Chapter 3

Regular Expressions and Languages

We begin this chapter by introducing the notation called "regular expressions." These expressions are another type of language-defining notation, which we sampled briefly in Section 1.1.2. Regular expressions also may be thought of as a "programming language," in which we express some important applications, such as text-search applications or compiler components. Regular expressions are closely related to nondeterministic finite automata and can be thought of as a "user-friendly" alternative to the NFA notation for describing software components.

In this chapter, after defining regular expressions, we show that they are capable of defining all and only the regular languages. We discuss the way that regular expressions are used in several software systems. Then, we examine the algebraic laws that apply to regular expressions. They have significant resemblance to the algebraic laws of arithmetic, yet there are also some important differences between the algebras of regular expressions and arithmetic expressions.

3.1 Regular Expressions

Now, we switch our attention from machine-like descriptions of languages — deterministic and nondeterministic finite automata — to an algebraic description: the "regular expression." We shall find that regular expressions can define exactly the same languages that the various forms of automata describe: the regular languages. However, regular expressions offer something that automata do not: a declarative way to express the strings we want to accept. Thus, regular expressions serve as the input language for many systems that process strings. Examples include:

1. Search commands such as the UNIX `grep` or equivalent commands for finding strings that one sees in Web browsers or text-formatting systems. These systems use a regular-expression-like notation for describing patterns that the user wants to find in a file. Different search systems convert the regular expression into either a DFA or an NFA, and simulate that automaton on the file being searched.

2. Lexical-analyzer generators, such as Lex or Flex. Recall that a lexical analyzer is the component of a compiler that breaks the source program into logical units (called *tokens*) of one or more characters that have a shared significance. Examples of tokens include keywords (e.g., `while`), identifiers (e.g., any letter followed by zero or more letters and/or digits), and signs, such as `+` or `<=`. A lexical-analyzer generator accepts descriptions of the forms of tokens, which are essentially regular expressions, and produces a DFA that recognizes which token appears next on the input.

3.1.1 The Operators of Regular Expressions

Regular expressions denote languages. For a simple example, the regular expression $\mathbf{01}^* + \mathbf{10}^*$ denotes the language consisting of all strings that are either a single 0 followed by any number of 1's or a single 1 followed by any number of 0's. We do not expect you to know at this point how to interpret regular expressions, so our statement about the language of this expression must be accepted on faith for the moment. We shortly shall define all the symbols used in this expression, so you can see why our interpretation of this regular expression is the correct one. Before describing the regular-expression notation, we need to learn the three operations on languages that the operators of regular expressions represent. These operations are:

1. The *union* of two languages L and M, denoted $L \cup M$, is the set of strings that are in either L or M, or both. For example, if $L = \{001, 10, 111\}$ and $M = \{\epsilon, 001\}$, then $L \cup M = \{\epsilon, 10, 001, 111\}$.

2. The *concatenation* of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M. Recall Section 1.5.2, where we defined the concatenation of a pair of strings; one string is followed by the other to form the result of the concatenation. We denote concatenation of languages either with a dot or with no operator at all, although the concatenation operator is frequently called "dot." For example, if $L = \{001, 10, 111\}$ and $M = \{\epsilon, 001\}$, then $L.M$, or just LM, is $\{001, 10, 111, 001001, 10001, 111001\}$. The first three strings in LM are the strings in L concatenated with ϵ. Since ϵ is the identity for concatenation, the resulting strings are the same as the strings of L. However, the last three strings in LM are formed by taking each string in L and concatenating it with the second string in M, which is 001. For instance, 10 from L concatenated with 001 from M gives us 10001 for LM.

3. The *closure* (or *star*, or *Kleene closure*)[1] of a language L is denoted L^* and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions (i.e., the same string may be selected more than once) and concatenating all of them. For instance, if $L = \{0, 1\}$, then L^* is all strings of 0's and 1's. If $L = \{0, 11\}$, then L^* consists of those strings of 0's and 1's such that the 1's come in pairs, e.g., 011, 11110, and ϵ, but not 01011 or 101. More formally, L^* is the infinite union $\cup_{i \geq 0} L^i$, where $L^0 = \{\epsilon\}$, $L^1 = L$, and L^i, for $i > 1$ is $LL \cdots L$ (the concatenation of i copies of L).

Example 3.1: Since the idea of the closure of a language is somewhat tricky, let us study a few examples. First, let $L = \{0, 11\}$. $L^0 = \{\epsilon\}$, independent of what language L is; the 0th power represents the selection of zero strings from L. $L^1 = L$, which represents the choice of one string from L. Thus, the first two terms in the expansion of L^* give us $\{\epsilon, 0, 11\}$.

Next, consider L^2. We pick two strings from L, with repetitions allowed, so there are four choices. These four selections give us $L^2 = \{00, 011, 110, 1111\}$. Similarly, L^3 is the set of strings that may be formed by making three choices of the two strings in L and gives us

$$\{000, 0011, 0110, 1100, 01111, 11011, 11110, 111111\}$$

To compute L^*, we must compute L^i for each i, and take the union of all these languages. L^i has 2^i members. Although each L^i is finite, the union of the infinite number of terms L^i is generally an infinite language, as it is in our example.

Now, let L be the set of all strings of 0's. Note that L is infinite, unlike our previous example, which is a finite language. However, it is not hard to discover what L^* is. $L^0 = \{\epsilon\}$, as always. $L^1 = L$. L^2 is the set of strings that can be formed by taking one string of 0's and concatenating it with another string of 0's. The result is still a string of 0's. In fact, every string of 0's can be written as the concatenation of two strings of 0's (don't forget that ϵ is a "string of 0's"; this string can always be one of the two strings that we concatenate). Thus, $L^2 = L$. Likewise, $L^3 = L$, and so on. Thus, the infinite union $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$ is L in the particular case that the language L is the set of all strings of 0's.

For a final example, $\emptyset^* = \{\epsilon\}$. Note that $\emptyset^0 = \{\epsilon\}$, while $\emptyset^i$, for any $i \geq 1$, is empty, since we can't select any strings from the empty set. In fact, $\emptyset$ is one of only two languages whose closure is *not* infinite. $\Box$

3.1.2 Building Regular Expressions

Algebras of all kinds start with some elementary expressions, usually constants and/or variables. Algebras then allow us to construct more expressions by

[1]The term "Kleene closure" refers to S. C. Kleene, who originated the regular expression notation and this operator.

Use of the Star Operator

We saw the star operator first in Section 1.5.2, where we applied it to an alphabet, e.g., Σ^*. That operator formed all strings whose symbols were chosen from alphabet Σ. The closure operator is essentially the same, although there is a subtle distinction of types.

Suppose L is the language containing strings of length 1, and for each symbol a in Σ there is a string a in L. Then, although L and Σ "look" the same, they are of different types; L is a set of strings, and Σ is a set of symbols. On the other hand, L^* denotes the same language as Σ^*.

applying a certain set of operators to these elementary expressions and to previously constructed expressions. Usually, some method of grouping operators with their operands, such as parentheses, is required as well. For instance, the familiar arithmetic algebra starts with constants such as integers and real numbers, plus variables, and builds more complex expressions with arithmetic operators such as $+$ and $\times$.

The algebra of regular expressions follows this pattern, using constants and variables that denote languages, and operators for the three operations of Section 3.1.1 —union, dot, and star. We can describe the regular expressions recursively, as follows. In this definition, we not only describe what the legal regular expressions are, but for each regular expression E, we describe the language it represents, which we denote $L(E)$.

BASIS: The basis consists of three parts:

1. The constants ϵ and $\emptyset$ are regular expressions, denoting the languages $\{\epsilon\}$ and $\emptyset$, respectively. That is, $L(\epsilon) = \{\epsilon\}$, and $L(\emptyset) = \emptyset$.

2. If a is any symbol, then $\mathbf{a}$ is a regular expression. This expression denotes the language $\{a\}$. That is, $L(\mathbf{a}) = \{a\}$. Note that we use boldface font to denote an expression corresponding to a symbol. The correspondence, e.g. that $\mathbf{a}$ refers to a, should be obvious.

3. A variable, usually capitalized and italic such as L, is a variable, representing any language.

INDUCTION: There are four parts to the inductive step, one for each of the three operators and one for the introduction of parentheses.

1. If E and F are regular expressions, then $E + F$ is a regular expression denoting the union of $L(E)$ and $L(F)$. That is, $L(E+F) = L(E) \cup L(F)$.

2. If E and F are regular expressions, then EF is a regular expression denoting the concatenation of $L(E)$ and $L(F)$. That is, $L(EF) = L(E)L(F)$.

Expressions and Their Languages

Strictly speaking, a regular expression E is just an expression, not a language. We should use $L(E)$ when we want to refer to the language that E denotes. However, it is common usage to refer to say "E" when we really mean "$L(E)$." We shall use this convention as long as it is clear we are talking about a language and not about a regular expression.

Note that the dot can optionally be used to denote the concatenation operator, either as an operation on languages or as the operator in a regular expression. For instance, $\mathbf{0.1}$ is a regular expression meaning the same as $\mathbf{01}$ and representing the language $\{01\}$. However, we shall avoid the dot as concatenation in regular expressions.[2]

3. If E is a regular expression, then E^* is a regular expression, denoting the closure of $L(E)$. That is, $L(E^*) = \big(L(E)\big)^*$.

4. If E is a regular expression, then (E), a parenthesized E, is also a regular expression, denoting the same language as E. Formally; $L\big((E)\big) = L(E)$.

Example 3.2: Let us write a regular expression for the set of strings that consist of alternating 0's and 1's. First, let us develop a regular expression for the language consisting of the single string 01. We can then use the star operator to get an expression for all strings of the form $0101 \cdots 01$.

The basis rule for regular expressions tells us that $\mathbf{0}$ and $\mathbf{1}$ are expressions denoting the languages $\{0\}$ and $\{1\}$, respectively. If we concatenate the two expressions, we get a regular expression for the language $\{01\}$; this expression is $\mathbf{01}$. As a general rule, if we want a regular expression for the language consisting of only the string w, we use w itself as the regular expression. Note that in the regular expression, the symbols of w will normally be written in boldface, but the change of font is only to help you distinguish expressions from strings and should not be taken as significant.

Now, to get all strings consisting of zero or more occurrences of 01, we use the regular expression $(\mathbf{01})^*$. Note that we first put parentheses around $\mathbf{01}$, to avoid confusing with the expression $\mathbf{01}^*$, whose language is all strings consisting of a 0 and any number of 1's. The reason for this interpretation is explained in Section 3.1.3, but briefly, star takes precedence over dot, and therefore the argument of the star is selected before performing any concatenations.

However, $L\big((\mathbf{01})^*\big)$ is not exactly the language that we want. It includes only those strings of alternating 0's and 1's that begin with 0 and end with 1. We also need to consider the possibility that there is a 1 at the beginning and/or

[2]In fact, UNIX regular expressions use the dot for an entirely different purpose: representing any ASCII character.

a 0 at the end. One approach is to construct three more regular expressions that handle the other three possibilities. That is, $(\mathbf{10})^*$ represents those alternating strings that begin with 1 and end with 0, while $\mathbf{0}(\mathbf{10})^*$ can be used for strings that both begin and end with 0 and $\mathbf{1}(\mathbf{01})^*$ serves for strings that begin and end with 1. The entire regular expression is

$$(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^*$$

Notice that we use the + operator to take the union of the four languages that together give us all the strings with alternating 0's and 1's.

However, there is another approach that yields a regular expression that looks rather different and is also somewhat more succinct. Start again with the expression $(\mathbf{01})^*$. We can add an optional 1 at the beginning if we concatenate on the left with the expression $\epsilon + \mathbf{1}$. Likewise, we add an optional 0 at the end with the expression $\epsilon + \mathbf{0}$. For instance, using the definition of the + operator:

$$L(\epsilon + \mathbf{1}) = L(\epsilon) \cup L(\mathbf{1}) = \{\epsilon\} \cup \{1\} = \{\epsilon, 1\}$$

If we concatenate this language with any other language L, the ϵ choice gives us all the strings in L, while the 1 choice gives us $1w$ for every string w in L. Thus, another expression for the set of strings that alternate 0's and 1's is:

$$(\epsilon + \mathbf{1})(\mathbf{01})^*(\epsilon + \mathbf{0})$$

Note that we need parentheses around each of the added expressions, to make sure the operators group properly. $\square$

3.1.3 Precedence of Regular-Expression Operators

Like other algebras, the regular-expression operators have an assumed order of "precedence," which means that operators are associated with their operands in a particular order. We are familiar with the notion of precedence from ordinary arithmetic expressions. For instance, we know that $xy+z$ groups the product xy before the sum, so it is equivalent to the parenthesized expression $(xy) + z$ and not to the expression $x(y + z)$. Similarly, we group two of the same operators from the left in arithmetic, so $x - y - z$ is equivalent to $(x - y) - z$, and not to $x - (y - z)$. For regular expressions, the following is the order of precedence for the operators:

1. The star operator is of highest precedence. That is, it applies only to the smallest sequence of symbols to its left that is a well-formed regular expression.

2. Next in precedence comes the concatenation or "dot" operator. After grouping all stars to their operands, we group concatenation operators to their operands. That is, all expressions that are *juxtaposed* (adjacent, with no intervening operator) are grouped together. Since concatenation

is an associative operator it does not matter in what order we group consecutive concatenations, although if there is a choice to be made, you should group them from the left. For instance, $\mathbf{012}$ is grouped $(\mathbf{01})\mathbf{2}$.

3. Finally, all unions (+ operators) are grouped with their operands. Since union is also associative, it again matters little in which order consecutive unions are grouped, but we shall assume grouping from the left.

Of course, sometimes we do not want the grouping in a regular expression to be as required by the precedence of the operators. If so, we are free to use parentheses to group operands exactly as we choose. In addition, there is never anything wrong with putting parentheses around operands that you want to group, even if the desired grouping is implied by the rules of precedence.

Example 3.3: The expression $\mathbf{01}^* + \mathbf{1}$ is grouped $\big(\mathbf{0}(\mathbf{1}^*)\big) + \mathbf{1}$. The star operator is grouped first. Since the symbol $\mathbf{1}$ immediately to its left is a legal regular expression, that alone is the operand of the star. Next, we group the concatenation between $\mathbf{0}$ and $(\mathbf{1}^*)$, giving us the expression $\big(\mathbf{0}(\mathbf{1}^*)\big)$. Finally, the union operator connects the latter expression and the expression to its right, which is $\mathbf{1}$.

Notice that the language of the given expression, grouped according to the precedence rules, is the string 1 plus all strings consisting of a 0 followed by any number of 1's (including none). Had we chosen to group the dot before the star, we could have used parentheses, as $(\mathbf{01})^* + \mathbf{1}$. The language of this expression is the string 1 and all strings that repeat 01, zero or more times. Had we wished to group the union first, we could have added parentheses around the union to make the expression $\mathbf{0}(\mathbf{1}^* + \mathbf{1})$. That expression's language is the set of strings that begin with 0 and have any number of 1's following. □

3.1.4 Exercises for Section 3.1

Exercise 3.1.1: Write regular expressions for the following languages:

* a) The set of strings over alphabet $\{a, b, c\}$ containing at least one a and at least one b.

b) The set of strings of 0's and 1's whose tenth symbol from the right end is 1.

c) The set of strings of 0's and 1's with at most one pair of consecutive 1's.

! **Exercise 3.1.2:** Write regular expressions for the following languages:

* a) The set of all strings of 0's and 1's such that every pair of adjacent 0's appears before any pair of adjacent 1's.

b) The set of strings of 0's and 1's whose number of 0's is divisible by five.

!! **Exercise 3.1.3:** Write regular expressions for the following languages:

a) The set of all strings of 0's and 1's not containing 101 as a substring.

b) The set of all strings with an equal number of 0's and 1's, such that no prefix has two more 0's than 1's, nor two more 1's than 0's.

c) The set of strings of 0's and 1's whose number of 0's is divisible by five and whose number of 1's is even.

! **Exercise 3.1.4:** Give English descriptions of the languages of the following regular expressions:

* a) $(\mathbf{1}+\epsilon)(\mathbf{00}^*\mathbf{1})^*\mathbf{0}^*$.

b) $(\mathbf{0}^*\mathbf{1}^*)^*\mathbf{000}(\mathbf{0}+\mathbf{1})^*$.

c) $(\mathbf{0}+\mathbf{10})^*\mathbf{1}^*$.

*! **Exercise 3.1.5:** In Example 3.1 we pointed out that $\emptyset$ is one of two languages whose closure is finite. What is the other?

3.2 Finite Automata and Regular Expressions

While the regular-expression approach to describing languages is fundamentally different from the finite-automaton approach, these two notations turn out to represent exactly the same set of languages, which we have termed the "regular languages." We have already shown that deterministic finite automata, and the two kinds of nondeterministic finite automata — with and without ϵ-transitions — accept the same class of languages. In order to show that the regular expressions define the same class, we must show that:

1. Every language defined by one of these automata is also defined by a regular expression. For this proof, we can assume the language is accepted by some DFA.

2. Every language defined by a regular expression is defined by one of these automata. For this part of the proof, the easiest is to show that there is an NFA with ϵ-transitions accepting the same language.

Figure 3.1 shows all the equivalences we have proved or will prove. An arc from class X to class Y means that we prove every language defined by class X is also defined by class Y. Since the graph is strongly connected (i.e., we can get from each of the four nodes to any other node) we see that all four classes are really the same.

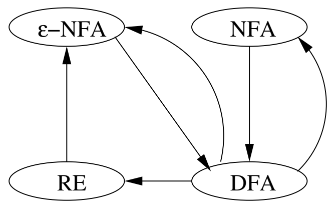

Figure 3.1: Plan for showing the equivalence of four different notations for regular languages

3.2.1 From DFA's to Regular Expressions

The construction of a regular expression to define the language of any DFA is surprisingly tricky. Roughly, we build expressions that describe sets of strings that label certain paths in the DFA's transition diagram. However, the paths are allowed to pass through only a limited subset of the states. In an inductive definition of these expressions, we start with the simplest expressions that describe paths that are not allowed to pass through *any* states (i.e., they are single nodes or single arcs), and inductively build the expressions that let the paths go through progressively larger sets of states. Finally, the paths are allowed to go through any state; i.e., the expressions we generate at the end represent all possible paths. These ideas appear in the proof of the following theorem.

Theorem 3.4: If $L = L(A)$ for some DFA A, then there is a regular expression R such that $L = L(R)$.

PROOF: Let us suppose that A's states are $\{1, 2, \ldots, n\}$ for some integer n. No matter what the states of A actually are, there will be n of them for some finite n, and by renaming the states, we can refer to the states in this manner, as if they were the first n positive integers. Our first, and most difficult, task is to construct a collection of regular expressions that describe progressively broader sets of paths in the transition diagram of A.

Let us use $R_{ij}^{(k)}$ as the name of a regular expression whose language is the set of strings w such that w is the label of a path from state i to state j in A, and that path has no intermediate node whose number is greater than k. Note that the beginning and end points of the path are not "intermediate," so there is no constraint that i and/or j be less than or equal to k.

Figure 3.2 suggests the requirement on the paths represented by $R_{ij}^{(k)}$. There, the vertical dimension represents the state, from 1 at the bottom to n at the top, and the horizontal dimension represents travel along the path. Notice that in this diagram we have shown both i and j to be greater than k, but either or both could be k or less. Also notice that the path passes through node k twice, but never goes through a state higher than k, except at the endpoints.

To construct the expressions $R_{ij}^{(k)}$, we use the following inductive definition, starting at $k = 0$ and finally reaching $k = n$. Notice that when $k = n$, there is

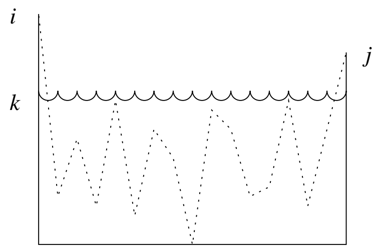

Figure 3.2: A path whose label is in the language of regular expression $R_{ij}^{(k)}$

no restriction at all on the paths represented, since there *are* no states greater than n.

BASIS: The basis is $k = 0$. Since all states are numbered 1 or above, the restriction on paths is that the path must have no intermediate states at all. There are only two kinds of paths that meet such a condition:

1. An arc from node (state) i to node j.

2. A path of length 0 that consists of only some node i.

If $i \neq j$, then only case (1) is possible. We must examine the DFA A and find those input symbols a such that there is a transition from state i to state j on symbol a.

a) If there is no such symbol a, then $R_{ij}^{(0)} = \emptyset$.

b) If there is exactly one such symbol a, then $R_{ij}^{(0)} = \mathbf{a}$.

c) If there are symbols $a_1, a_2, \ldots, a_k$ that label arcs from state i to state j, then $R_{ij}^{(0)} = \mathbf{a}_1 + \mathbf{a}_2 + \cdots + \mathbf{a}_k$.

However, if $i = j$, then the legal paths are the path of length 0 and all loops from i to itself. The path of length 0 is represented by the regular expression ϵ, since that path has no symbols along it. Thus, we add ϵ to the various expressions devised in (a) through (c) above. That is, in case (a) [no symbol a] the expression becomes ϵ, in case (b) [one symbol a] the expression becomes $\epsilon + \mathbf{a}$, and in case (c) [multiple symbols] the expression becomes $\epsilon + \mathbf{a}_1 + \mathbf{a}_2 + \cdots + \mathbf{a}_k$.

INDUCTION: Suppose there is a path from state i to state j that goes through no state higher than k. There are two possible cases to consider:

1. The path does not go through state k at all. In this case, the label of the path is in the language of $R_{ij}^{(k-1)}$.

2. The path goes through state k at least once. Then we can break the path into several pieces, as suggested by Fig. 3.3. The first goes from state i to state k without passing through k, the last piece goes from k to j without passing through k, and all the pieces in the middle go from k to itself, without passing through k. Note that if the path goes through state k only once, then there are no "middle" pieces, just a path from i to k and a path from k to j. The set of labels for all paths of this type is represented by the regular expression $R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$. That is, the first expression represents the part of the path that gets to state k the first time, the second represents the portion that goes from k to itself, zero times, once, or more than once, and the third expression represents the part of the path that leaves k for the last time and goes to state j.

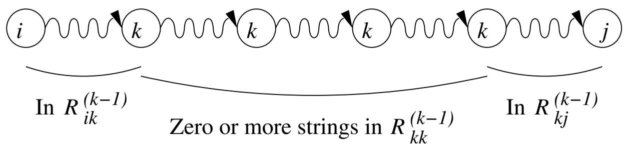

Figure 3.3: A path from i to j can be broken into segments at each point where it goes through state k

When we combine the expressions for the paths of the two types above, we have the expression

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

for the labels of all paths from state i to state j that go through no state higher than k. If we construct these expressions in order of increasing superscript, then since each $R_{ij}^{(k)}$ depends only on expressions with a smaller superscript, then all expressions are available when we need them.

Eventually, we have $R_{ij}^{(n)}$ for all i and j. We may assume that state 1 is the start state, although the accepting states could be any set of the states. The regular expression for the language of the automaton is then the sum (union) of all expressions $R_{1j}^{(n)}$ such that state j is an accepting state. $\square$

Example 3.5 : Let us convert the DFA of Fig. 3.4 to a regular expression. This DFA accepts all strings that have at least one 0 in them. To see why, note that the automaton goes from the start state 1 to accepting state 2 as soon as it sees an input 0. The automaton then stays in state 2 on all input sequences.

Below are the basis expressions in the construction of Theorem 3.4.

$R_{11}^{(0)}$	$\epsilon + \mathbf{1}$
$R_{12}^{(0)}$	$\mathbf{0}$
$R_{21}^{(0)}$	$\emptyset$
$R_{22}^{(0)}$	$(\epsilon + \mathbf{0} + \mathbf{1})$

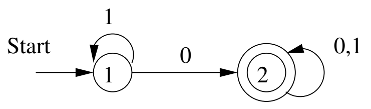

Figure 3.4: A DFA accepting all strings that have at least one 0

For instance, $R_{11}^{(0)}$ has the term ϵ because the beginning and ending states are the same, state 1. It has the term **1** because there is an arc from state 1 to state 1 on input 1. As another example, $R_{12}^{(0)}$ is **0** because there is an arc labeled 0 from state 1 to state 2. There is no ϵ term because the beginning and ending states are different. For a third example, $R_{21}^{(0)} = \emptyset$, because there is no arc from state 2 to state 1.

Now, we must do the induction part, building more complex expressions that first take into account paths that go through state 1, and then paths that can go through states 1 and 2, i.e., any path. The rule for computing the expressions $R_{ij}^{(1)}$ are instances of the general rule given in the inductive part of Theorem 3.4:

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}(R_{11}^{(0)})^* R_{1j}^{(0)} \tag{3.1}$$

The table in Fig. 3.5 gives first the expressions computed by direct substitution into the above formula, and then a simplified expression that we can show, by ad-hoc reasoning, to represent the same language as the more complex expression.

	By direct substitution	Simplified
$R_{11}^{(1)}$	$\epsilon + \mathbf{1} + (\epsilon + \mathbf{1})(\epsilon + \mathbf{1})^*(\epsilon + \mathbf{1})$	$\mathbf{1}^*$
$R_{12}^{(1)}$	$\mathbf{0} + (\epsilon + \mathbf{1})(\epsilon + \mathbf{1})^*\mathbf{0}$	$\mathbf{1}^*\mathbf{0}$
$R_{21}^{(1)}$	$\emptyset + \emptyset(\epsilon + \mathbf{1})^*(\epsilon + \mathbf{1})$	$\emptyset$
$R_{22}^{(1)}$	$\epsilon + \mathbf{0} + \mathbf{1} + \emptyset(\epsilon + \mathbf{1})^*\mathbf{0}$	$\epsilon + \mathbf{0} + \mathbf{1}$

Figure 3.5: Regular expressions for paths that can go through only state 1

For example, consider $R_{12}^{(1)}$. Its expression is $R_{12}^{(0)} + R_{11}^{(0)}(R_{11}^{(0)})^* R_{12}^{(0)}$, which we get from (3.1) by substituting $i = 1$ and $j = 2$.

To understand the simplification, note the general principle that if R is any regular expression, then $(\epsilon + R)^* = R^*$. The justification is that both sides of the equation describe the language consisting of any concatenation of zero or more strings from $L(R)$. In our case, we have $(\epsilon + \mathbf{1})^* = \mathbf{1}^*$; notice that both expressions denote any number of 1's. Further, $(\epsilon + \mathbf{1})\mathbf{1}^* = \mathbf{1}^*$. Again, it can be observed that both expressions denote "any number of 1's." Thus, the original expression $R_{12}^{(1)}$ is equivalent to $\mathbf{0} + \mathbf{1}^*\mathbf{0}$. This expression denotes the language containing the string 0 and all strings consisting of a 0 preceded by any number of 1's. This language is also expressed by the simpler expression $\mathbf{1}^*\mathbf{0}$.

The simplification of $R_{11}^{(1)}$ is similar to the simplification of $R_{12}^{(1)}$ that we just considered. The simplification of $R_{21}^{(1)}$ and $R_{22}^{(1)}$ depends on two rules about how $\emptyset$ operates. For any regular expression R:

1. $\emptyset R = R\emptyset = \emptyset$. That is, $\emptyset$ is an *annihilator* for concatenation; it results in itself when concatenated, either on the left or right, with any expression. This rule makes sense, because for a string to be in the result of a concatenation, we must find strings from both arguments of the concatenation. Whenever one of the arguments is $\emptyset$, it will be impossible to find a string from that argument.

2. $\emptyset + R = R + \emptyset = R$. That is, $\emptyset$ is the identity for union; it results in the other expression whenever it appears in a union.

As a result, an expression like $\emptyset(\epsilon + \mathbf{1})^*(\epsilon + \mathbf{1})$ can be replaced by $\emptyset$. The last two simplifications should now be clear.

Now, let us compute the expressions $R_{ij}^{(2)}$. The inductive rule applied with $k = 2$ gives us:

$$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)}(R_{22}^{(1)})^* R_{2j}^{(1)} \tag{3.2}$$

If we substitute the simplified expressions from Fig. 3.5 into (3.2), we get the expressions of Fig. 3.6. That figure also shows simplifications following the same principles that we described for Fig. 3.5.

	By direct substitution	Simplified
$R_{11}^{(2)}$	$\mathbf{1}^* + \mathbf{1}^*\mathbf{0}(\epsilon + \mathbf{0} + \mathbf{1})^*\emptyset$	$\mathbf{1}^*$
$R_{12}^{(2)}$	$\mathbf{1}^*\mathbf{0} + \mathbf{1}^*\mathbf{0}(\epsilon + \mathbf{0} + \mathbf{1})^*(\epsilon + \mathbf{0} + \mathbf{1})$	$\mathbf{1}^*\mathbf{0}(\mathbf{0} + \mathbf{1})^*$
$R_{21}^{(2)}$	$\emptyset + (\epsilon + \mathbf{0} + \mathbf{1})(\epsilon + \mathbf{0} + \mathbf{1})^*\emptyset$	$\emptyset$
$R_{22}^{(2)}$	$\epsilon + \mathbf{0} + \mathbf{1} + (\epsilon + \mathbf{0} + \mathbf{1})(\epsilon + \mathbf{0} + \mathbf{1})^*(\epsilon + \mathbf{0} + \mathbf{1})$	$(\mathbf{0} + \mathbf{1})^*$

Figure 3.6: Regular expressions for paths that can go through any state

The final regular expression equivalent to the automaton of Fig. 3.4 is constructed by taking the union of all the expressions where the first state is the start state and the second state is accepting. In this example, with 1 as the start state and 2 as the only accepting state, we need only the expression $R_{12}^{(2)}$. This expression is $\mathbf{1}^*\mathbf{0}(\mathbf{0} + \mathbf{1})^*$. It is simple to interpret this expression. Its language consists of all strings that begin with zero or more 1's, then have a 0, and then any string of 0's and 1's. Put another way, the language is all strings of 0's and 1's with at least one 0. $\square$

3.2.2 Converting DFA's to Regular Expressions by Eliminating States

The method of Section 3.2.1 for converting a DFA to a regular expression always works. In fact, as you may have noticed, it doesn't really depend on the automaton being deterministic, and could just as well have been applied to an NFA or even an ϵ-NFA. However, the construction of the regular expression is expensive. Not only do we have to construct about n^3 expressions for an n-state automaton, but the length of the expression can grow by a factor of 4 on the average, with each of the n inductive steps, if there is no simplification of the expressions. Thus, the expressions themselves could reach on the order of 4^n symbols.

There is a similar approach that avoids duplicating work at some points. For example, for every i and j, the formula for $R_{ij}^{(k)}$ in the construction of Theorem 3.4 uses the subexpression $(R_{kk}^{(k-1)})^*$; the work of writing that expression is therefore repeated n^2 times.

The approach to constructing regular expressions that we shall now learn involves eliminating states. When we eliminate a state s, all the paths that went through s no longer exist in the automaton. If the language of the automaton is not to change, we must include, on an arc that goes directly from q to p, the labels of paths that went from some state q to state p, through s. Since the label of this arc may now involve strings, rather than single symbols, and there may even be an infinite number of such strings, we cannot simply list the strings as a label. Fortunately, there is a simple, finite way to represent all such strings: use a regular expression.

Thus, we are led to consider automata that have regular expressions as labels. The language of the automaton is the union over all paths from the start state to an accepting state of the language formed by concatenating the languages of the regular expressions along that path. Note that this rule is consistent with the definition of the language for any of the varieties of automata we have considered so far. Each symbol a, or ϵ if it is allowed, can be thought of as a regular expression whose language is a single string, either $\{a\}$ or $\{\epsilon\}$. We may regard this observation as the basis of a state-elimination procedure, which we describe next.

Figure 3.7 shows a generic state s about to be eliminated. We suppose that the automaton of which s is a state has predecessor states $q_1, q_2, \ldots, q_k$ for s and successor states $p_1, p_2, \ldots, p_m$ for s. It is possible that some of the q's are also p's, but we assume that s is not among the q's or p's, even if there is a loop from s to itself, as suggested by Fig. 3.7. We also show a regular expression on each arc from one of the q's to s; expression Q_i labels the arc from q_i. Likewise, we show a regular expression P_i labeling the arc from s to p_i, for all i. We show a loop on s with label S. Finally, there is a regular expression R_{ij} on the arc from q_i to p_j, for all i and j. Note that some of these arcs may not exist in the automaton, in which case we take the expression on that arc to be $\emptyset$.

Figure 3.8 shows what happens when we eliminate state s. All arcs involving

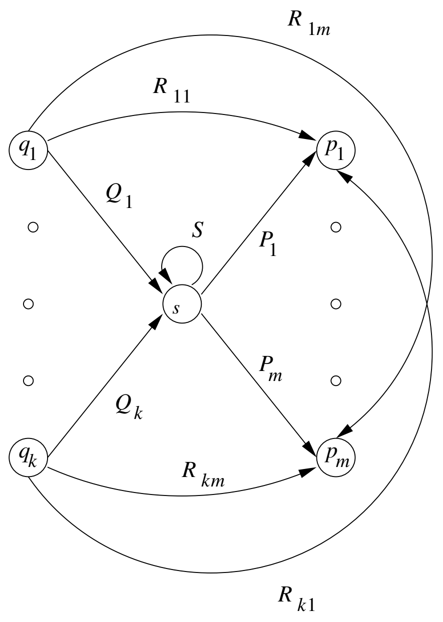

Figure 3.7: A state s about to be eliminated

state s are deleted. To compensate, we introduce, for each predecessor q_i of s and each successor p_j of s, a regular expression that represents all the paths that start at q_i, go to s, perhaps loop around s zero or more times, and finally go to p_j. The expression for these paths is $Q_i S^* P_j$. This expression is added (with the union operator) to the arc from q_i to p_j. If there was no arc $q_i \to p_j$, then first introduce one with regular expression $\emptyset$.

The strategy for constructing a regular expression from a finite automaton is as follows:

1. For each accepting state q, apply the above reduction process to produce an equivalent automaton with regular-expression labels on the arcs. Eliminate all states except q and the start state q_0.

2. If $q \neq q_0$, then we shall be left with a two-state automaton that looks like Fig. 3.9. The regular expression for the accepted strings can be described in various ways. One is $(R + SU^*T)^*SU^*$. In explanation, we can go from the start state to itself any number of times, by following a sequence of paths whose labels are in either $L(R)$ or $L(SU^*T)$. The expression SU^*T represents paths that go to the accepting state via a path in $L(S)$, perhaps return to the accepting state several times using a sequence of paths with labels in $L(U)$, and then return to the start state with a path whose label is in $L(T)$. Then we must go to the accepting state, never to return to the start state, by following a path with a label in $L(S)$. Once

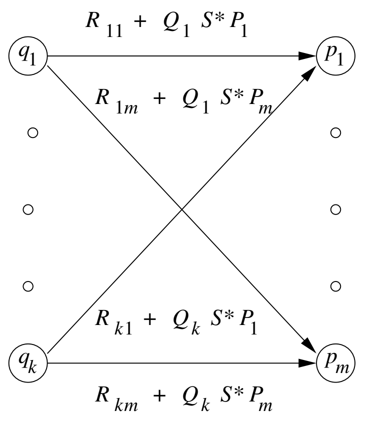

Figure 3.8: Result of eliminating state s from Fig. 3.7

in the accepting state, we can return to it as many times as we like, by following a path whose label is in $L(U)$.

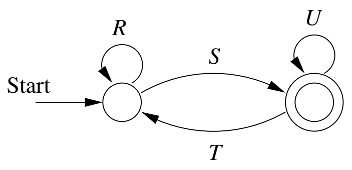

Figure 3.9: A generic two-state automaton

3. If the start state is also an accepting state, then we must also perform a state-elimination from the original automaton that gets rid of every state but the start state. When we do so, we are left with a one-state automaton that looks like Fig. 3.10. The regular expression denoting the strings that it accepts is R^*.

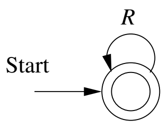

Figure 3.10: A generic one-state automaton

4. The desired regular expression is the sum (union) of all the expressions derived from the reduced automata for each accepting state, by rules (2) and (3).

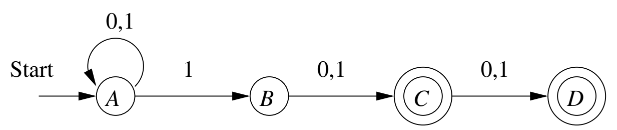

Figure 3.11: An NFA accepting strings that have a 1 either two or three positions from the end

Example 3.6: Let us consider the NFA in Fig. 3.11 that accepts all strings of 0's and 1's such that either the second or third position from the end has a 1. Our first step is to convert it to an automaton with regular expression labels. Since no state elimination has been performed, all we have to do is replace the labels "0,1" with the equivalent regular expression $\mathbf{0}+\mathbf{1}$. The result is shown in Fig. 3.12.

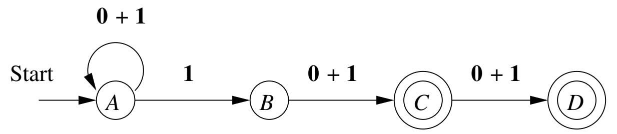

Figure 3.12: The automaton of Fig. 3.11 with regular-expression labels

Let us first eliminate state B. Since this state is neither accepting nor the start state, it will not be in any of the reduced automata. Thus, we save work if we eliminate it first, before developing the two reduced automata that correspond to the two accepting states.

State B has one predecessor, A, and one successor, C. In terms of the regular expressions in the diagram of Fig. 3.7: $Q_1 = \mathbf{1}$, $P_1 = \mathbf{0}+\mathbf{1}$, $R_{11} = \emptyset$ (since the arc from A to C does not exist), and $S = \emptyset$ (because there is no loop at state B). As a result, the expression on the new arc from A to C is $\emptyset + \mathbf{1}\emptyset^*(\mathbf{0}+\mathbf{1})$.

To simplify, we first eliminate the initial $\emptyset$, which may be ignored in a union. The expression thus becomes $\mathbf{1}\emptyset^*(\mathbf{0}+\mathbf{1})$. Note that the regular expression $\emptyset^*$ is equivalent to the regular expression ϵ, since

$$L(\emptyset^*) = \{\epsilon\} \cup L(\emptyset) \cup L(\emptyset)L(\emptyset) \cup \cdots$$

Since all the terms but the first are empty, we see that $L(\emptyset^*) = \{\epsilon\}$, which is the same as $L(\epsilon)$. Thus, $\mathbf{1}\emptyset^*(\mathbf{0}+\mathbf{1})$ is equivalent to $\mathbf{1}(\mathbf{0}+\mathbf{1})$, which is the expression we use for the arc $A \rightarrow C$ in Fig. 3.13.

Now, we must branch, eliminating states C and D in separate reductions. To eliminate state C, the mechanics are similar to those we performed above to eliminate state B, and the resulting automaton is shown in Fig. 3.14.

In terms of the generic two-state automaton of Fig. 3.9, the regular expressions from Fig. 3.14 are: $R = \mathbf{0}+\mathbf{1}$, $S = \mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$, $T = \emptyset$, and $U = \emptyset$. The expression U^* can be replaced by ϵ, i.e., eliminated in a concatenation;

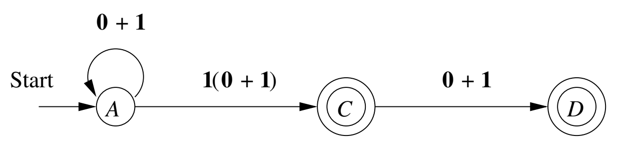

Figure 3.13: Eliminating state B

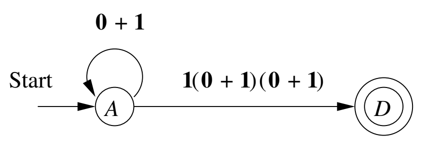

Figure 3.14: A two-state automaton with states A and D

the justification is that $\emptyset^* = \epsilon$, as we discussed above. Also, the expression SU^*T is equivalent to $\emptyset$, since T, one of the terms of the concatenation, is $\emptyset$. The generic expression $(R + SU^*T)^*SU^*$ thus simplifies in this case to R^*S, or $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$. In informal terms, the language of this expression is any string ending in 1, followed by two symbols that are each either 0 or 1. That language is one portion of the strings accepted by the automaton of Fig. 3.11: those strings whose third position from the end has a 1.

Now, we must start again at Fig. 3.13 and eliminate state D instead of C. Since D has no successors, an inspection of Fig. 3.7 tells us that there will be no changes to arcs, and the arc from C to D is eliminated, along with state D. The resulting two-state automaton is shown in Fig. 3.15.

This automaton is very much like that of Fig. 3.14; only the label on the arc from the start state to the accepting state is different. Thus, we can apply the rule for two-state automata and simplify the expression to get $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$. This expression represents the other type of string the automaton accepts: those with a 1 in the second position from the end.

All that remains is to sum the two expressions to get the expression for the entire automaton of Fig. 3.11. This expression is

$$(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1}) + (\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$$

□

3.2.3 Converting Regular Expressions to Automata

We shall now complete the plan of Fig. 3.1 by showing that every language L that is $L(R)$ for some regular expression R, is also $L(E)$ for some ϵ-NFA E. The proof is a structural induction on the expression R. We start by showing how to construct automata for the basis expressions: single symbols, ϵ, and $\emptyset$. We then show how to combine these automata into larger automata that accept the union, concatenation, or closure of the language accepted by smaller automata.

All of the automata we construct are ϵ-NFA's with a single accepting state.

Ordering the Elimination of States

As we observed in Example 3.6, when a state is neither the start state nor an accepting state, it gets eliminated in all the derived automata. Thus, one of the advantages of the state-elimination process compared with the mechanical generation of regular expressions that we described in Section 3.2.1 is that we can start by eliminating all the states that are neither start nor accepting, once and for all. We only have to begin duplicating the reduction effort when we need to eliminate some accepting states.

Even there, we can combine some of the effort. For instance, if there are three accepting states p, q, and r, we can eliminate p and then branch to eliminate either q or r, thus producing the automata for accepting states r and q, respectively. We then start again with all three accepting states and eliminate both q and r to get the automaton for p.

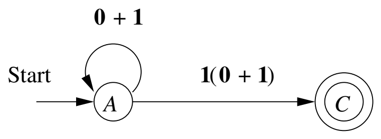

Figure 3.15: Two-state automaton resulting from the elimination of D

Theorem 3.7: Every language defined by a regular expression is also defined by a finite automaton.

PROOF: Suppose $L = L(R)$ for a regular expression R. We show that $L = L(E)$ for some ϵ-NFA E with:

1. Exactly one accepting state.
2. No arcs into the initial state.
3. No arcs out of the accepting state.

The proof is by structural induction on R, following the recursive definition of regular expressions that we had in Section 3.1.2.

BASIS: There are three parts to the basis, shown in Fig. 3.16. In part (a) we see how to handle the expression ϵ. The language of the automaton is easily seen to be $\{\epsilon\}$, since the only path from the start state to an accepting state is labeled ϵ. Part (b) shows the construction for $\emptyset$. Clearly there are no paths from start state to accepting state, so $\emptyset$ is the language of this automaton. Finally, part (c) gives the automaton for a regular expression $\mathbf{a}$. The language of this automaton evidently consists of the one string a, which is also $L(\mathbf{a})$. It

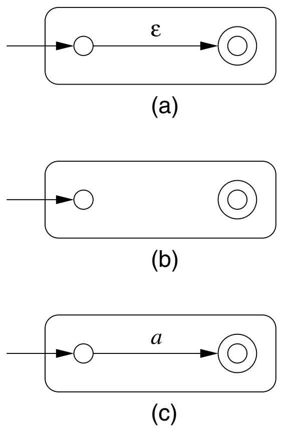

Figure 3.16: The basis of the construction of an automaton from a regular expression

is easy to check that these automata all satisfy conditions (1), (2), and (3) of the inductive hypothesis.

INDUCTION: The three parts of the induction are shown in Fig. 3.17. We assume that the statement of the theorem is true for the immediate subexpressions of a given regular expression; that is, the languages of these subexpressions are also the languages of ϵ-NFA's with a single accepting state. The four cases are:

1. The expression is $R + S$ for some smaller expressions R and S. Then the automaton of Fig. 3.17(a) serves. That is, starting at the new start state, we can go to the start state of either the automaton for R or the automaton for S. We then reach the accepting state of one of these automata, following a path labeled by some string in $L(R)$ or $L(S)$, respectively. Once we reach the accepting state of the automaton for R or S, we can follow one of the ϵ-arcs to the accepting state of the new automaton. Thus, the language of the automaton in Fig. 3.17(a) is $L(R) \cup L(S)$.

2. The expression is RS for some smaller expressions R and S. The automaton for the concatenation is shown in Fig. 3.17(b). Note that the start state of the first automaton becomes the start state of the whole, and the accepting state of the second automaton becomes the accepting state of the whole. The idea is that the only paths from start to accepting state go first through the automaton for R, where it must follow a path labeled by a string in $L(R)$, and then through the automaton for S, where it follows a path labeled by a string in $L(S)$. Thus, the paths in the automaton of Fig. 3.17(b) are all and only those labeled by strings in $L(R)L(S)$.

3. The expression is R^* for some smaller expression R. Then we use the

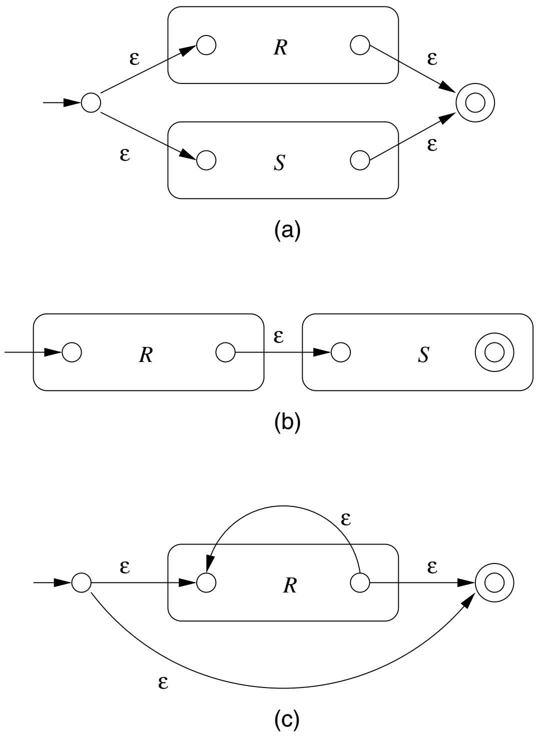

Figure 3.17: The inductive step in the regular-expression-to-ϵ-NFA construction

automaton of Fig. 3.17(c). That automaton allows us to go either:

(a) Directly from the start state to the accepting state along a path labeled ϵ. That path lets us accept ϵ, which is in $L(R^*)$ no matter what expression R is.

(b) To the start state of the automaton for R, through that automaton one or more times, and then to the accepting state. This set of paths allows us to accept strings in $L(R)$, $L(R)L(R)$, $L(R)L(R)L(R)$, and so on, thus covering all strings in $L(R^*)$ except perhaps ϵ, which was covered by the direct arc to the accepting state mentioned in (3a).

4. The expression is (R) for some smaller expression R. The automaton for R also serves as the automaton for (R), since the parentheses do not change the language defined by the expression.

It is a simple observation that the constructed automata satisfy the three conditions given in the inductive hypothesis — one accepting state, with no arcs into the initial state or out of the accepting state. $\square$

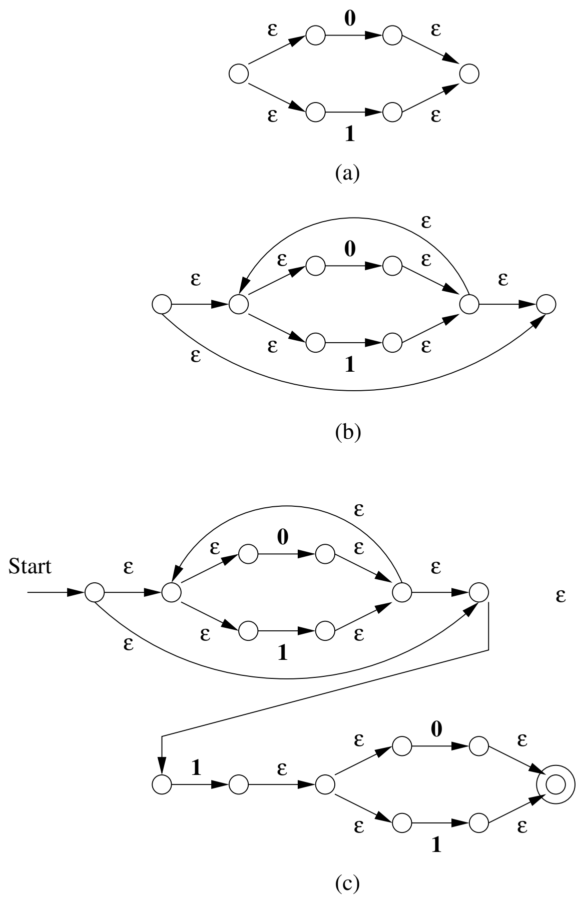

Figure 3.18: Automata constructed for Example 3.8

Example 3.8: Let us convert the regular expression $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$ to an ϵ-NFA. Our first step is to construct an automaton for $\mathbf{0}+\mathbf{1}$. We use two automata constructed according to Fig. 3.16(c), one with label $\mathbf{0}$ on the arc and one with label $\mathbf{1}$. These two automata are then combined using the union construction of Fig. 3.17(a). The result is shown in Fig. 3.18(a).

Next, we apply to Fig. 3.18(a) the star construction of Fig. 3.17(c). This automaton is shown in Fig. 3.18(b). The last two steps involve applying the concatenation construction of Fig. 3.17(b). First, we connect the automaton of Fig. 3.18(b) to another automaton designed to accept only the string 1. This automaton is another application of the basis construction of Fig. 3.16(c) with label $\mathbf{1}$ on the arc. Note that we must create a *new* automaton to recognize 1; we must not use the automaton for 1 that was part of Fig. 3.18(a). The third automaton in the concatenation is another automaton for $\mathbf{0}+\mathbf{1}$. Again, we must create a copy of the automaton of Fig. 3.18(a); we must not use the same copy that became part of Fig. 3.18(b). The complete automaton is shown in Fig. 3.18(c). Note that this ϵ-NFA, when ϵ-transitions are removed, looks just like the much simpler automaton of Fig. 3.15 that also accepts the strings that have a 1 in their next-to-last position. □

3.2.4 Exercises for Section 3.2

Exercise 3.2.1: Here is a transition table for a DFA:

	0	1
$\rightarrow q_1$	q_2	q_1
q_2	q_3	q_1
$*q_3$	q_3	q_2

* a) Give all the regular expressions $R_{ij}^{(0)}$. *Note*: Think of state q_i as if it were the state with integer number i.

* b) Give all the regular expressions $R_{ij}^{(1)}$. Try to simplify the expressions as much as possible.

c) Give all the regular expressions $R_{ij}^{(2)}$. Try to simplify the expressions as much as possible.

d) Give a regular expression for the language of the automaton.

* e) Construct the transition diagram for the DFA and give a regular expression for its language by eliminating state q_2.

Exercise 3.2.2: Repeat Exercise 3.2.1 for the following DFA:

	0	1
$\rightarrow q_1$	q_2	q_3
q_2	q_1	q_3
$*q_3$	q_2	q_1

Note that solutions to parts (a), (b) and (e) are *not* available for this exercise.

Exercise 3.2.3: Convert the following DFA to a regular expression, using the state-elimination technique of Section 3.2.2.

	0	1
$\rightarrow *p$	s	p
q	p	s
r	r	q
s	q	r

Exercise 3.2.4: Convert the following regular expressions to NFA's with ϵ-transitions.

* a) $\mathbf{01}^*$.

b) $(\mathbf{0}+\mathbf{1})\mathbf{01}$.

c) $\mathbf{00}(\mathbf{0}+\mathbf{1})^*$.

Exercise 3.2.5: Eliminate ϵ-transitions from your ϵ-NFA's of Exercise 3.2.4. A solution to part (a) appears in the book's Web pages.

! **Exercise 3.2.6:** Let $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ be an ϵ-NFA such that there are no transitions into q_0 and no transitions out of q_f. Describe the language accepted by each of the following modifications of A, in terms of $L = L(A)$:

* a) The automaton constructed from A by adding an ϵ-transition from q_f to q_0.

* b) The automaton constructed from A by adding an ϵ-transition from q_0 to every state reachable from q_0 (along a path whose labels may include symbols of Σ as well as ϵ).

c) The automaton constructed from A by adding an ϵ-transition to q_f from every state that can reach q_f along some path.

d) The automaton constructed from A by doing both (b) and (c).

!! **Exercise 3.2.7:** There are some simplifications to the constructions of Theorem 3.7, where we converted a regular expression to an ϵ-NFA. Here are three:

1. For the union operator, instead of creating new start and accepting states, merge the two start states into one state with all the transitions of both start states. Likewise, merge the two accepting states, having all transitions to either go to the merged state instead.

2. For the concatenation operator, merge the accepting state of the first automaton with the start state of the second.

3. For the closure operator, simply add ϵ-transitions from the accepting state to the start state and vice-versa.

Each of these simplifications, by themselves, still yield a correct construction; that is, the resulting ϵ-NFA for any regular expression accepts the language of the expression. Which subsets of changes (1), (2), and (3) may be made to the construction together, while still yielding a correct automaton for every regular expression?

*!! **Exercise 3.2.8:** Give an algorithm that takes a DFA A and computes the number of strings of length n (for some given n, not related to the number of states of A) accepted by A. Your algorithm should be polynomial in both n and the number of states of A. *Hint*: Use the technique suggested by the construction of Theorem 3.4.

3.3 Applications of Regular Expressions

A regular expression that gives a "picture" of the pattern we want to recognize is the medium of choice for applications that search for patterns in text. The regular expressions are then compiled, behind the scenes, into deterministic or nondeterministic automata, which are then simulated to produce a program that recognizes patterns in text. In this section, we shall consider two important classes of regular-expression-based applications: lexical analyzers and text search.

3.3.1 Regular Expressions in UNIX

Before seeing the applications, we shall introduce the UNIX notation for extended regular expressions. This notation gives us a number of additional capabilities. In fact, the UNIX extensions include certain features, especially the ability to name and refer to previous strings that have matched a pattern, that actually allow nonregular languages to be recognized. We shall not consider these features here; rather we shall only introduce the shorthands that allow complex regular expressions to be written succinctly.

The first enhancement to the regular-expression notation concerns the fact that most real applications deal with the ASCII character set. Our examples have typically used a small alphabet, such as $\{0,1\}$. The existence of only two symbols allowed us to write succinct expressions like $\mathbf{0}+\mathbf{1}$ for "any character." However, if there were 128 characters, say, the same expression would involve listing them all, and would be highly inconvenient to write. Thus, UNIX regular expressions allow us to write *character classes* to represent large sets of characters as succinctly as possible. The rules for character classes are:

- The symbol `.` (dot) stands for "any character."

- The sequence `[`$a_1a_2\cdots a_k$`]` stands for the regular expression

$$a_1 + a_2 + \cdots + a_k$$

  This notation saves about half the characters, since we don't have to write the $+$-signs. For example, we could express the four characters used in C comparison operators by `[<>=!]`.

- Between the square braces we can put a range of the form x-y to mean all the characters from x to y in the ASCII sequence. Since the digits have codes in order, as do the upper-case letters and the lower-case letters, we can express many of the classes of characters that we really care about with just a few keystrokes. For example, the digits can be expressed `[0-9]`, the upper-case letters can be expressed `[A-Z]`, and the set of all letters and digits can be expressed `[A-Za-z0-9]`. If we want to include a minus sign among a list of characters, we can place it first or last, so it is not confused with its use to form a character range. For example, the set of digits, plus the dot, plus, and minus signs that are used to form signed decimal numbers may be expressed `[-+.0-9]`. Square brackets, or other characters that have special meanings in UNIX regular expressions can be represented as characters by preceding them with a backslash (\).

- There are special notations for several of the most common classes of characters. For instance:

  a) `[:digit:]` is the set of ten digits, the same as `[0-9]`.[3]

  b) `[:alpha:]` stands for any alphabetic character, as does `[A-Za-z]`.

  c) `[:alnum:]` stands for the digits and letters (alphabetic and numeric characters), as does `[A-Za-z0-9]`.

In addition, there are several operators that are used in UNIX regular expressions that we have not encountered previously. None of these operators extend what languages can be expressed, but they sometimes make it easier to express what we want.

1. The operator $|$ is used in place of $+$ to denote union.

2. The operator `?` means "zero or one of." Thus, $R?$ in UNIX is the same as $\epsilon + R$ in this book's regular-expression notation.

3. The operator `+` means "one or more of." Thus, $R+$ in UNIX is shorthand for RR^* in our notation.

4. The operator $\{n\}$ means "n copies of." Thus, $R\{5\}$ in UNIX is shorthand for $RRRRR$.

Note that UNIX regular expressions allow parentheses to group subexpressions, just as for the regular expressions described in Section 3.1.2, and the same operator precedence is used (with `?`, `+` and $\{n\}$ treated like `*` as far as precedence is concerned). The star operator `*` is used in UNIX (without being a superscript, of course) with the same meaning as we have used.

3.3.2 Lexical Analysis

One of the oldest applications of regular expressions was in specifying the component of a compiler called a "lexical analyzer." This component scans the source program and recognizes all *tokens*, those substrings of consecutive characters that belong together logically. Keywords and identifiers are common examples of tokens, but there are many others.

[3]The notation `[:digit:]` has the advantage that should some code other than ASCII be used, including a code where the digits did not have consecutive codes, `[:digit:]` would still represent `[0123456789]`, while `[0-9]` would represent whatever characters had codes between the codes for 0 and 9, inclusive.

The Complete Story for UNIX Regular Expressions

The reader who wants to get the complete list of operators and shorthands available in the UNIX regular-expression notation can find them in the manual pages for various commands. There are some differences among the various versions of UNIX, but a command like `man grep` will get you the notation used for the `grep` command, which is fundamental. "Grep" stands for "Global (search for) Regular Expression and Print," incidentally.

The UNIX command `lex` and its GNU version `flex`, accept as input a list of regular expressions, in the UNIX style, each followed by a bracketed section of code that indicates what the lexical analyzer is to do when it finds an instance of that token. Such a facility is called a *lexical-analyzer generator*, because it takes as input a high-level description of a lexical analyzer and produces from it a function that is a working lexical analyzer.

Commands such as `lex` and `flex` have been found extremely useful because the regular-expression notation is exactly as powerful as we need to describe tokens. These commands are able to use the regular-expression-to-DFA conversion process to generate an efficient function that breaks source programs into tokens. They make the implementation of a lexical analyzer an afternoon's work, while before the development of these regular-expression-based tools, the hand-generation of the lexical analyzer could take months. Further, if we need to modify the lexical analyzer for any reason, it is often a simple matter to change a regular expression or two, instead of having to go into mysterious code to fix a bug.

Example 3.9 : In Fig. 3.19 is an example of partial input to the `lex` command, describing some of the tokens that are found in the language C. The first line handles the keyword `else` and the action is to return a symbolic constant (`ELSE` in this example) to the parser for further processing. The second line contains a regular expression describing identifiers: a letter followed by zero or more letters and/or digits. The action is first to enter that identifier in the symbol table if not already there; `lex` isolates the token found in a buffer, so this piece of code knows exactly what identifier was found. Finally, the lexical analyzer returns the symbolic constant `ID`, which has been chosen in this example to represent identifiers.

The third entry in Fig. 3.19 is for the sign `>=`, a two-character operator. The last example we show is for the sign `=`, a one-character operator. There would in practice appear expressions describing each of the keywords, each of the signs and punctuation symbols like commas and parentheses, and families of constants such as numbers and strings. Many of these are very simple, just a sequence of one or more specific characters. However, some have more

```
else                    {return(ELSE);}

[A-Za-z][A-Za-z0-9]*    {code to enter the found identifier
                         in the symbol table;
                         return(ID);
                        }

>=                      {return(GE);}

=                       {return(ASGN);}

...
```

Figure 3.19: A sample of `lex` input

of the flavor of identifiers, requiring the full power of the regular-expression notation to describe. The integers, floating-point numbers, character strings, and comments are other examples of sets of strings that profit from the regular-expression capabilities of commands like `lex`. □

The conversion of a collection of expressions, such as those suggested in Fig. 3.19, to an automaton proceeds approximately as we have described formally in the preceding sections. We start by building an automaton for the union of all the expressions. This automaton in principle tells us only that *some* token has been recognized. However, if we follow the construction of Theorem 3.7 for the union of expressions, the ϵ-NFA state tells us exactly which token has been recognized.

The only problem is that more than one token may be recognized at once; for instance, the string `else` matches not only the regular expression **else** but also the expression for identifiers. The standard resolution is for the lexical-analyzer generator to give priority to the first expression listed. Thus, if we want keywords like `else` to be *reserved* (not usable as identifiers), we simply list them ahead of the expression for identifiers.

3.3.3 Finding Patterns in Text

In Section 2.4.1 we introduced the notion that automata could be used to search efficiently for a set of words in a large repository such as the Web. While the tools and technology for doing so are not so well developed as that for lexical analyzers, the regular-expression notation is valuable for describing searches for interesting patterns. As for lexical analyzers, the capability to go from the natural, descriptive regular-expression notation to an efficient (automaton-based) implementation offers substantial intellectual leverage.

The general problem for which regular-expression technology has been found useful is the description of a vaguely defined class of patterns in text. The vagueness of the description virtually guarantees that we shall not describe the pattern correctly at first — perhaps we can never get exactly the right description. By using regular-expression notation, it becomes easy to describe the patterns at a high level, with little effort, and to modify the description quickly when things go wrong. A "compiler" for regular expressions is useful to turn the expressions we write into executable code.

Let us explore an extended example of the sort of problem that arises in many Web applications. Suppose that we want to scan a very large number of Web pages and detect addresses. We might simply want to create a mailing list. Or, perhaps we are trying to classify businesses by their location so that we can answer queries like "find me a restaurant within 10 minutes drive of where I am now."

We shall focus on recognizing street addresses in particular. What is a street address? We'll have to figure that out, and if, while testing the software, we find we miss some cases, we'll have to modify the expressions to capture what we were missing. To begin, a street address will probably end in "Street" or its abbreviation, "St." However, some people live on "Avenues" or "Roads," and these might be abbreviated in the address as well. Thus, we might use as the ending for our regular expression something like:

```
Street|St\.|Avenue|Ave\.|Road|Rd\.
```

In the above expression, we have used UNIX-style notation, with the vertical bar, rather than +, as the union operator. Note also that the dots are *escaped* with a preceding backslash, since dot has the special meaning of "any character" in UNIX expressions, and in this case we really want only the period or "dot" character to end the three abbreviations.

The designation such as `Street` must be preceded by the name of the street. Usually, the name is a capital letter followed by some lower-case letters. We can describe this pattern by the UNIX expression `[A-Z][a-z]*`. However, some streets have a name consisting of more than one word, such as Rhode Island Avenue in Washington DC. Thus, after discovering that we were missing addresses of this form, we could revise our description of street names to be

```
'[A-Z][a-z]*( [A-Z][a-z]*)*'
```

The expression above starts with a group consisting of a capital and zero or more lower-case letters. There follow zero or more groups consisting of a blank, another capital letter, and zero or more lower-case letters. The blank is an ordinary character in UNIX expressions, but to avoid having the above expression look like two expressions separated by a blank in a UNIX command line, we are required to place quotation marks around the whole expression. The quotes are not part of the expression itself.

Now, we need to include the house number as part of the address. Most house numbers are a string of digits. However, some will have a letter following, as in "123A Main St." Thus, the expression we use for numbers has an optional capital letter following: `[0-9]+[A-Z]?`. Notice that we use the UNIX + operator for "one or more" digits and the ? operator for "zero or one" capital letter. The entire expression we have developed for street addresses is:

```
'[0-9]+[A-Z]? [A-Z][a-z]*( [A-Z][a-z]*)*
(Street|St\.|Avenue|Ave\.|Road|Rd\.)'
```

If we work with this expression, we shall do fairly well. However, we shall eventually discover that we are missing:

1. Streets that are called something other than a street, avenue, or road. For example, we shall miss "Boulevard," "Place," "Way," and their abbreviations.

2. Street names that are numbers, or partially numbers, like "42nd Street."

3. Post-Office boxes and rural-delivery routes.

4. Street names that don't end in anything like "Street." An example is El Camino Real in Silicon Valley. Being Spanish for "the royal road," saying "El Camino Real Road" would be redundant, so one has to deal with complete addresses like "2000 El Camino Real."

5. All sorts of strange things we can't even imagine. Can you?

Thus, having a regular-expression compiler can make the process of slow convergence to the complete recognizer for addresses much easier than if we had to recode every change directly in a conventional programming language.

3.3.4 Exercises for Section 3.3

! **Exercise 3.3.1:** Give a regular expression to describe phone numbers in all the various forms you can think of. Consider international numbers as well as the fact that different countries have different numbers of digits in area codes and in local phone numbers.

!! **Exercise 3.3.2:** Give a regular expression to represent salaries as they might appear in employment advertising. Consider that salaries might be given on a per hour, week, month, or year basis. They may or may not appear with a dollar sign, or other unit such as "K" following. There may be a word or words nearby that identify a salary. Suggestion: look at classified ads in a newspaper, or on-line jobs listings to get an idea of what patterns might be useful.

! **Exercise 3.3.3:** At the end of Section 3.3.3 we gave some examples of improvements that could be possible for the regular expression that describes addresses. Modify the expression developed there to include all the mentioned options.

3.4 Algebraic Laws for Regular Expressions

In Example 3.5, we saw the need for simplifying regular expressions, in order to keep the size of expressions manageable. There, we gave some ad-hoc arguments why one expression could be replaced by another. In all cases, the basic issue was that the two expressions were *equivalent*, in the sense that they defined the same languages. In this section, we shall offer a collection of algebraic laws that bring to a higher level the issue of when two regular expressions are equivalent. Instead of examining specific regular expressions, we shall consider pairs of regular expressions with variables as arguments. Two expressions with variables are *equivalent* if whatever languages we substitute for the variables, the results of the two expressions are the same language.

An example of this process in the algebra of arithmetic is as follows. It is one matter to say that $1+2 = 2+1$. That is an example of the commutative law of addition, and it is easy to check by applying the addition operator on both sides and getting $3 = 3$. However, the *commutative law of addition* says more; it says that $x + y = y + x$, where x and y are variables that can be replaced by any two numbers. That is, no matter what two numbers we add, we get the same result regardless of the order in which we sum them.

Like arithmetic expressions, the regular expressions have a number of laws that work for them. Many of these are similar to the laws for arithmetic, if we think of union as addition and concatenation as multiplication. However, there are a few places where the analogy breaks down, and there are also some laws that apply to regular expressions but have no analog for arithmetic, especially when the closure operator is involved. The next sections form a catalog of the major laws. We conclude with a discussion of how one can check whether a proposed law for regular expressions is indeed a law; i.e., it will hold for any languages that we may substitute for the variables.

3.4.1 Associativity and Commutativity

Commutativity is the property of an operator that says we can switch the order of its operands and get the same result. An example for arithmetic was given above: $x + y = y + x$. *Associativity* is the property of an operator that allows us to regroup the operands when the operator is applied twice. For example, the associative law of multiplication is $(x \times y) \times z = x \times (y \times z)$. Here are three laws of these types that hold for regular expressions:

- $L + M = M + L$. This law, the *commutative law for union*, says that we may take the union of two languages in either order.

- $(L + M) + N = L + (M + N)$. This law, the *associative law for union*, says that we may take the union of three languages either by taking the union of the first two initially, or taking the union of the last two initially. Note that, together with the commutative law for union, we conclude that we can take the union of any collection of languages with any order and grouping, and the result will be the same. Intuitively, a string is in $L_1 \cup L_2 \cup \cdots \cup L_k$ if and only if it is in one or more of the L_i's.

- $(LM)N = L(MN)$. This law, the *associative law for concatenation*, says that we can concatenate three languages by concatenating either the first two or the last two initially.

Missing from this list is the "law" $LM = ML$, which would say that concatenation is commutative. However, this law is false.

Example 3.10: Consider the regular expressions **01** and **10**. These expressions denote the languages $\{01\}$ and $\{10\}$, respectively. Since the languages are different the general law $LM = ML$ cannot hold. If it did, we could substitute the regular expression **0** for L and **1** for M and conclude falsely that $\mathbf{01} = \mathbf{10}$. □

3.4.2 Identities and Annihilators

An *identity* for an operator is a value such that when the operator is applied to the identity and some other value, the result is the other value. For instance, 0 is the identity for addition, since $0 + x = x + 0 = x$, and 1 is the identity for multiplication, since $1 \times x = x \times 1 = x$. An *annihilator* for an operator is a value such that when the operator is applied to the annihilator and some other value, the result is the annihilator. For instance, 0 is an annihilator for multiplication, since $0 \times x = x \times 0 = 0$. There is no annihilator for addition.

There are three laws for regular expressions involving these concepts; we list them below.

- $\emptyset + L = L + \emptyset = L$. This law asserts that $\emptyset$ is the identity for union.
- $\epsilon L = L\epsilon = L$. This law asserts that ϵ is the identity for concatenation.
- $\emptyset L = L\emptyset = \emptyset$. This law asserts that $\emptyset$ is the annihilator for concatenation.

These laws are powerful tools in simplifications. For example, if we have a union of several expressions, some of which are, or have been simplified to $\emptyset$, then the $\emptyset$'s can be dropped from the union. Likewise, if we have a concatenation of several expressions, some of which are, or have been simplified to ϵ, we can drop the ϵ's from the concatenation. Finally, if we have a concatenation of any number of expressions, and even one of them is $\emptyset$, then the entire concatenation can be replaced by $\emptyset$.

3.4.3 Distributive Laws

A *distributive law* involves two operators, and asserts that one operator can be pushed down to be applied to each argument of the other operator individually. The most common example from arithmetic is the distributive law of multiplication over addition, that is, $x \times (y + z) = x \times y + x \times z$. Since multiplication is commutative, it doesn't matter whether the multiplication is on the left or right of the sum. However, there is an analogous law for regular expressions, that we must state in two forms, since concatenation is not commutative. These laws are:

- $L(M+N) = LM + LN$. This law, is the *left distributive law of concatenation over union.*
- $(M+N)L = ML + NL$. This law, is the *right distributive law of concatenation over union.*

Let us prove the left distributive law; the other is proved similarly. The proof will refer to languages only; it does not depend on the languages having regular expressions.

Theorem 3.11: If L, M, and N are any languages, then

$$L(M \cup N) = LM \cup LN$$

PROOF: The proof is similar to another proof about a distributive law that we saw in Theorem 1.10. We need first to show that a string w is in $L(M \cup N)$ if and only if it is in $LM \cup LN$.

(Only-if) If w is in $L(M \cup N)$, then $w = xy$, where x is in L and y is in either M or N. If y is in M, then xy is in LM, and therefore in $LM \cup LN$. Likewise, if y is in N, then xy is in LN and therefore in $LM \cup LN$.

(If) Suppose w is in $LM \cup LN$. Then w is in either LM or in LN. Suppose first that w is in LM. Then $w = xy$, where x is in L and y is in M. As y is in M, it is also in $M \cup N$. Thus, xy is in $L(M \cup N)$. If w is not in LM, then it is surely in LN, and a similar argument shows it is in $L(M \cup N)$. □

Example 3.12: Consider the regular expression $\mathbf{0}+\mathbf{01}^*$. We can "factor out a $\mathbf{0}$" from the union, but first we have to recognize that the expression $\mathbf{0}$ by itself is actually the concatenation of $\mathbf{0}$ with something, namely ϵ. That is, we use the identity law for concatenation to replace $\mathbf{0}$ by $\mathbf{0}\epsilon$, giving us the expression $\mathbf{0}\epsilon+\mathbf{01}^*$. Now, we can apply the left distributive law to replace this expression by $\mathbf{0}(\epsilon+\mathbf{1}^*)$. If we further recognize that ϵ is in $L(\mathbf{1}^*)$, then we observe that $\epsilon+\mathbf{1}^* = \mathbf{1}^*$, and can simplify to $\mathbf{01}^*$. □

3.4.4 The Idempotent Law

An operator is said to be *idempotent* if the result of applying it to two of the same values as arguments is that value. The common arithmetic operators are not idempotent; $x + x \neq x$ in general and $x \times x \neq x$ in general (although there are *some* values of x for which the equality holds, such as $0 + 0 = 0$). However, union and intersection are common examples of idempotent operators. Thus, for regular expressions, we may assert the following law:

- $L + L = L$. This law, the *idempotence law for union*, states that if we take the union of two identical expressions, we can replace them by one copy of the expression.

3.4.5 Laws Involving Closures

There are a number of laws involving the closure operators and its UNIX-style variants $^+$ and ?. We shall list them here, and give some explanation for why they are true.

- $(L^*)^* = L^*$. This law says that closing an expression that is already closed does not change the language. The language of $(L^*)^*$ is all strings created by concatenating strings in the language of L^*. But those strings are themselves composed of strings from L. Thus, the string in $(L^*)^*$ is also a concatenation of strings from L and is therefore in the language of L^*.

- $\emptyset^* = \epsilon$. The closure of $\emptyset$ contains only the string ϵ, as we discussed in Example 3.6.

- $\epsilon^* = \epsilon$. It is easy to check that the only string that can be formed by concatenating any number of copies of the empty string is the empty string itself.

- $L^+ = LL^* = L^*L$. Recall that L^+ is defined to be $L + LL + LLL + \cdots$. Also, $L^* = \epsilon + L + LL + LLL + \cdots$. Thus,

  $$LL^* = L\epsilon + LL + LLL + LLLL + \cdots$$

  When we remember that $L\epsilon = L$, we see that the infinite expansions for LL^* and for L^+ are the same. That proves $L^+ = LL^*$. The proof that $L^+ = L^*L$ is similar.[4]

- $L^* = L^+ + \epsilon$. The proof is easy, since the expansion of L^+ includes every term in the expansion of L^* except ϵ. Note that if the language L contains the string ϵ, then the additional "$+\epsilon$" term is not needed; that is, $L^+ = L^*$ in this special case.

- $L? = \epsilon + L$. This rule is really the definition of the ? operator.

3.4.6 Discovering Laws for Regular Expressions

Each of the laws above was proved, formally or informally. However, there is an infinite variety of laws about regular expressions that might be proposed. Is there a general methodology that will make our proofs of the correct laws

[4]Notice that, as a consequence, any language L commutes (under concatenation) with its own closure; $LL^* = L^*L$. That rule does not contradict the fact that, in general, concatenation is not commutative.

easy? It turns out that the truth of a law reduces to a question of the equality of two specific languages. Interestingly, the technique is closely tied to the regular-expression operators, and cannot be extended to expressions involving some other operators, such as intersection.

To see how this test works, let us consider a proposed law, such as

$$(L+M)^* = (L^*M^*)^*$$

This law says that if we have any two languages L and M, and we close their union, we get the same language as if we take the language L^*M^*, that is, all strings composed of zero or more choices from L followed by zero or more choices from M, and close that language.

To prove this law, suppose first that string w is in the language of $(L+M)^*$.[5] Then we can write $w = w_1w_2\cdots w_k$ for some k, where each w_i is in either L or M. It follows that each w_i is in the language of L^*M^*. To see why, if w_i is in L, pick one string, w_i, from L; this string is also in L^*. Pick no strings from M; that is, pick ϵ from M^*. If w_i is in M, the argument is similar. Once every w_i is seen to be in L^*M^*, it follows that w is in the closure of this language.

To complete the proof, we also have to prove the converse: that strings in $(L^*M^*)^*$ are also in $(L+M)^*$. We omit this part of the proof, since our objective is not to prove the law, but to notice the following important property of regular expressions.

Any regular expression with variables can be thought of as a *concrete* regular expression, one that has no variables, by thinking of each variable as if it were a distinct symbol. For example, the expression $(L+M)^*$ can have variables L and M replaced by symbols a and b, respectively, giving us the regular expression $(\mathbf{a}+\mathbf{b})^*$.

The language of the concrete expression guides us regarding the form of strings in any language that is formed from the original expression when we replace the variables by languages. Thus, in our analysis of $(L+M)^*$, we observed that any string w composed of a sequence of choices from either L or M, would be in the language of $(L+M)^*$. We can arrive at that conclusion by looking at the language of the concrete expression, $L\big((\mathbf{a}+\mathbf{b})^*\big)$, which is evidently the set of all strings of a's and b's. We could substitute any string in L for any occurrence of a in one of those strings, and we could substitute any string in M for any occurrence of b, with possibly different choices of strings for different occurrences of a or b. Those substitutions, applied to all the strings in $(\mathbf{a}+\mathbf{b})^*$, gives us all strings formed by concatenating strings from L and/or M, in any order.

The above statement may seem obvious, but as is pointed out in the box on "Extensions of the Test Beyond Regular Expressions May Fail," it is not even true when some other operators are added to the three regular-expression operators. We prove the general principle for regular expressions in the next theorem.

[5]For simplicity, we shall identify the regular expressions and their languages, and avoid saying "the language of" in front of every regular expression.

Theorem 3.13: Let E be a regular expression with variables $L_1, L_2, \ldots, L_m$. Form concrete regular expression C by replacing each occurrence of L_i by the symbol a_i, for $i = 1, 2, \ldots, m$. Then for any languages $L_1, L_2, \ldots, L_m$, every string w in $L(E)$ can be written $w = w_1 w_2 \cdots w_k$, where each w_i is in one of the languages, say L_{j_i}, and the string $a_{j_1} a_{j_2} \cdots a_{j_k}$ is in the language $L(C)$. Less formally, we can construct $L(E)$ by starting with each string in $L(C)$, say $a_{j_1} a_{j_2} \cdots a_{j_k}$, and substituting for each of the a_{j_i}'s any string from the corresponding language L_{j_i}.

PROOF: The proof is a structural induction on the expression E.

BASIS: The basis cases are where E is ϵ, $\emptyset$, or a variable L. In the first two cases, there is nothing to prove, since the concrete expression C is the same as E. If E is a variable L, then $L(E) = L$. The concrete expression C is just **a**, where a is the symbol corresponding to L. Thus, $L(C) = \{a\}$. If we substitute any string in L for the symbol a in this one string, we get the language L, which is also $L(E)$.

INDUCTION: There are three cases, depending on the final operator of E. First, suppose that $E = F + G$; i.e., a union is the final operator. Let C and D be the concrete expressions formed from F and G, respectively, by substituting concrete symbols for the language-variables in these expressions. Note that the same symbol must be substituted for all occurrences of the same variable, in both F and G. Then the concrete expression that we get from E is $C + D$, and $L(C + D) = L(C) + L(D)$.

Suppose that w is a string in $L(E)$, when the language variables of E are replaced by specific languages. Then w is in either $L(F)$ or $L(G)$. By the inductive hypothesis, w is obtained by starting with a concrete string in $L(C)$ or $L(D)$, respectively, and substituting for the symbols strings in the corresponding languages. Thus, in either case, the string w can be constructed by starting with a concrete string in $L(C+D)$, and making the same substitutions of strings for symbols.

We must also consider the cases where E is FG or F^*. However, the arguments are similar to the union case above, and we leave them for you to complete. $\Box$

3.4.7 The Test for a Regular-Expression Algebraic Law

Now, we can state and prove the test for whether or not a law of regular expressions is true. The test for whether $E = F$ is true, where E and F are two regular expressions with the same set of variables, is:

1. Convert E and F to concrete regular expressions C and D, respectively, by replacing each variable by a concrete symbol.

2. Test whether $L(C) = L(D)$. If so, then $E = F$ is a true law, and if not, then the "law" is false. Note that we shall not see the test for whether two

regular expressions denote the same language until Section 4.4. However, we can use ad-hoc means to decide the equality of the pairs of languages that we actually care about. Recall that if the languages are *not* the same, then it is sufficient to provide one counterexample: a single string that is in one language but not the other.

Theorem 3.14: The above test correctly identifies the true laws for regular expressions.

PROOF: We shall show that $L(E) = L(F)$ for any languages in place of the variables of E and F if and only if $L(C) = L(D)$.

(Only-if) Suppose $L(E) = L(F)$ for all choices of languages for the variables. In particular, choose for every variable L the concrete symbol a that replaces L in expressions C and D. Then for this choice, $L(C) = L(E)$, and $L(D) = L(F)$. Since $L(E) = L(F)$ is given, it follows that $L(C) = L(D)$.

(If) Suppose $L(C) = L(D)$. By Theorem 3.13, $L(E)$ and $L(F)$ are each constructed by replacing the concrete symbols of strings in $L(C)$ and $L(D)$, respectively, by strings in the languages that correspond to those symbols. If the strings of $L(C)$ and $L(D)$ are the same, then the two languages constructed in this manner will also be the same; that is, $L(E) = L(F)$. □

Example 3.15: Consider the prospective law $(L + M)^* = (L^*M^*)^*$. If we replace variables L and M by concrete symbols a and b respectively, we get the regular expressions $(\mathbf{a} + \mathbf{b})^*$ and $(\mathbf{a}^*\mathbf{b}^*)^*$. It is easy to check that both these expressions denote the language with all strings of a's and b's. Thus, the two concrete expressions denote the same language, and the law holds.

For another example of a law, consider $L^* = L^*L^*$. The concrete languages are $\mathbf{a}^*$ and $\mathbf{a}^*\mathbf{a}^*$, respectively, and each of these is the set of all strings of a's. Again, the law is found to hold; that is, concatenation of a closed language with itself yields that language.

Finally, consider the prospective law $L + ML = (L + M)L$. If we choose symbols a and b for variables L and M, respectively, we have the two concrete regular expressions $\mathbf{a} + \mathbf{ba}$ and $(\mathbf{a} + \mathbf{b})\mathbf{a}$. However, the languages of these expressions are not the same. For example, the string aa is in the second, but not the first. Thus, the prospective law is false. □

3.4.8 Exercises for Section 3.4

Exercise 3.4.1: Verify the following identities involving regular expressions.

* a) $R + S = S + R$.

b) $(R + S) + T = R + (S + T)$.

c) $(RS)T = R(ST)$.

Extensions of the Test Beyond Regular Expressions May Fail

Let us consider an extended regular-expression algebra that includes the intersection operator. Interestingly, adding $\cap$ to the three regular-expression operators does not increase the set of languages we can describe, as we shall see in Theorem 4.8. However, it does make the test for algebraic laws invalid.

Consider the "law" $L \cap M \cap N = L \cap M$; that is, the intersection of any three languages is the same as the intersection of the first two of these languages. This "law" is patently false. For example, let $L = M = \{a\}$ and $N = \emptyset$. But the test based on concretizing the variables would fail to see the difference. That is, if we replaced L, M, and N by the symbols a, b, and c, respectively, we would test whether $\{a\} \cap \{b\} \cap \{c\} = \{a\} \cap \{b\}$. Since both sides are the empty set, the equality of languages holds and the test would imply that the "law" is true.

d) $R(S + T) = RS + RT$.

e) $(R + S)T = RT + ST$.

* f) $(R^*)^* = R^*$.

g) $(\epsilon + R)^* = R^*$.

h) $(R^*S^*)^* = (R + S)^*$.

! **Exercise 3.4.2:** Prove or disprove each of the following statements about regular expressions.

* a) $(R + S)^* = R^* + S^*$.

b) $(RS + R)^*R = R(SR + R)^*$.

* c) $(RS + R)^*RS = (RR^*S)^*$.

d) $(R + S)^*S = (R^*S)^*$.

e) $S(RS + S)^*R = RR^*S(RR^*S)^*$.

Exercise 3.4.3: In Example 3.6, we developed the regular expression

$$(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1}) + (\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$$

Use the distributive laws to develop two different, simpler, equivalent expressions.

Exercise 3.4.4: At the beginning of Section 3.4.6, we gave part of a proof that $(L^*M^*)^* = (L+M)^*$. Complete the proof by showing that strings in $(L^*M^*)^*$ are also in $(L+M)^*$.

! **Exercise 3.4.5:** Complete the proof of Theorem 3.13 by handling the cases where regular expression E is of the form FG or of the form F^*.

3.5 Summary of Chapter 3

✦ *Regular Expressions*: This algebraic notation describes exactly the same languages as finite automata: the regular languages. The regular-expression operators are union, concatenation (or "dot"), and closure (or "star").

✦ *Regular Expressions in Practice*: Systems such as UNIX and various of its commands use an extended regular-expression language that provides shorthands for many common expressions. Character classes allow the easy expression of sets of symbols, while operators such as one-or-more-of and at-most-one-of augment the usual regular-expression operators.

✦ *Equivalence of Regular Expressions and Finite Automata*: We can convert a DFA to a regular expression by an inductive construction in which expressions for the labels of paths allowed to pass through increasingly larger sets of states are constructed. Alternatively, we can use a state-elimination procedure to build the regular expression for a DFA. In the other direction, we can construct recursively an ϵ-NFA from regular expressions, and then convert the ϵ-NFA to a DFA, if we wish.

✦ *The Algebra of Regular Expressions*: Regular expressions obey many of the algebraic laws of arithmetic, although there are differences. Union and concatenation are associative, but only union is commutative. Concatenation distributes over union. Union is idempotent.

✦ *Testing Algebraic Identities*: We can tell whether a regular-expression equivalence involving variables as arguments is true by replacing the variables by distinct constants and testing whether the resulting languages are the same.

3.6 Gradiance Problems for Chapter 3

The following is a sample of problems that are available on-line through the Gradiance system at `www.gradiance.com/pearson`. Each of these problems is worked like conventional homework. The Gradiance system gives you four choices that sample your knowledge of the solution. If you make the wrong choice, you are given a hint or advice and encouraged to try the same problem again.

Problem 3.1: Here is a finite automaton [shown on-line by the Gradiance system]. Which of the following regular expressions defines the same language as the finite automaton? Hint: each of the correct choices uses component expressions. Some of these components are:

1. The ways to get from A to D without going through D.
2. The ways to get from D to itself, without going through D.
3. The ways to get from A to itself, without going through A.

It helps to write down these expressions first, and then look for an expression that defines all the paths from A to D.

Problem 3.2: When we convert an automaton to a regular expression, we need to build expressions for the labels along paths from one state to another state that do not go through certain other states. Below is a nondeterministic finite automaton with three states [shown on-line by the Gradiance system]. For each of the six orders of the three states, find regular expressions that give the set of labels along all paths from the first state to the second state that never go through the third state. Then identify one of these expressions from the list of choices below.

Problem 3.3: Identify from the list below the regular expression that generates all and only the strings over alphabet $\{0, 1\}$ that end in 1.

Problem 3.4: Apply the construction in Fig. 3.16 and Fig. 3.17 to convert the regular expression $(0+1)^*(0+\epsilon)$ to an epsilon-NFA. Then, identify the true statement about your epsilon-NFA from the list below.

Problem 3.5: Consider the following identities for regular expressions; some are false and some are true. You are asked to decide which and in case it is false to provide the correct counterexample.

a) $R(S+T) = RS+RT$

b) $(R^*)^* = R^*$

c) $(R^*S^*)^* = (R+S)^*$

d) $(R+S)^* = R^* + S^*$

e) $S(RS+S)^*R = RR^*S(RR^*S)^*$

f) $(RS+R)^*R = R(SR+R)^*$

Problem 3.6: In this question you are asked to consider the truth or falsehood of six equivalences for regular expressions. If the equivalence is true, you must also identify the law from which it follows. In each case the statement $R = S$ is conventional shorthand for "$L(R) = L(S)$." The six proposed equivalences are:

1. $0^*1^* = 1^*0^*$
2. $01\emptyset = \emptyset$
3. $\epsilon 01 = 01$
4. $(0^* + 1^*)0 = 0^*0 + 1^*0$
5. $(0^*1)0^* = 0^*(10^*)$
6. $01 + 01 = 01$

Identify the correct statement from the list below.

Problem 3.7: Which of the following strings is **not** in the Kleene closure of the language $\{011, 10, 110\}$?

Problem 3.8: Here are seven regular expressions [shown on-line by the Gradiance system]. Determine the language of each of these expressions. Then, find in the list below a pair of equivalent expressions.

Problem 3.9: Converting a DFA such as the following [shown on-line by the Gradiance system]. to a regular expression requires us to develop regular expressions for limited sets of paths — those that take the automaton from one particular state to another particular state, without passing through some set of states. For the automaton above, determine the languages for the following limitations:

1. L_{AA} = the set of path labels that go from A to A without passing through C or D.
2. L_{AB} = the set of path labels that go from A to B without passing through C or D.
3. L_{BA} = the set of path labels that go from B to A without passing through C or D.
4. L_{BB} = the set of path labels that go from B to B without passing through C or D.

Then, identify a correct regular expression from the list below.

3.7 References for Chapter 3

The idea of regular expressions and the proof of their equivalence to finite automata is the work of S. C. Kleene [3]. However, the construction of an ϵ-NFA from a regular expression, as presented here, is the "McNaughton-Yamada construction," from [4]. The test for regular-expression identities by treating variables as constants was written down by J. Gischer [2]. Although thought to be folklore, this report demonstrated how adding several other operations such as intersection or shuffle (See Exercise 7.3.4) makes the test fail, even though they do not extend the class of languages representable.

Even before developing UNIX, K. Thompson was investigating the use of regular expressions in commands such as `grep`, and his algorithm for processing such commands appears in [5]. The early development of UNIX produced several other commands that make heavy use of the extended regular-expression notation, such as M. Lesk's `lex` command. A description of this command and other regular-expression techniques can be found in [1].

1. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986.

2. J. L. Gischer, STAN-CS-TR-84-1033 (1984).

3. S. C. Kleene, "Representation of events in nerve nets and finite automata," In C. E. Shannon and J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, pp. 3–42.

4. R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IEEE Trans. Electronic Computers* **9**:1 (Jan., 1960), pp. 39–47.

5. K. Thompson, "Regular expression search algorithm," *Comm. ACM* **11**:6 (June, 1968), pp. 419–422.