Hilbert in 1900. The ground-breaking work on automata was done in the mid-1930s by the English mathematician and logician Alan M. Turing. In the 1940s and 1950s, Turing's work played an important role in the development of real-world automatic computers.

Definition of a Finite-State Automaton

A general *finite-state automaton* is completely described by giving a set of states, together with an indication about which is the initial state and which are the accepting states (when something special happens), a list of all input elements, and specification for a *next-state function* that defines which state is produced by each input in each state. This is formalized in the following definition:

• Definition

A **finite-state automaton** A consists of five objects:

1. A finite set I, called the **input alphabet,** of input symbols;
2. A finite set S of **states** the automaton can be in;
3. A designated state s_0 called the **initial state;**
4. A designated set of states called the set of **accepting states;**
5. A **next-state function** $N: S \times I \rightarrow S$ that associates a "next-state" to each ordered pair consisting of a "current state" and a "current input." For each state s in S and input symbol m in I, $N(s, m)$ is the state to which A goes if m is input to A when A is in state s.

The operation of a finite-state automaton is commonly described by a diagram called a **(state-)transition diagram,** similar to that of Figure 12.2.1. It is called a *transition diagram* because it shows the transitions the machine makes from one state to another in response to various inputs. In a transition diagram, states are represented by circles and accepting states by double circles. There is one arrow that points to the initial state and other arrows that are labeled with input symbols and point from each state to other

states to indicate the action of the next-state function. Specifically, an arrow from state s to state t labeled m means that $N(s, m) = t$.

The **next-state table** for an automaton shows the values of the next-state function N for all possible states s and input symbols i. In the **annotated next-state table,** the initial state is indicated by an arrow and the accepting states are marked by double circles.

Example 12.2.2 A Finite-State Automaton Given by a Transition Diagram

Consider the finite-state automaton A defined by the transition diagram shown in Figure 12.2.2.

a. What are the states of A?

b. What are the input symbols of A?

c. What is the initial state of A?

d. What are the accepting states of A?

e. Determine $N(s_1, 1)$.

f. Construct the annotated next-state table for A.

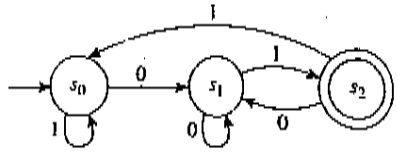


Figure 12.2.2

Solution

a. The states of A are s_0, s_1, and s_2 *[since these are the labels of the circles].*

b. The input symbols of A are 0 and 1 *[since these are the labels of the arrows].*

c. The initial state of A is s_0 *[since the unlabeled arrow points to s_0].*

d. The only accepting state of A is s_2 *[since this is the only state marked by a double circle].*

e. $N(s_1, 1) = s_2$ *[since there is an arrow from s_1 to s_2 labeled 1].*

f.

	State	Input 0	Input 1
→	s_0	s_1	s_0
	s_1	s_1	s_2
◎	s_2	s_1	s_0

■

Example 12.2.3 A Finite-State Automaton Given by an Annotated Next-State Table

Consider the finite-state automaton A defined by the following annotated next-state table:

a. What are the states of A?

b. What are the input symbols of A?

c. What is the initial state of A?

d. What are the accepting states of A?

e. Find $N(U, c)$.

f. Draw the transition diagram for A.

	State	Input a	Input b	Input c
→	U	Z	Y	Y
◎	V	V	V	V
	Y	Z	V	Y
◎	Z	Z	Z	Z

Solution

a. The states of A are U, V, Y, and Z.

b. The input symbols of A are a, b, and c.

c. The initial state of A is U *[since the arrow points to U]*.

d. The accepting states of A are V and Z *[since these are marked with double circles]*.

e. $N(U, c) = Y$ *[since the entry in the row labeled U and the column labeled c of the next-state table is Y]*.

f. The transition diagram for A is shown in Figure 12.2.3. It can be drawn more compactly by labeling arrows with multiple-input symbols where appropriate. This is illustrated in Figure 12.2.4.

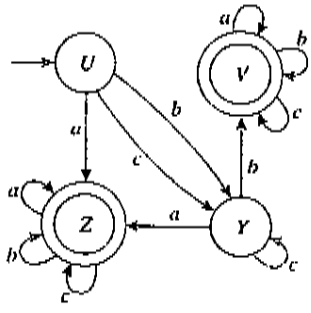


Figure 12.2.3

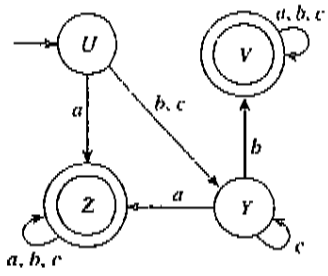


Figure 12.2.4

■

The Language Accepted by an Automaton

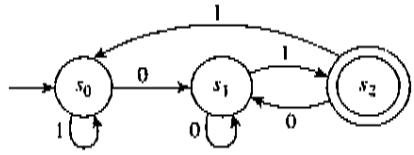
Now suppose a string of input symbols is fed into a finite-state automaton in sequence. At the end of the process, after each successive input symbol has changed the state of the automaton, the automaton ends up in a certain state, which may be either an accepting state or a nonaccepting state. In this way, the action of a finite-state automaton separates the set of all strings of input symbols into two subsets: those that send the automaton to an accepting state and those that do not. Those strings that send the automaton to an accepting state are said to be *accepted* by the automaton.

• Definition

Let A be a finite-state automaton with set of input symbols I. Let I^* be the set of all strings over I, and let w be a string in I^*. **Then w is accepted by A** if, and only if, A goes to an accepting state when the symbols of w are input to A in sequence from left to right, starting when A is in its initial state. **The language accepted by A,** denoted $L(A)$, is the set of all strings that are accepted by A.

Example 12.2.4 Finding the Language Accepted by an Automaton

Consider the finite-state automaton A defined in Example 12.2.2 and shown again below.



a. To what states does A go if the symbols of the following strings are input to A in sequence, starting from the initial state?

(i) 01 (ii) 0011 (iii) 0101100 (iv) 10101

b. Which of the strings in part (a) send A to an accepting state?

c. What is the language accepted by A?

d. Is there a regular expression that defines the same language?

Solution

a. (i) s_2 (ii) s_0 (iii) s_1 (iv) s_2

b. The strings 01 and 10101 send A to an accepting state.

c. Observe that if w is any string that ends in 01, then w is accepted by A. For if w is any string of length $n \geq 2$, then after the first $n - 2$ symbols of w have been input, A is in one of its three states: s_0, s_1, or s_2. But from any of these three states, input of the symbols 01 in sequence sends A first to s_1 and then to the accepting state s_2. Hence any string that ends in 01 is accepted by A.

Also note that the only strings accepted by A are those that end in 01. (That is, no other strings besides those ending in 01 are accepted by A.) The reason for this is that the only accepting state of A is s_2, and the only arrow pointing to s_2 comes from s_1 and is labeled 1. Thus in order for an input string w of length n to send A to an accepting state, the last symbol of w must be a 1 and the first $n - 1$ symbols of w must send A to state s_1. Now three arrows point to s_1, one from each of the three states of A, and all are labeled 0. Thus the last of the first $n - 1$ symbols of w must be 0, or, in other words, the next-to-the-last symbol of w must be 0. Hence the last two symbols of w must be 01, and thus

$$L(A) = \text{the set of all strings of 0's and 1's that end in 01.}$$

d. Yes. One regular expression that defines $L(A)$ is $(0 \mid 1)^*01$. ■

A finite-state automaton with multiple accepting states can have output devices attached to each one so that the automaton can classify input strings into a variety of different categories, one for each accepting state. This is how finite-state automata are used in the lexical scanner component of a computer compiler to group the symbols from a stream of input characters into identifiers, keywords, and so forth.

The Eventual-State Function

Now suppose a finite-state automaton is in one of its states (not necessarily the initial state) and a string of input symbols is fed into it in sequence. To what state will the automaton eventually go? The function that gives the answer to this question for every possible combination of input strings and states of the automaton is called the *eventual-state function*.

Definition

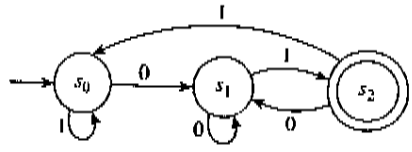
Let A be a finite-state automaton with set of input symbols I, set of states S, and next-state function N: $S \times I \to S$. Let I^* be the set of all strings over I, and define the **eventual-state function** N^*: $S \times I^* \to S$ as follows:

For any state s and for any input string w,

$$N^*(s, w) = \begin{bmatrix} \text{the state to which } A \text{ goes if the} \\ \text{symbols of } w \text{ are input to } A \text{ in sequence,} \\ \text{starting when } A \text{ is in state } s \end{bmatrix}.$$

Example 12.2.5 Computing Values of the Eventual-State Function

Consider again the finite-state automaton of Example 12.2.2 shown below for convenience. Find $N^*(s_1, 10110)$.



Solution By definition of the eventual-state function,

$$N^*(s_1, 10110) = \begin{bmatrix} \text{the state to which } A \text{ goes if the} \\ \text{symbols of 10110 are input to } A \text{ in} \\ \text{sequence, starting when } A \text{ is in state } s_1 \end{bmatrix}.$$

By referring to the transition diagram for A, you can see that starting from s_1, when a 1 is input, A goes to s_2; then when a 0 is input, A goes back to s_1; after that, when a 1 is input, A goes to s_2; from there, when a 1 is input, A goes to s_0; and finally, when a 0 is input, A goes back to s_1. This sequence of state transitions can be written as follows:

$$s_1 \xrightarrow{1} s_2 \xrightarrow{0} s_1 \xrightarrow{1} s_2 \xrightarrow{1} s_0 \xrightarrow{0} s_1.$$

Thus, after all the symbols of 10110 have been input in sequence, the eventual state of A is s_1, so

$$N^*(s_1, 10110) = s_1.$$

■

The definitions of string and language accepted by an automaton can be restated symbolically using the eventual-state function. Suppose A is a finite-state automaton with set of input symbols I and next-state function N, and suppose that I^* is the set of all strings over I and that w is a string in I^*.

w is accepted by $A \iff N^*(s_0, w)$ is an accepting state of A

$$L(A) = \{w \in I^* \mid N^*(s_0, w) \text{ is an accepting state of } A\}$$

Designing a Finite-State Automaton

Now consider the problem of starting with a description of a language and designing an automaton to accept exactly that language.

Example 12.2.6 A Finite-State Automaton That Accepts the Set of Strings of 0's and 1's for Which the Number of 1's Is Divisible by 3

a. Design a finite-state automaton A that accepts the set of all strings of 0's and 1's such that the number of 1's in the string is divisible by 3.

b. Is there a regular expression that defines this set?

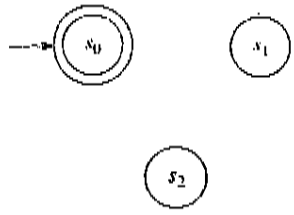
Solution

a. Let s_0 be the initial state of A, s_1 its state after one 1 has been input, and s_2 its state after two 1's have been input. Note that s_0 is the state of A after zero 1's have been input, and since zero is divisible by 3 ($0 = 0 \cdot 3$), s_0 must be an accepting state. The states s_0, s_1, and s_2 must be different from one another because from state s_0 three 1's are needed to reach a new total divisible by 3, whereas from state s_1 two additional 1's are necessary, and from state s_2 just one more 1 is required.

Now the state of A after three 1's have been input can also be taken to be s_0 because after three 1's have been input, three more are needed to reach a new total divisible by 3. More generally, if $3k$ 1's have been input to A, where k is any nonnegative integer, then three more are needed for the total again to be divisible by 3 (since $3k + 3 = 3(k + 1)$). Thus the state in which $3k$ 1's have been input, for any nonnegative integer k, can be taken to be the initial state s_0.

By similar reasoning, the states in which $(3k + 1)$ 1's and $(3k + 2)$ 1's have been input, where k is a nonnegative integer, can be taken to be s_1 and s_2, respectively.

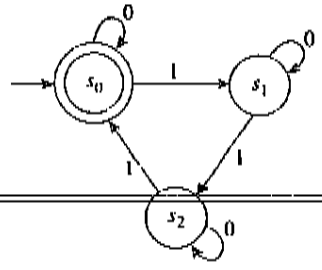
Now every nonnegative integer can be written in one of the three forms $3k$, $3k + 1$, or $3k + 2$ (see Section 4.4), so the three states s_0, s_1, and s_2 are all that is needed to create A. Thus the states of A can be drawn and labeled as shown below.



Next consider the possible inputs to A in each of its states. No matter what state A is in, if a 0 is input the total number of 1's in the input string remains unchanged. Thus there is a loop at each state labeled 0.

Now suppose a 1 is input to A when it is in state s_0. Then A goes to state s_1 (since the total number of 1's in the input string has changed from $3k$ to $3k + 1$). Similarly, if a 1 is input to A when it is in state s_1, then A goes to state s_2 (since the total number of 1's in the input string has changed from $3k + 1$ to $3k + 2$). Finally, if a 1 is input to A when it is in state s_2, then it goes to state s_0 (since the total number of 1's in the input string becomes $(3k + 2) + 1 = 3k + 3 = 3(k + 1)$, which is a multiple of 3.)

It follows that the transition diagram for A has the appearance shown below.



This automaton accepts the set of strings of 0's and 1's for which the number of 1's is divisible by 3.

b. A regular expression that defines the given set is $0^* \mid (0^*10^*10^*10^*)^*$. ■

Example 12.2.7 A Finite-State Automaton That Accepts the Set of All Strings of 0's and 1's Containing Exactly One 1

a. Design a finite-state automaton A to accept the set of all strings of 0's and 1's that contain exactly one 1.

b. Is there a regular expression that defines this set?

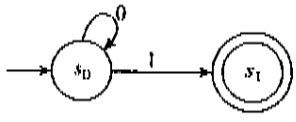
Solution

a. The automaton A must have at least two distinct states:

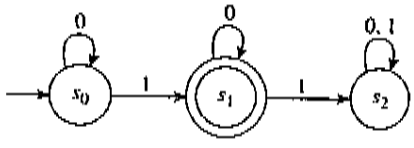
s_0: initial state;

s_1: state to which A goes when the input string contains exactly one 1.

If A is in state s_0 and a 0 is input, A may as well stay in state s_0 (since it still needs to wait for a 1 to move to state s_1), but as soon as a 1 is input, A moves to state s_1. Thus a partial drawing of the transition diagram is as shown below.



Now consider what happens when A is in state s_1. If a 0 is input, the input string still has a single 1, so A stays in state s_1. But if a 1 is input, then the input string contains more than one 1, so A must leave s_1 (since no string with more than one 1 is to be accepted by A). It cannot go back to state s_0 because there is a way to get from s_0 to s_1, and after input of the second 1, A can never return to state s_1. Hence A must go to a third state, s_2, from which there is no return to s_1. Thus from s_2 every input may as well leave A in state s_2. It follows that the completed transition diagram for A has the appearance shown below.

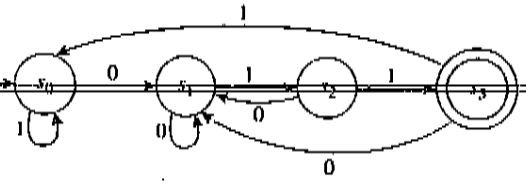


This automaton accepts the set of strings 0's and 1's, with exactly one 1.

b. A regular expression that defines the given set is 0^*10^*. ■

Simulating a Finite-State Automaton Using Software

Suppose items have been coded with strings of 0's and 1's. A program is to be written to govern the processing of items coded with strings that end 011; items coded any other way are to be ignored. This situation can be modeled by the finite-state automaton shown in Figure 12.2.5.



This automaton recognizes strings that end 011.

Figure 12.2.5

The symbols of the code for the item are fed into this automaton in sequence, and every string of symbols in a given code sends the automaton to one of the four states s_0, s_1, s_2, or s_3. If state s_3 is reached, the item is processed; if not, the item is ignored.

The action of this finite-state automaton can be simulated by a computer algorithm as given in Algorithm 12.2.1.

Algorithm 12.2.1 A Finite-State Automaton

[This algorithm simulates the action of the finite-state automaton of Figure 12.2.5 by mimicking the functioning of the transition diagram. The states are denoted 0, 1, 2, *and* 3.*]*

Input: *string [a string of* 0*'s and* 1*'s plus an end marker e]*

Algorithm Body:

```
state := 0
symbol := first symbol in the input string
while (symbol ≠ e)
    if   state = 0   then if symbol = 0
                          then state := 1
                          else state := 0
    else if state = 1 then if symbol = 0
                          then state := 1
                          else state := 2
    else if state = 2 then if symbol = 0
                          then state := 1
                          else state := 3
    else if state = 3 then if symbol = 0
                          then state := 1
                          else state := 0
    symbol := next symbol in the input string
end while
[After execution of the while loop, the value of state is 3 if, and only if, the input string ends in 011e.]
```

Output: *state*

Note how use of the finite-state automaton allows the creator of the algorithm to focus on each step of the analysis of the input string independently of the other steps.

An alternative way to program this automaton is to enter the values of the next-state function directly as a two-dimensional array. This is done in Algorithm 12.2.2.

Algorithm 12.2.2 A Finite-State Automaton

[This algorithm simulates the action of the finite-state automaton of Figure 12.2.5 by repeated application of the next-state function. The states are denoted 0, 1, 2, *and* 3.*]*

Input: *string [a string of* 0*'s and* 1*'s plus an end marker e]*

Algorithm Body:

$N(0, 0) := 1$, $N(0, 1) := 0$, $N(1, 0) := 1$, $N(1, 1) := 2$,
$N(2, 0) := 1$, $N(2, 1) := 3$, $N(3, 0) := 1$, $N(3, 1) := 0$
state : = 0
symbol : = first symbol in the input string
while (*symbol* $\neq$ *e*)
    state := *N* (*state, symbol*).
    symbol := next symbol in the input string
end while
[After execution of the **while** *loop, the value of state is* 3 *if, and only if, the input string ends in* 011*e.]*
Output: *state*

Finite-State Automata and Regular Expressions

In the previous sections, each time we considered a language accepted by a finite-state automaton, we found a regular expression that defined the same language. Stephen Kleene showed that our ability to do this is not sheer coincidence. He proved that any language accepted by a finite-state automaton can be defined by a regular expression and that, conversely, any language defined by a regular expression is accepted by a finite-state automaton. Thus for the many applications of regular expressions discussed in Section 12.1, it is theoretically possible to find a corresponding finite-state automaton, which can then be simulated using the kinds of computer algorithms described in the previous subsection.

In practice, it is often of interest to retain only pieces of the patterns sought. For instance, to obtain a reference in an HTML document, one would specify a regular expression defining the full HTML tag, <a href= "the desired URL">, but one would be interested in retrieving only the string between the quotation marks. Because of these kinds of considerations, actual implementations of finite-state automata include additional features.*

We break the statement of Kleene's theorem into two parts.

Kleene's Theorem, Part 1

Given any language that is accepted by a finite-state automaton, there is a regular expression that defines the same language.

Proof:

Suppose A is a finite-state automaton with a set I of input symbols, a set S of n states, and a next-state function N: $S \times I \to S$. Let I^* denote the set of all strings over I. Number the states $s_1, s_2, s_3, \ldots, s_n$, using s_1 to denote the initial state, and for each integer $k = 1, 2, 3, \ldots, n$, let

$$L_{i,j}^{k} = \left\{ x \in I^* \;\middle|\; \begin{array}{l} \text{when the symbols of } x \text{ are input to } A \text{ in sequence, } A \\ \text{goes from state } s_i \text{ to state } s_j \text{ without traveling through} \\ \text{an intermediate state } s_h \text{ for which } h > k \end{array} \right\}.$$

continued on page 802

*For more information, see *Mastering Regular Expressions*, 3rd ed., by Jeffrey E. F. Friedl, (Sebastopol, CA: O'Reilly & Associates, 2006).

Note that either index i or index j in $L^k_{i,j}$ could be greater than k; the only restriction is that the symbols of a string in $L^k_{i,j}$ cannot make A both enter and exit an intermediate state with index greater than k.

If s_j is an accepting state and if $k = n$ and $i = 1$, then $L^n_{1,j}$ is the set of all strings that send A to s_j when the symbols of the string are input to A in sequence starting from s_1. Thus

$$L^n_{1,j} \subseteq L(A).$$

Moreover, because the sequence of symbols in every string in $L(A)$ sends A to *some* accepting state s_j,

$L(A)$ is the union of all the sets $L^n_{1,j}$, where s_j is an accepting state.

We use a version of mathematical induction to build up a set of regular expressions over I. Let the property $P(m)$ be the sentence

For any pair of integers i and j with $1 \le i, j \le n$,
there is a regular expression $r^m_{i,j}$ that defines $L^m_{i,j}$. $\leftarrow P(m)$

Show that P(0) is true: For each pair of integers i and j with $1 \le i, j \le n$, $L^0_{i,j}$ is the set of all strings that send A from s_i to s_j without sending it through any intermediate state s_h for which $h > 0$. Because the subscript of every state in A is greater than zero, the strings in $L^0_{i,j}$ do not send A through any intermediate states at all, and so each is a single input symbol from I. In other words, for all integers i and j with $1 \le i, j \le n$,

$$L^0_{i,j} = \{a \in I \mid N(s_i, a) = s_j\}.$$

Hence $L^0_{i,j}$ is a subset of I, and so (because I is finite) we may denote the elements of $L^0_{i,j}$ as follows:

$$L^0_{i,j} = \{a_1, a_2, a_3, \ldots, a_M\} \subseteq I.$$

Now, by definition of regular expression, each single input symbol of I is a regular expression over I; thus every element of $L^0_{i,j}$ is a regular expression over I. The result is that for all integers i and j with $1 \le i, j \le n$, the following regular expression defines $L^0_{i,j}$:

$$a_1 \,|\, a_2 \,|\, a_3 | \cdots | a_M$$

Show that for all integers k with $0 \le k < n$, if P(k) is true then P(k + 1) is true: Let k be any integer with $1 \le k < n$, and suppose that

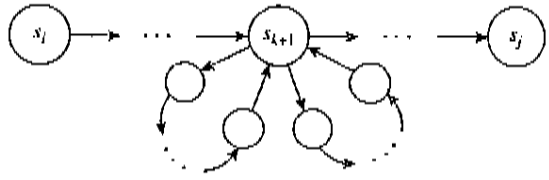
For each pair of integers p and q with $1 \le p, q \le n$, $\leftarrow P(k)$
there is a regular expression $r^k_{p,q}$ that defines $L^k_{p,q}$. inductive hypothesis

We will show that

For each pair of integers i and j with $1 \le i, j \le n$,
there is a regular expression $r^{k+1}_{i,j}$ that defines $L^{k+1}_{i,j}$. $\leftarrow P(k+1)$

So suppose that i and j are any pair of integers with $1 \le i, j \le n$, and observe that any string in $L^{k+1}_{i,j}$ sends A from s_i to s_j, either by a route that makes A pass through s_{k+1} or by a route that does not make A pass through s_{k+1}. Now each string that sends A from s_i to s_j and makes A pass through s_{k+1} one or more times can be broken into segments. The symbols in the first segment send A from s_i to s_{k+1} without making A pass through s_{k+1}; those in each of the intermediate segments send s_{k+1} to itself without making A pass through s_{k+1}; and those in the final segment send A from

s_{k+1} to s_j without making A pass through s_{k+1}. (The intermediate segment could be the null string.) A typical path showing two intermediate segments is illustrated below.



Note that each intermediate segment of the string is in $L^k_{k+1,k+1}$, and by assumption the regular expression $r^k_{k+1,k+1}$ defines this set. By the same reasoning, $r^k_{i,k+1}$ defines the set of all possible first segments of the string, and $r^k_{k+1,j}$ defines the set of all possible final segments of the string. In addition, $r^k_{i,j}$ defines the set of all strings that send A from s_i to s_j without making it pass through a state s_m with $m > k$. Thus we may define the regular expression $r^{k+1}_{i,j}$ as follows:

$$r^{k+1}_{i,j} = r^k_{i,j} \mid r^k_{i,k+1} \left(r^k_{k+1,k+1}\right)^* r^k_{k+1,j}.$$

Then $r^{k+1}_{i,j}$ defines the set of all strings that send A from s_i to s_j without making it pass through any states s_m with $m > k+1$, and so $r^{k+1}_{1,j}$ defines $L^{k+1}_{1,j}$ *[as was to be shown]*.

To complete the proof, let $s_{j_1}, s_{j_2}, \ldots, s_{j_n}$ be the accepting state of A. Because $L(A)$ is the union of all the $L^n_{1,j}$ where s_j is an accepting state, we have

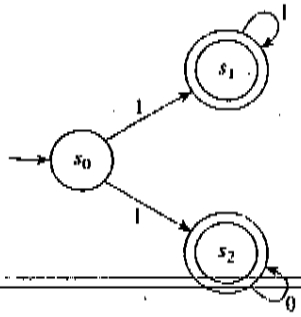
$$\begin{aligned} L(A) &= L\left(r^n_{1,j_1}\right) \cup L\left(r^n_{1,j_2} \cup \cdots \cup L\left(r^n_{1,j_n}\right)\right. \\ &= L\left(r^n_{1,j_1} \mid r^n_{1,j_2} \mid \cdots \mid r^n_{1,j_n}\right) \end{aligned}$$

by the recursive definition for the language defined by a regular expression

Thus if we let $r = r^n_{1,j_1} \left| r^n_{1,j_2} \right| \cdots \left| r^n_{1,j_n}\right.$, we have that $L(A) = L(r)$. In other words, we have constructed a regular expression r that defines the language accepted by A.

Kleene's Theorem, Part 2

Given any language defined by a regular expression, there is a finite-state automaton that accepts the same language.



The most common way to prove part 2 of Kleene's theorem is to introduce a new category of automata called *nondeterministic finite-state automata*. These are similar to the (deterministic) finite-state automata we have been discussing, except that for any given state and input symbol, the next state is a subset of the set of states of the automaton, possibly even the empty set. Thus the next state of the automaton is not uniquely determined by the combination of a current state and an input symbol. A string is accepted by a nondeterministic finite-state automaton if, and only if, when the symbols in the string are input to the automaton in sequence, starting from an initial state, there is *some* sequence of next states through which the automaton could travel that would send it to an accepting state. For instance, the transition diagram at the left is an example of a very simple nondeterministic finite-state automaton that accepts the set of all strings beginning with a 1. Observe that $N(s_0, 1) = \{s_1, s_2\}$ and $N(s_0, 0) = \emptyset$.

Given a language defined by any regular expression, there is a straightforward recursive algorithm for finding a nondeterministic finite-state automaton that defines the same

language. The proof of Kleene's theorem is completed by showing that for any such nondeterministic finite-state automaton, there is a (deterministic) finite-state automaton that defines the same language. We leave the details of the proof to a course in automata theory.

Regular Languages

According to Kleene's theorem, the set of languages defined by regular expressions is identical to the set of languages accepted by finite-state automata. Any such language is called a **regular language.** The brief allusions we made earlier to context-free languages and Chomsky's classification of languages suggest that not every language is regular. We will prove this by giving an example of a nonregular language.

To construct the example, note that because a finite-state automaton can assume only a finite number of states and because there are infinitely many input sequences, by the pigeonhole principle there must be at least one state to which the automaton returns over and over again. This is the essential feature of an automaton that makes it possible to find a nonregular language.

Example 12.2.8 Showing That a Language is Not Regular

Let the language L consist of all strings of the form $a^k b^k$, where k is a positive integer. Symbolically, L is the language over the alphabet $\Sigma = \{a, b\}$ defined by

$$L = \{s \in \Sigma^* \mid s = a^k b^k, \text{ where } k \text{ is a positive integer}\}.$$

Use the pigeonhole principle to show that L is not regular. In other words, show that there is no finite-state automaton that accepts L.

Solution *[Use a proof by contradiction.]* Suppose not. That is, suppose there is a finite-state automaton A that accepts L. *[A contradiction will be derived.]* Since A has only a finite number of states, these states can be denoted $s_1, s_2, s_3, \ldots, s_n$, where n is a positive integer. Consider all input strings that consist entirely of a's: $a, a^2, a^3, a^4, \ldots$. Now there are infinitely many such strings and only finitely many states. Thus, by the pigeonhole principle, there must be a state s_m and two input strings a^p and a^q with $p \neq q$ such that when either a^p or a^q is input to A, A goes to state s_m. (See Figure 12.2.6.) *[The pigeons are the strings of a's, the pigeonholes are the states, and the correspondence associates each string with the state to which A goes when the string is input.]*

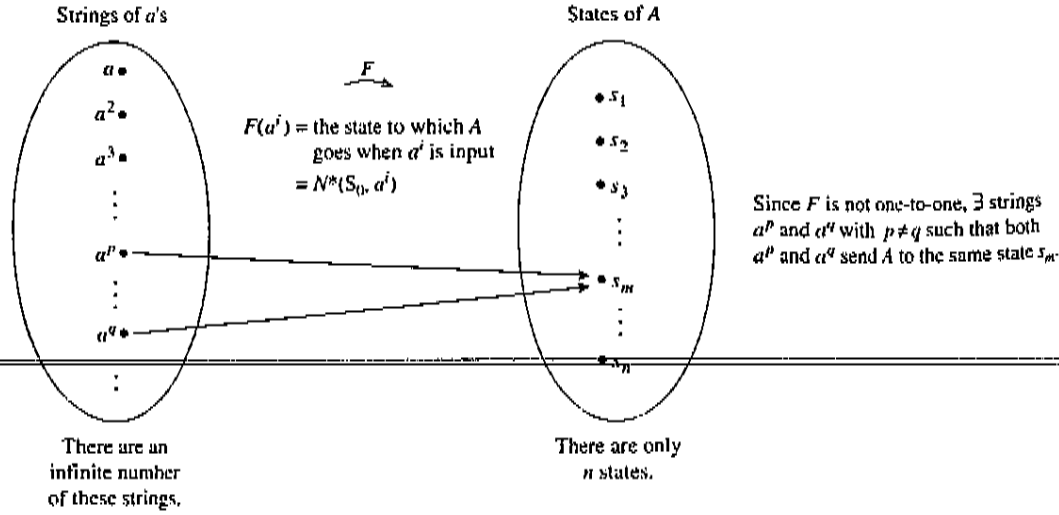


Figure 12.2.6

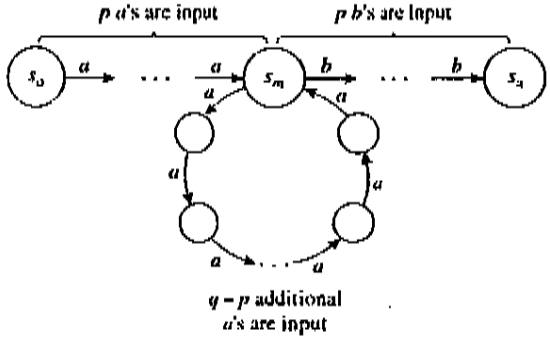
Now, by supposition, A accepts L. Hence A accepts the string

$$a^p b^p.$$

This means that after p a's have been input, at which point A is in state s_m, inputting p additional b's sends A into an accepting state, say s_u. But that implies that

$$a^q b^p$$

also sends A to the accepting state s_u, and so $a^q b^p$ is accepted by A. The reason is that after q a's have been input, A is also in state s_m, and from that point, inputting p additional b's sends A to state s_u, which is an accepting state. Pictorially, if $p < q$, then



Now, by supposition, L is the language accepted by A. Thus since s is accepted by A, $s \in L$. But by definition of L, L consists only of strings with equal numbers of a's and b's. So since $p \neq q$, $s \notin L$. Hence $s \in L$ and $s \notin L$, which is a contradiction.

It follows that the supposition is false, and so there is no finite-state automaton that accepts L. ■

Test Yourself

1. The five objects that make up a finite-state automaton are _____, _____, _____, _____, and _____.
2. The next-state table for an automaton shows the values of _____.
3. In the annotated next-state table, the initial state is indicated with an _____ and the accepting states are marked by _____.
4. A string w consisting of input symbols is accepted by a finite-state automation A if, and only if, _____.
5. The language accepted by a finite-state automaton A is _____.
6. If N is the next-state function for a finite-state automation A, the eventual-state function N^* is defined as follows: For each state s of A and for each string w that consists of input symbols of A, $N^*(s, w) =$ _____.
7. One part of Kleene's theorem says that given any language that is accepted by a finite-state automaton, there is _____.
8. The second part of Kleene's theorem says that given any language defined by a regular expression, there is _____.
9. A regular language is _____.
10. Given the language consisting of all strings of the form $a^k b^k$, where k is a positive integer, the pigeonhole principle can be used to show that the language is _____.

Exercise Set 12.2

Answers for Test Yourself

1. a finite set of input symbols; a finite set of states; a designated initial state; a designated set of accepting states; a next-state function that associates a "next-state" with each state and input symbol of the automaton 2. the next-state function for each state and input symbol of the automaton 3. arrow; double circles 4. when the symbols in the string are input to the automaton in sequence from left to right, starting from the initial state, the automaton ends up in an accepting state 5. the set of strings that are accepted by A 6. the state to which A goes if it is in state s and the characters of w are input to it in sequence 7. a regular expression that defines the same language 8. a finite-state automaton that accepts the same language 9. a language defined by a regular expression (*Or*: a language accepted by a finite-state automaton) 10. not regular

12.3 Simplifying Finite-State Automata

Our life is frittered away by detail. . . . Simplify, simplify.
— Henry David Thoreau, *Walden*, 1854

Any string input to a finite-state automaton either sends the automaton to an accepting state or not, and the set of all strings accepted by an automaton is the language accepted by the automaton. It often happens that when an automaton is created to do a certain job (as in compiler construction, for example), the automaton that emerges "naturally" from the development process is unnecessarily complicated; that is, there may be an automaton with fewer states that accepts exactly the same language. It is desirable to find such an automaton because the memory space required to store an automaton with n states is approximately proportional to n^2. Thus approximately 10,000 memory spaces are required to store an automaton with 100 states, whereas only about 100 memory spaces are needed to store an automaton with 10 states. In addition, the fewer states an automaton has, the easier it is to write a computer algorithm based on it; and to see that two automata both accept the same language, it is easiest to simplify each to a minimal number of states and compare the simplified automata. In this section we show how to take a given automaton and simplify it in the sense of finding an automaton with fewer states that accepts the same language.

Example 12.3.1 An Overview

Consider the finite-state automata A and A' in Figure 12.3.1. A moment's thought should convince you that A' accepts all those strings, and only those strings, that contain an even number of 1's. But A, although it appears more complicated, accepts exactly those

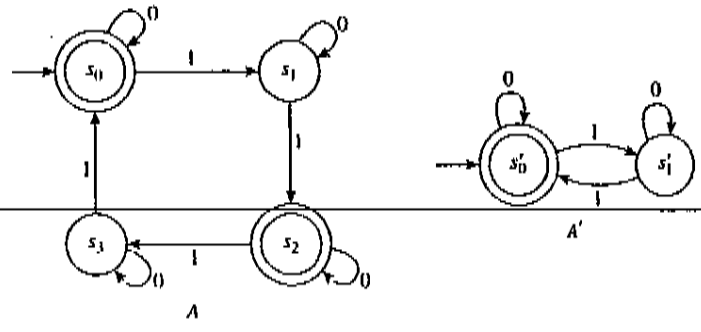


Figure 12.3.1 Two Equivalent Automata

strings also. Thus the two automata are "equivalent" in the sense that they accept the same language, even though A' has fewer states than A.

Roughly speaking, the reason for the equivalence of these automata is that some of the states of A can be combined without affecting the acceptance or nonacceptance of any input string. It turns out that s_2 can be combined with state s_0 and that s_3 can be combined with state s_1. (How to figure out which states can be combined is explained later in this section.) The automaton with the two combined states $\{s_0, s_2\}$ and $\{s_1, s_3\}$ is called the *quotient automaton* of A and is denoted $\overline{A}$. Its transition diagram is obtained by combining the circles for s_0 and s_2 and for s_1 and s_3 and by replacing any arrow from a state s to a state t by an arrow from the combined state containing s to the combined state containing t. For instance, since there is an arrow labeled 1 from s_1 to s_2 in A, there is an arrow labeled 1 from $\{s_1, s_3\}$ to $\{s_0, s_2\}$ in $\overline{A}$. The complete transition diagram for $\overline{A}$ is shown in Figure 12.3.2. As you can see, except for labeling the names of the states, it is identical to the diagram for A'.

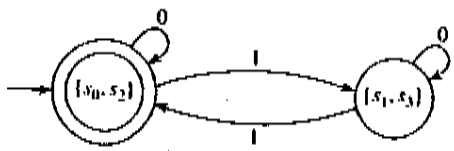


Figure 12.3.2

In general, simplification of a finite-state automaton involves identifying "equivalent states" that can be combined without affecting the action of the automaton on input strings. Mathematically speaking, this means defining an equivalence relation on the set of states of the automaton and forming a new automaton whose states are the equivalence classes of the relation. The rest of this section is devoted to developing an algorithm to carry out this process in a practical way.

Equivalence of States