MODULE 2

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 $\mathbf{01^* + 10^*}$ 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 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. □

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, **0.1** is a regular expression meaning the same as **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 $0101\cdots01$.

The basis rule for regular expressions tells us that **0** and **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 **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 **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. □

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, **012** is grouped **(01)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 $\mathbf{01^* + 1}$ is grouped $\mathbf{(0(1^*)) + 1}$. The star operator is grouped first. Since the symbol **1** immediately to its left is a legal regular expression, that alone is the operand of the star. Next, we group the concatenation between **0** and $\mathbf{(1^*)}$, giving us the expression $\mathbf{(0(1^*))}$. Finally, the union operator connects the latter expression and the expression to its right, which is **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 $\mathbf{(01)^* + 1}$. 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 $\mathbf{0(1^* + 1)}$. 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) $(1+\epsilon)(00^*1)^*0^*$.

b) $(0^*1^*)^*000(0+1)^*$.

c) $(0+10)^*1^*$.

*! **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.

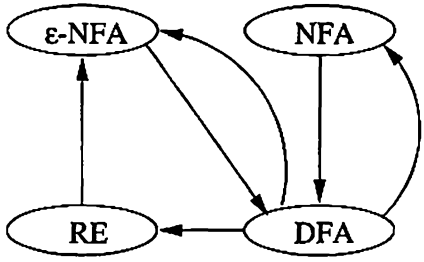


Figure 3.1: Plan for showing the equivalence of four different notations for regular languages

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.

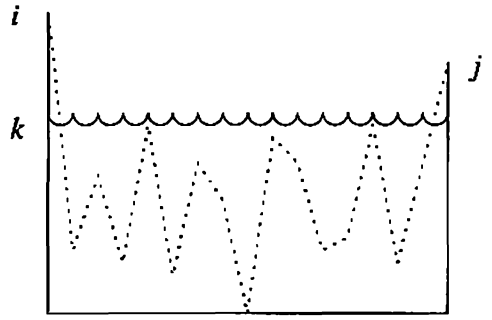


Figure 3.2: A path whose label is in the language of regular expression $R_{ij}^{(k)}$

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 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.

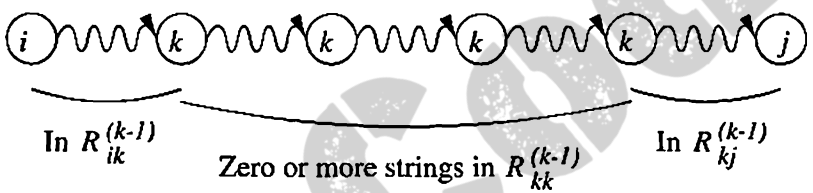


Figure 3.3: A path from i to j can be broken into segments at each point where it goes through state k

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. □

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.

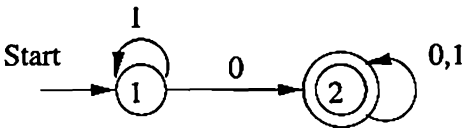


Figure 3.4: A DFA accepting all strings that have at least one 0

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

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

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 $(\epsilon + 1)^* = 1^*$; notice that both expressions denote any number of 1's. Further, $(\epsilon+1)1^* = 1^*$. Again, it can be observed that both expressions denote "any number of 1's." Thus, the original expression $R_{12}^{(1)}$ is equivalent to $0 + 1^*0$. 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 1^*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 $\emptyset(\epsilon + 1)^*(\epsilon + 1)$ 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
$R_{11}^{(2)}$	$1^* + 1^*0(\epsilon + 0 + 1)^*\emptyset$	1^*
$R_{12}^{(2)}$	$1^*0 + 1^*0(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$	$1^*0(0 + 1)^*$
$R_{21}^{(2)}$	$\emptyset + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*\emptyset$	$\emptyset$
$R_{22}^{(2)}$	$\epsilon + 0 + 1 + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$	$(0 + 1)^*$

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 $1^*0(0+1)^*$. 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. □

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, all of the expressions with superscript $(k+1)$ in the construction of Theorem 3.4 use the same subexpression $(R_{kk}^{(k)})^*$; 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,

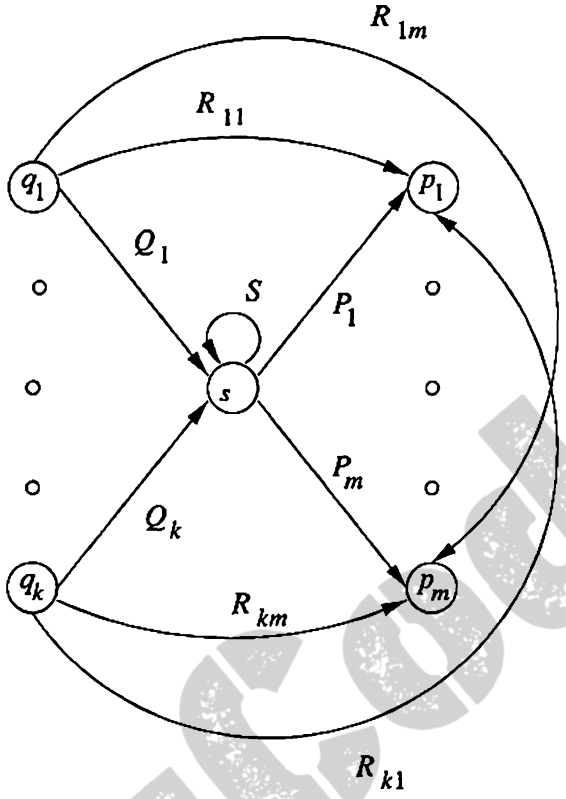


Figure 3.7: A state s about to be eliminated

we show a regular expression P_j 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 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

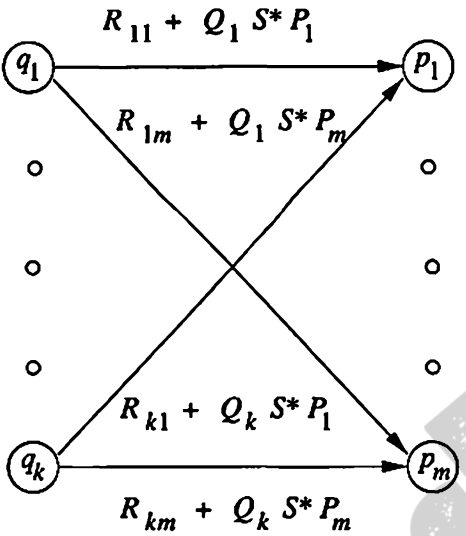


Figure 3.8: Result of eliminating state s from Fig. 3.7

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

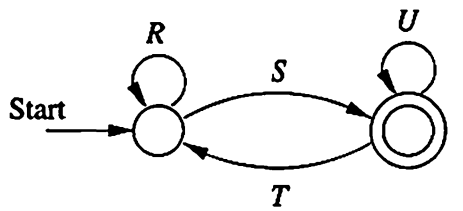


Figure 3.9: A generic two-state automaton

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^*.

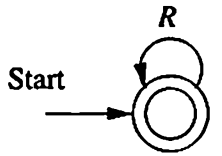


Figure 3.10: A generic one-state automaton

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).

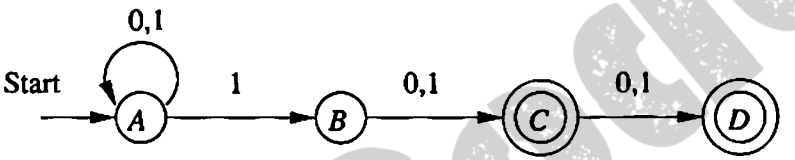


Figure 3.11: An NFA accepting strings that have a 1 either two or three positions from the end

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 $\mathbf{0+1}$. The result is shown in Fig. 3.12.

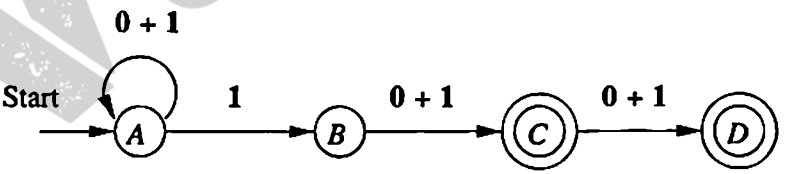


Figure 3.12: The automaton of Fig. 3.11 with regular-expression labels

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: $Q_1 = \mathbf{1}$, $P_1 = \mathbf{0+1}$, $R_{11} = \emptyset$ (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 $\emptyset + 1\emptyset^*(0+1)$.

To simplify, we first eliminate the initial $\emptyset$, which may be ignored in a union. The expression thus becomes $1\emptyset^*(0+1)$. 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, $1\emptyset^*(0+1)$ is equivalent to $1(0+1)$, which is the expression we use for the arc $A \to C$ in Fig. 3.13.

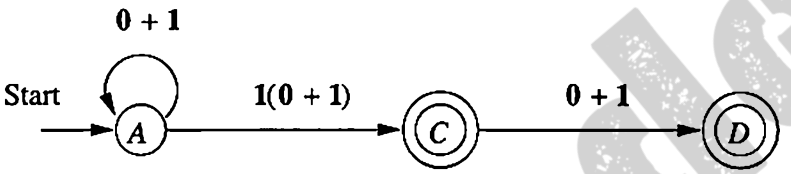


Figure 3.13: Eliminating state B

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.

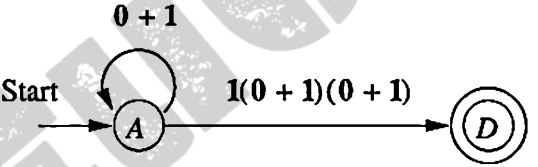


Figure 3.14: A two-state automaton with states A and D

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

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.

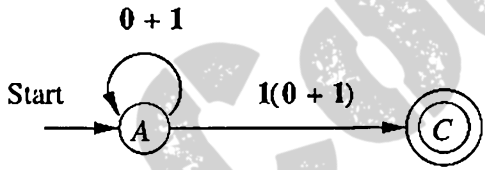


Figure 3.15: Two-state automaton resulting from the elimination of D

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

$$(0+1)^*1(0+1) + (0+1)^*1(0+1)(0+1)$$

□

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.

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.

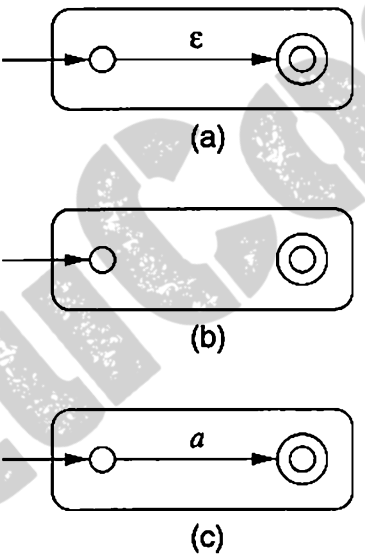


Figure 3.16: The basis of the construction of an automaton from a regular expression

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 **a**. The language of this automaton evidently consists of the one string a, which is also $L(\mathbf{a})$. It is easy to check that these automata all satisfy conditions (1), (2), and (3) of the inductive hypothesis.

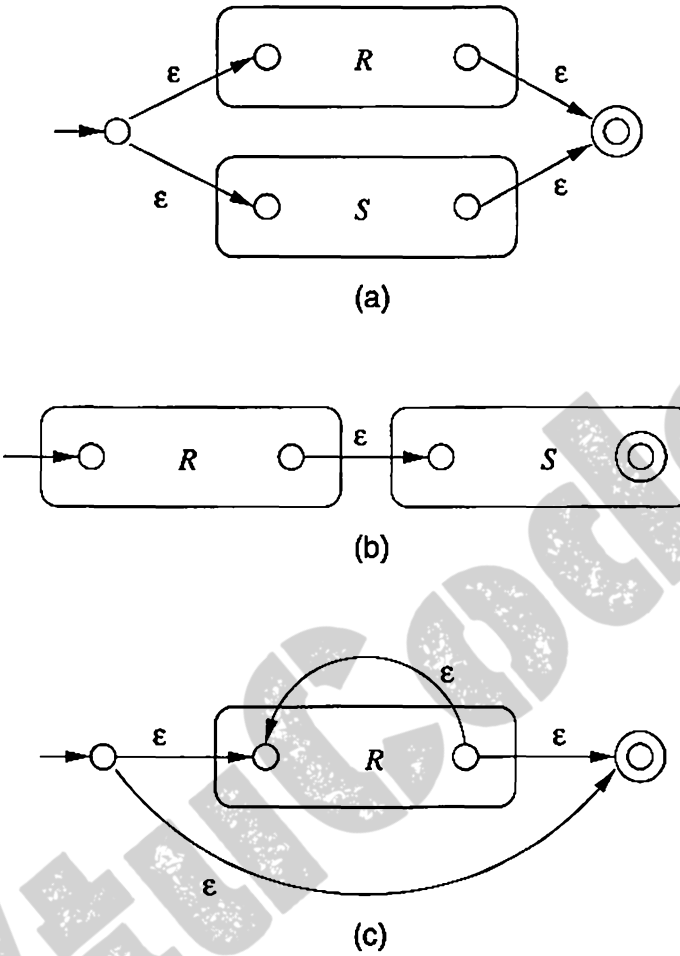


Figure 3.17: The inductive step in the regular-expression-to-ϵ-NFA construction

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 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. □

Example 3.8: Let us convert the regular expression $(0 + 1)^*1(0 + 1)$ to an ϵ-NFA. Our first step is to construct an automaton for $0 + 1$. We use two automata constructed according to Fig. 3.16(c), one with label 0 on the arc and one with label 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 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 $0 + 1$. 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

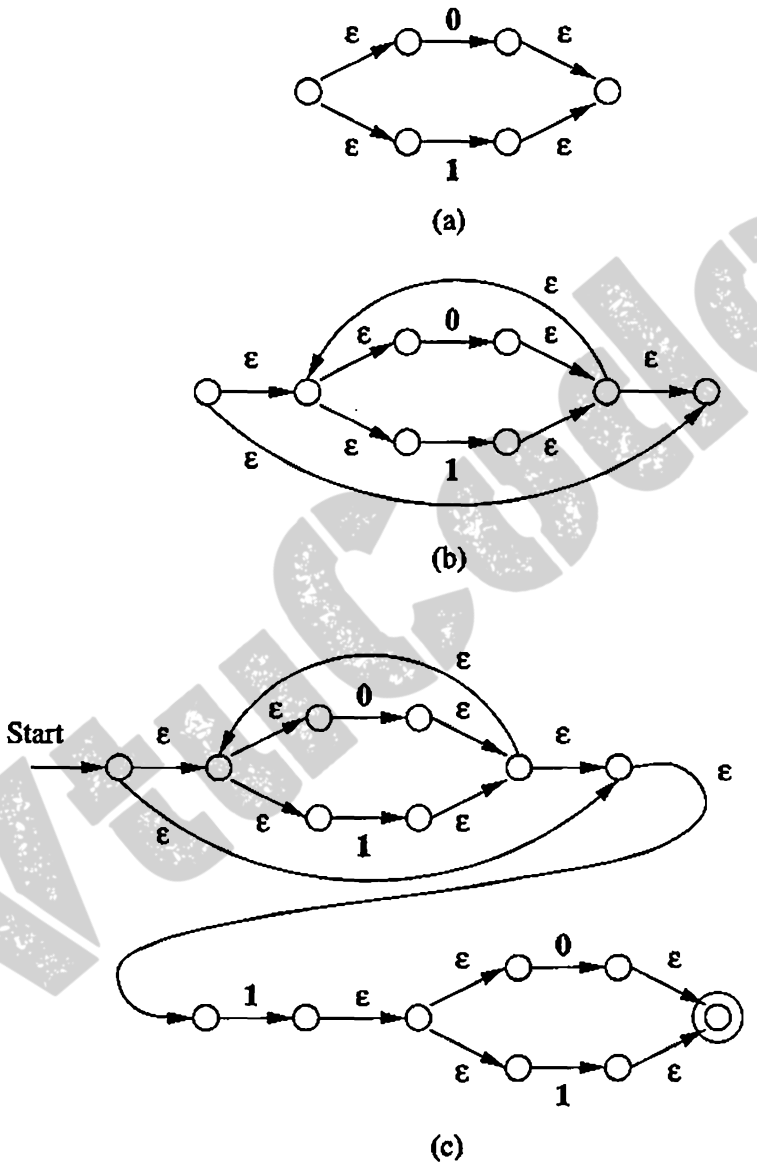


Figure 3.18: Automata constructed for Example 3.8

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) 01^*.

b) $(0+1)01$.

c) $00(0+1)^*$.

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 $0 + 1$ 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 $[a_1 a_2 \cdots a_k]$ 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(EQ);}

...
```

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.

4.1 Proving Languages not to be Regular

We have established that the class of languages known as the regular languages has at least four different descriptions. They are the languages accepted by DFA's, by NFA's, and by ϵ-NFA's; they are also the languages defined by regular expressions.

Not every language is a regular language. In this section, we shall introduce a powerful technique, known as the "pumping lemma," for showing certain languages not to be regular. We then give several examples of nonregular languages. In Section 4.2 we shall see how the pumping lemma can be used in tandem with closure properties of the regular languages to prove other languages not to be regular.

4.1.1 The Pumping Lemma for Regular Languages

Let us consider the language $L_{01} = \{0^n1^n \mid n \geq 1\}$. This language contains all strings 01, 0011, 000111, and so on, that consist of one or more 0's followed by an equal number of 1's. We claim that L_{01} is not a regular language. The intuitive argument is that if L_{01} were regular, then L_{01} would be the language of some DFA A. This automaton has some particular number of states, say k states. Imagine this automaton receiving k 0's as input. It is in some state after consuming each of the $k+1$ prefixes of the input: $\epsilon, 0, 00, \ldots, 0^k$. Since there are only k different states, the pigeonhole principle tells us that after reading two different prefixes, say 0^i and 0^j, A must be in the same state, say state q.

However, suppose instead that after reading i or j 0's, the automaton A starts receiving 1's as input. After receiving i 1's, it must accept if it previously received i 0's, but not if it received j 0's. Since it was in state q when the 1's started, it cannot "remember" whether it received i or j 0's, so we can "fool" A and make it do the wrong thing — accept if it should not, or fail to accept when it should.

The above argument is informal, but can be made precise. However, the same conclusion, that the language L_{01} is not regular, can be reached using a general result, as follows.

Theorem 4.1: (The *pumping lemma for regular languages*) Let L be a regular language. Then there exists a constant n (which depends on L) such that for every string w in L such that $|w| \geq n$, we can break w into three strings, $w = xyz$, such that:

1. $y \neq \epsilon$.
2. $|xy| \leq n$.
3. For all $k \geq 0$, the string xy^kz is also in L.

That is, we can always find a nonempty string y not too far from the beginning of w that can be "pumped"; that is, repeating y any number of times, or deleting it (the case $k = 0$), keeps the resulting string in the language L.

PROOF: Suppose L is regular. Then $L = L(A)$ for some DFA A. Suppose A has n states. Now, consider any string w of length n or more, say $w = a_1 a_2 \cdots a_m$, where $m \geq n$ and each a_i is an input symbol. For $i = 0, 1, \ldots, n$ define state p_i to be $\hat{\delta}(q_0, a_1 a_2 \cdots a_i)$, where δ is the transition function of A, and q_0 is the *start state of* A. That is, p_i is the state A is in after reading the first i symbols of w. Note that $p_0 = q_0$.

By the pigeonhole principle, it is not possible for the $n+1$ different p_i's for $i = 0, 1, \ldots, n$ to be distinct, since there are only n different states. Thus, we can find two different integers i and j, with $0 \leq i < j \leq n$, such that $p_i = p_j$. Now, we can break $w = xyz$ as follows:

1. $x = a_1 a_2 \cdots a_i$.
2. $y = a_{i+1} a_{i+2} \cdots a_j$.
3. $z = a_{j+1} a_{j+2} \cdots a_m$.

That is, x takes us to p_i once; y takes us from p_i back to p_i (since p_i is also p_j), and z is the balance of w. The relationships among the strings and states are suggested by Fig. 4.1. Note that x may be empty, in the case that $i = 0$. Also, z may be empty if $j = n = m$. However, y can not be empty, since i is strictly less than j.

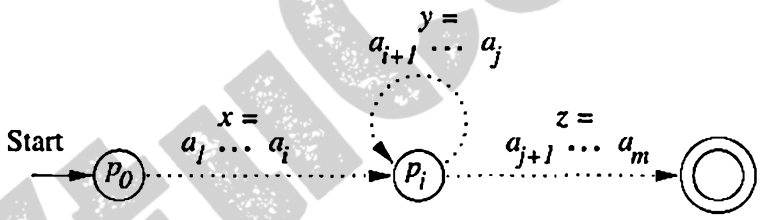


Figure 4.1: Every string longer than the number of states must cause a state to repeat

Now, consider what happens if the automaton A receives the input $xy^k z$ for any $k \geq 0$. If $k = 0$, then the automaton goes from the start state q_0 (which is also p_0) to p_i on input x. Since p_i is also p_j, it must be that A goes from p_i to the accepting state shown in Fig. 4.1 on input z. Thus, A accepts xz.

If $k > 0$, then A goes from q_0 to p_i on input x, circles from p_i to p_i k times on input y^k, and then goes to the accepting state on input z. Thus, for any $k \geq 0$, $xy^k z$ is also accepted by A; that is, $xy^k z$ is in L. □

4.1.2 Applications of the Pumping Lemma

Let us see some examples of how the pumping lemma is used. In each case, we shall propose a language and use the pumping lemma to prove that the language is not regular.

The Pumping Lemma as an Adversarial Game

Recall our discussion from Section 1.2.3 where we pointed out that a theorem whose statement involves several alternations of "for-all" and "there-exists" quantifiers can be thought of as a game between two players. The pumping lemma is an important example of this type of theorem, since it in effect involves four different quantifiers: "**for all** regular languages L **there exists** n such that **for all** w in L with $|w| \geq n$ **there exists** xyz equal to w such that $\cdots$." We can see the application of the pumping lemma as a game, in which:

1. Player 1 picks the language L to be proved nonregular.
2. Player 2 picks n, but doesn't reveal to player 1 what n is; player 1 must devise a play for all possible n's.
3. Player 1 picks w, which may depend on n and which must be of length at least n.
4. Player 2 divides w into x, y, and z, obeying the constraints that are stipulated in the pumping lemma; $y \neq \epsilon$ and $|xy| \leq n$. Again, player 2 does not have to tell player 1 what x, y, and z are, although they must obey the constraints.
5. Player 1 "wins" by picking k, which may be a function of n, x, y, and z, such that xy^kz is not in L.

Example 4.2: Let us show that the language L_{eq} consisting of all strings with an equal number of 0's and 1's (not in any particular order) is not a regular language. In terms of the "two-player game" described in the box on "The Pumping Lemma as an Adversarial Game," we shall be player 1 and we must deal with whatever choices player 2 makes. Suppose n is the constant that must exist if L_{eq} is regular, according to the pumping lemma; i.e., "player 2" picks n. We shall pick $w = 0^n1^n$, that is, n 0's followed by n 1's, a string that surely is in L_{eq}.

Now, "player 2" breaks our w up into xyz. All we know is that $y \neq \epsilon$, and $|xy| \leq n$. However, that information is very useful, and we "win" as follows. Since $|xy| \leq n$, and xy comes at the front of w, we know that x and y consist only of 0's. The pumping lemma tells us that xz is in L_{eq}, if L_{eq} is regular. This conclusion is the case $k = 0$ in the pumping lemma.[1] However, xz has n 1's, since all the 1's of w are in z. But xz also has fewer than n 0's, because we

[1] Observe in what follows that we could have also succeeded by picking $k = 2$, or indeed any value of k other than 1.

lost the 0's of y. Since $y \neq \epsilon$ we know that there can be no more than $n-1$ 0's among x and z. Thus, after assuming L_{eq} is a regular language, we have proved a fact known to be false, that xz is in L_{eq}. We have a proof by contradiction of the fact that L_{eq} is not regular. □

Example 4.3: Let us show that the language L_{pr} consisting of all strings of 1's whose length is a prime is not a regular language. Suppose it were. Then there would be a constant n satisfying the conditions of the pumping lemma. Consider some prime $p \geq n+2$; there must be such a p, since there are an infinity of primes. Let $w = 1^p$.

By the pumping lemma, we can break $w = xyz$ such that $y \neq \epsilon$ and $|xy| \leq n$. Let $|y| = m$. Then $|xz| = p - m$. Now consider the string $xy^{p-m}z$, which must be in L_{pr} by the pumping lemma, if L_{pr} really is regular. However,

$$|xy^{p-m}z| = |xz| + (p-m)|y| = p - m + (p-m)m = (m+1)(p-m)$$

It looks like $|xy^{p-m}z|$ is not a prime, since it has two factors $m+1$ and $p-m$. However, we must check that neither of these factors are 1, since then $(m+1)(p-m)$ might be a prime after all. But $m+1 > 1$, since $y \neq \epsilon$ tells us $m \geq 1$. Also, $p - m > 1$, since $p \geq n+2$ was chosen, and $m \leq n$ since

$$m = |y| \leq |xy| \leq n$$

Thus, $p - m \geq 2$.

Again we have started by assuming the language in question was regular, and we derived a contradiction by showing that some string not in the language was required by the pumping lemma to be in the language. Thus, we conclude that L_{pr} is not a regular language. □

4.1.3 Exercises for Section 4.1

Exercise 4.1.1: Prove that the following are not regular languages.

a) $\{0^n1^n \mid n \geq 1\}$. This language, consisting of a string of 0's followed by an equal-length string of 1's, is the language L_{01} we considered informally at the beginning of the section. Here, you should apply the pumping lemma in the proof.

b) The set of strings of balanced parentheses. These are the strings of characters "(" and ")" that can appear in a well-formed arithmetic expression.

* c) $\{0^n10^n \mid n \geq 1\}$.

d) $\{0^n1^m2^n \mid n$ and m are arbitrary integers$\}$.

e) $\{0^n1^m \mid n \leq m\}$.

f) $\{0^n1^{2n} \mid n \geq 1\}$.

! **Exercise 4.1.2:** Prove that the following are not regular languages.

* a) $\{0^n \mid n \text{ is a perfect square}\}$.

b) $\{0^n \mid n \text{ is a perfect cube}\}$.

c) $\{0^n \mid n \text{ is a power of 2}\}$.

d) The set of strings of 0's and 1's whose length is a perfect square.

e) The set of strings of 0's and 1's that are of the form ww, that is, some string repeated.

f) The set of strings of 0's and 1's that are of the form ww^R, that is, some string followed by its reverse. (See Section 4.2.2 for a formal definition of the reversal of a string.)

g) The set of strings of 0's and 1's of the form $w\overline{w}$, where $\overline{w}$ is formed from w by replacing all 0's by 1's, and vice-versa; e.g., $\overline{011} = 100$, and 011100 is an example of a string in the language.

h) The set of strings of the form $w1^n$, where w is a string of 0's and 1's of length n.

! **Exercise 4.1.3:** Prove that the following are not regular languages.

a) The set of strings of 0's and 1's, beginning with a 1, such that when interpreted as an integer, that integer is a prime.

b) The set of strings of the form 0^i1^j such that the greatest common divisor of i and j is 1.

! **Exercise 4.1.4:** When we try to apply the pumping lemma to a regular language, the "adversary wins," and we cannot complete the proof. Show what goes wrong when we choose L to be one of the following languages:

* a) The empty set.

* b) $\{00, 11\}$.

* c) $(00 + 11)^*$.

d) 01^*0^*1.

4.2 Closure Properties of Regular Languages

In this section, we shall prove several theorems of the form "if certain languages are regular, and a language L is formed from them by certain operations (e.g., L is the union of two regular languages), then L is also regular." These theorems are often called *closure properties* of the regular languages, since they show that the class of regular languages is closed under the operation mentioned. Closure properties express the idea that when one (or several) languages are regular, then certain related languages are also regular. They also serve as an interesting illustration of how the equivalent representations of the regular languages (automata and regular expressions) reinforce each other in our understanding of the class of languages, since often one representation is far better than the others in supporting a proof of a closure property. Here is a summary of the principal closure properties for regular languages:

1. The union of two regular languages is regular.
2. The intersection of two regular languages is regular.
3. The complement of a regular language is regular.
4. The difference of two regular languages is regular.
5. The reversal of a regular language is regular.
6. The closure (star) of a regular language is regular.
7. The concatenation of regular languages is regular.
8. A homomorphism (substitution of strings for symbols) of a regular language is regular.
9. The inverse homomorphism of a regular language is regular.

4.2.1 Closure of Regular Languages Under Boolean Operations

Our first closure properties are the three boolean operations: union, intersection, and complementation:

1. Let L and M be languages over alphabet Σ. Then $L \cup M$ is the language that contains all strings that are in either or both of L and M.
2. Let L and M be languages over alphabet Σ. Then $L \cap M$ is the language that contains all strings that are in both L and M.
3. Let L be a language over alphabet Σ. Then $\overline{L}$, the *complement* of L, is the set of strings in Σ^* that are not in L.

It turns out that the regular languages are closed under all three of the boolean operations. The proofs take rather different approaches though, as we shall see.

What if Languages Have Different Alphabets?

When we take the union or intersection of two languages L and M, they might have different alphabets. For example, it is possible that $L_1 \subseteq \{a, b\}$ while $L_2 \subseteq \{b, c, d\}$. However, if a language L consists of strings with symbols in Σ, then we can also think of L as a language over any finite alphabet that is a superset of Σ. Thus, for example, we can think of both L_1 and L_2 above as being languages over alphabet $\{a, b, c, d\}$. The fact that none of L_1's strings contain symbols c or d is irrelevant, as is the fact that L_2's strings will not contain a.

Likewise, when taking the complement of a language L that is a subset of Σ_1^* for some alphabet Σ_1, we may choose to take the complement *with respect to* some alphabet Σ_2 that is a superset of Σ_1. If so, then the complement of L will be $\Sigma_2^* - L$; that is, the complement of L with respect to Σ_2 includes (among other strings) all those strings in Σ_2^* that have at least one symbol that is in Σ_2 but not in Σ_1. Had we taken the complement of L with respect to Σ_1, then no string with symbols in $\Sigma_2 - \Sigma_1$ would be in $\overline{L}$. Thus, to be strict, we should always state the alphabet with respect to which a complement is taken. However, often it is obvious which alphabet is meant; e.g., if L is defined by an automaton, then the specification of that automaton includes the alphabet. Thus, we shall often speak of the "complement" without specifying the alphabet.

Closure Under Union

Theorem 4.4: If L and M are regular languages, then so is $L \cup M$.

PROOF: This proof is simple. Since L and M are regular, they have regular expressions; say $L = L(R)$ and $M = L(S)$. Then $L \cup M = L(R + S)$ by the definition of the + operator for regular expressions. □

Closure Under Complementation

The theorem for union was made very easy by the use of the regular-expression representation for the languages. However, let us next consider complementation. Do you see how to take a regular expression and change it into one that defines the complement language? Well neither do we. However, it can be done, because as we shall see in Theorem 4.5, it is easy to start with a DFA and construct a DFA that accepts the complement. Thus, starting with a regular expression, we could find a regular expression for its complement as follows:

1. Convert the regular expression to an ϵ-NFA.
2. Convert that ϵ-NFA to a DFA by the subset construction.

Closure Under Regular Operations

The proof that regular languages are closed under union was exceptionally easy because union is one of the three operations that define the regular expressions. The same idea as Theorem 4.4 applies to concatenation and closure as well. That is:

- If L and M are regular languages, then so is LM.
- If L is a regular language, then so is L^*.

3. Complement the accepting states of that DFA.
4. Turn the complement DFA back into a regular expression using the construction of Sections 3.2.1 or 3.2.2.

Theorem 4.5: If L is a regular language over alphabet Σ, then $\overline{L} = \Sigma^* - L$ is also a regular language.

PROOF: Let $L = L(A)$ for some DFA $A = (Q, \Sigma, \delta, q_0, F)$. Then $\overline{L} = L(B)$, where B is the DFA $(Q, \Sigma, \delta, q_0, Q - F)$. That is, B is exactly like A, but the accepting states of A have become nonaccepting states of B, and vice versa. Then w is in $L(B)$ if and only if $\hat{\delta}(q_0, w)$ is in $Q - F$, which occurs if and only if w is not in $L(A)$. □

Notice that it is important for the above proof that $\hat{\delta}(q_0, w)$ is always some state; i.e., there are no missing transitions in A. If there were, then certain strings might lead neither to an accepting nor nonaccepting state of A, and those strings would be missing from both $L(A)$ and $L(B)$. Fortunately, we have defined a DFA to have a transition on every symbol of Σ from every state, so each string leads either to a state in F or a state in $Q - F$.

Example 4.6: Let A be the automaton of Fig. 2.14. Recall that DFA A accepts all and only the strings of 0's and 1's that end in 01; in regular-expression terms, $L(A) = (0 + 1)^*01$. The complement of $L(A)$ is therefore all strings of 0's and 1's that do *not* end in 01. Figure 4.2 shows the automaton for $\{0, 1\}^* - L(A)$. It is the same as Fig. 2.14 but with the accepting state made nonaccepting and the two nonaccepting states made accepting. □

Example 4.7: In this example, we shall apply Theorem 4.5 to show a certain language not to be regular. In Example 4.2 we showed that the language L_{eq} consisting of strings with an equal number of 0's and 1's and is not regular. This proof was a straightforward application of the pumping lemma. Now consider

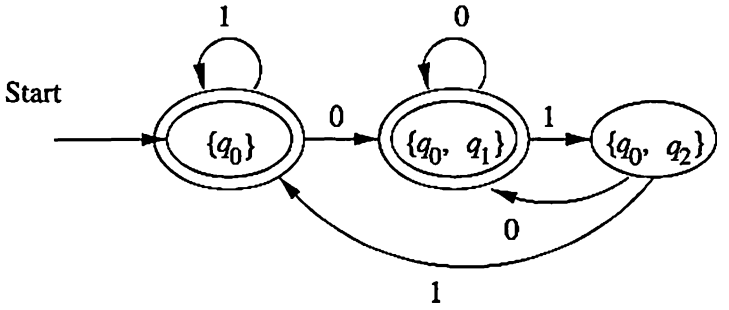


Figure 4.2: DFA accepting the complement of the language $(\mathbf{0+1})^*\mathbf{01}$

the language M consisting of those strings of 0's and 1's that have an unequal number of 0's and 1's.

It would be hard to use the pumping lemma to show M is not regular. Intuitively, if we start with some string w in M, break it into $w = xyz$, and "pump" y, we might find that y itself was a string like 01 that had an equal number of 0's and 1's. If so, then for no k will xy^kz have an equal number of 0's and 1's, since xyz has an unequal number of 0's and 1's, and the numbers of 0's and 1's change equally as we "pump" y. Thus, we can never use the pumping lemma to contradict the assumption that M is regular.

However, M is still not regular. The reason is that $M = \overline{L}$. Since the complement of the complement is the set we started with, it also follows that $L = \overline{M}$. If M is regular, then by Theorem 4.5, L is regular. But we know L is *not* regular, so we have a proof by contradiction that M is not regular. □

Closure Under Intersection

Now, let us consider the intersection of two regular languages. We actually have little to do, since the three boolean operations are not independent. Once we have ways of performing complementation and union, we can obtain the intersection of languages L and M by the identity

$$L \cap M = \overline{\overline{L} \cup \overline{M}} \tag{4.1}$$

In general, the intersection of two sets is the set of elements that are not in the complement of either set. That observation, which is what Equation (4.1) says, is one of *DeMorgan's laws*. The other law is the same with union and intersection interchanged; that is, $L \cup M = \overline{\overline{L} \cap \overline{M}}$.

However, we can also perform a direct construction of a DFA for the intersection of two regular languages. This construction, which essentially runs two DFA's in parallel, is useful in its own right. For instance, we used it to construct the automaton in Fig. 2.3 that represented the "product" of what two participants — the bank and the store — were doing. We shall make the *product construction* formal in the next theorem.

Theorem 4.8: If L and M are regular languages, then so is $L \cap M$.

PROOF: Let L and M be the languages of automata $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ and $A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$. Notice that we are assuming that the alphabets of both automata are the same; that is, Σ is the union of the alphabets of L and M, if those alphabets are different. The product construction actually works for NFA's as well as DFA's, but to make the argument as simple as possible, we assume that A_L and A_M are DFA's.

For $L \cap M$ we shall construct an automaton A that simulates both A_L and A_M. The states of A are pairs of states, the first from A_L and the second from A_M. To design the transitions of A, suppose A is in state (p, q), where p is the state of A_L and q is the state of A_M. If a is the input symbol, we see what A_L does on input a; say it goes to state s. We also see what A_M does on input a; say it makes a transition to state t. Then the next state of A will be (s, t). In that manner, A has simulated the effect of both A_L and A_M. The idea is sketched in Fig. 4.3.

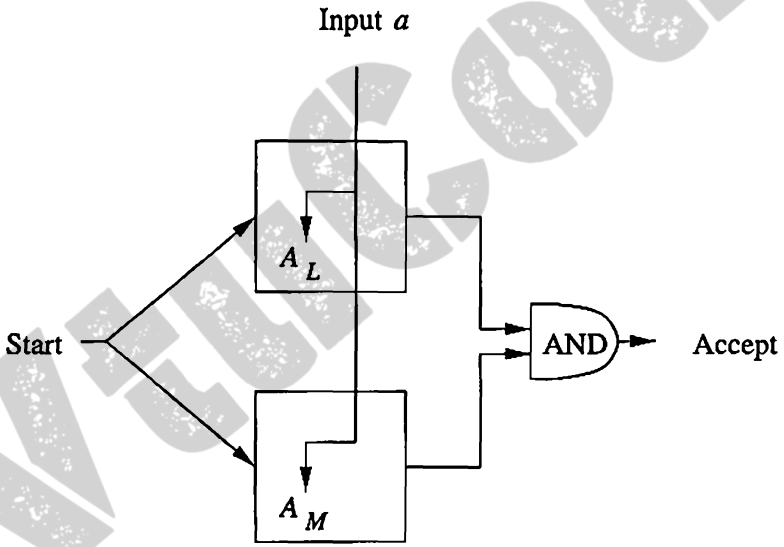


Figure 4.3: An automaton simulating two other automata and accepting if and only if both accept

The remaining details are simple. The start state of A is the pair of start states of A_L and A_M. Since we want to accept if and only if both automata accept, we select as the accepting states of A all those pairs (p, q) such that p is an accepting state of A_L and q is an accepting state of A_M. Formally, we define:

$$A = (Q_L \times Q_M, \Sigma, \delta, (q_L, q_M), F_L \times F_M)$$

where $\delta\big((p, q), a\big) = \big(\delta_L(p, a), \delta_M(q, a)\big)$.

To see why $L(A) = L(A_L) \cap L(A_M)$, first observe that an easy induction on $|w|$ proves that $\hat{\delta}\big((q_L, q_M), w\big) = \big(\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w)\big)$. But A accepts w if and only if $\hat{\delta}\big((q_L, q_M), w\big)$ is a pair of accepting states. That is, $\hat{\delta}_L(q_L, w)$ must be in F_L, and $\hat{\delta}_M(q_M, w)$ must be in F_M. Put another way, w is accepted by A if and only if both A_L and A_M accept w. Thus, A accepts the intersection of L and M. □

Example 4.9: In Fig. 4.4 we see two DFA's. The automaton in Fig. 4.4(a) accepts all those strings that have a 0, while the automaton in Fig. 4.4(b) accepts all those strings that have a 1. We show in Fig. 4.4(c) the product of these two automata. Its states are labeled by the pairs of states of the automata in (a) and (b).

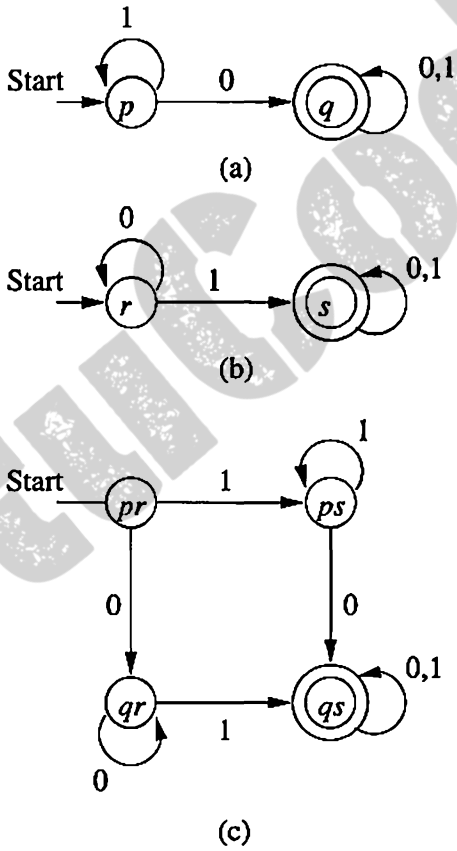


Figure 4.4: The product construction

It is easy to argue that this automaton accepts the intersection of the first two languages: those strings that have both a 0 and a 1. State *pr* represents only the initial condition, in which we have seen neither 0 nor 1. State *qr* means that we have seen only 0's, while state *ps* represents the condition that we have

seen only 1's. The accepting state qs represents the condition where we have seen both 0's and 1's. □

Closure Under Difference

There is a fourth operation that is often applied to sets and is related to the boolean operations: set difference. In terms of languages, $L - M$, the *difference* of L and M, is the set of strings that are in language L but not in language M. The regular languages are also closed under this operation, and the proof follows easily from the theorems just proven.

Theorem 4.10: If L and M are regular languages, then so is $L - M$.

PROOF: Observe that $L - M = L \cap \overline{M}$. By Theorem 4.5, $\overline{M}$ is regular, and by Theorem 4.8 $L \cap \overline{M}$ is regular. Therefore $L - M$ is regular. □

4.2.2 Reversal

The *reversal* of a string $a_1 a_2 \cdots a_n$ is the string written backwards, that is, $a_n a_{n-1} \cdots a_1$. We use w^R for the reversal of string w. Thus, 0010^R is 0100, and $\epsilon^R = \epsilon$.

The reversal of a language L, written L^R, is the language consisting of the reversals of all its strings. For instance, if $L = \{001, 10, 111\}$, then $L^R = \{100, 01, 111\}$.

Reversal is another operation that preserves regular languages; that is, if L is a regular language, so is L^R. There are two simple proofs, one based on automata and one based on regular expressions. We shall give the automaton-based proof informally, and let you fill in the details if you like. We then prove the theorem formally using regular expressions.

Given a language L that is $L(A)$ for some finite automaton, perhaps with nondeterminism and ϵ-transitions, we may construct an automaton for L^R by:

1. Reverse all the arcs in the transition diagram for A.
2. Make the start state of A be the only accepting state for the new automaton.
3. Create a new start state p_0 with transitions on ϵ to all the accepting states of A.

The result is an automaton that simulates A "in reverse," and therefore accepts a string w if and only if A accepts w^R. Now, we prove the reversal theorem formally.

Theorem 4.11: If L is a regular language, so is L^R.

PROOF: Assume L is defined by regular expression E. The proof is a structural induction on the size of E. We show that there is another regular expression E^R such that $L(E^R) = \big(L(E)\big)^R$; that is, the language of E^R is the reversal of the language of E.

BASIS: If E is ϵ, $\emptyset$, or $\mathbf{a}$, for some symbol a, then E^R is the same as E. That is, we know $\{\epsilon\}^R = \{\epsilon\}$, $\emptyset^R = \emptyset$, and $\{a\}^R = \{a\}$.

INDUCTION: There are three cases, depending on the form of E.

1. $E = E_1 + E_2$. Then $E^R = E_1^R + E_2^R$. The justification is that the reversal of the union of two languages is obtained by computing the reversals of the two languages and taking the union of those languages.

2. $E = E_1E_2$. Then $E^R = E_2^R E_1^R$. Note that we reverse the order of the two languages, as well as reversing the languages themselves. For instance, if $L(E_1) = \{01, 111\}$ and $L(E_2) = \{00, 10\}$, then $L(E_1E_2) = \{0100, 0110, 11100, 11110\}$. The reversal of the latter language is

$$\{0010, 0110, 00111, 01111\}$$

   If we concatenate the reversals of $L(E_2)$ and $L(E_1)$ in that order, we get

$$\{00, 01\}\{10, 111\} = \{0010, 00111, 0110, 01111\}$$

   which is the same language as $\big(L(E_1E_2)\big)^R$. In general, if a word w in $L(E)$ is the concatenation of w_1 from $L(E_1)$ and w_2 from $L(E_2)$, then $w^R = w_2^R w_1^R$.

3. $E = E_1^*$. Then $E^R = (E_1^R)^*$. The justification is that any string w in $L(E)$ can be written as $w_1w_2 \cdots w_n$, where each w_i is in $L(E)$. But

$$w^R = w_n^R w_{n-1}^R \cdots w_1^R$$

   Each w_i^R is in $L(E^R)$, so w^R is in $L\big((E_1^R)^*\big)$. Conversely, any string in $L\big((E_1^R)^*\big)$ is of the form $w_1w_2 \cdots w_n$, where each w_i is the reversal of a string in $L(E_1)$. The reversal of this string, $w_n^R w_{n-1}^R \cdots w_1^R$, is therefore a string in $L(E_1^*)$, which is $L(E)$. We have thus shown that a string is in $L(E)$ if and only if its reversal is in $L\big((E_1^R)^*\big)$.

□

Example 4.12: Let L be defined by the regular expression $(\mathbf{0} + \mathbf{1})\mathbf{0}^*$. Then L^R is the language of $(\mathbf{0}^*)^R(\mathbf{0} + \mathbf{1})^R$, by the rule for concatenation. If we apply the rules for closure and union to the two parts, and then apply the basis rule that says the reversals of $\mathbf{0}$ and $\mathbf{1}$ are unchanged, we find that L^R has regular expression $\mathbf{0}^*(\mathbf{0} + \mathbf{1})$. □

4.2.3 Homomorphisms

A string *homomorphism* is a function on strings that works by substituting a particular string for each symbol.

Example 4.13: The function h defined by $h(0) = ab$ and $h(1) = \epsilon$ is a homomorphism. Given any string of 0's and 1's, it replaces all 0's by the string ab and replaces all 1's by the empty string. For example, h applied to the string 0011 is $abab$. □

Formally, if h is a homomorphism on alphabet Σ, and $w = a_1a_2\cdots a_n$ is a string of symbols in Σ, then $h(w) = h(a_1)h(a_2)\cdots h(a_n)$. That is, we apply h to each symbol of w and concatenate the results, in order. For instance, if h is the homomorphism in Example 4.13, and $w = 0011$, then $h(w) = h(0)h(0)h(1)h(1) = (ab)(ab)(\epsilon)(\epsilon) = abab$, as we claimed in that example.

Further, we can apply a homomorphism to a language by applying it to each of the strings in the language. That is, if L is a language over alphabet Σ, and h is a homomorphism on Σ, then $h(L) = \{h(w) \mid w \text{ is in } L\}$. For instance, if L is the language of regular expression $\mathbf{10^*1}$, i.e., any number of 0's surrounded by single 1's, then $h(L)$ is the language $(\mathbf{ab})^*$. The reason is that h of Example 4.13 effectively drops the 1's, since they are replaced by ϵ, and turns each 0 into ab. The same idea, applying the homomorphism directly to the regular expression, can be used to prove that the regular languages are closed under homomorphisms.

Theorem 4.14: If L is a regular language over alphabet Σ, and h is a homomorphism on Σ, then $h(L)$ is also regular.

PROOF: Let $L = L(R)$ for some regular expression R. In general, if E is a regular expression with symbols in Σ, let $h(E)$ be the expression we obtain by replacing each symbol a of Σ in E by $h(a)$. We claim that $h(R)$ defines the language $h(L)$.

The proof is an easy structural induction that says whenever we take a subexpression E of R and apply h to it to get $h(E)$, the language of $h(E)$ is the same language we get if we apply h to the language $L(E)$. Formally, $L\big(h(E)\big) = h\big(L(E)\big)$.

BASIS: If E is ϵ or $\emptyset$, then $h(E)$ is the same as E, since h does not affect the string ϵ or the language $\emptyset$. Thus, $L\big(h(E)\big) = L(E)$. However, if E is $\emptyset$ or ϵ, then $L(E)$ contains either no strings or a string with no symbols, respectively. Thus $h\big(L(E)\big) = L(E)$ in either case. We conclude $L\big(h(E)\big) = L(E) = h\big(L(E)\big)$.

The only other basis case is if $E = \mathbf{a}$ for some symbol a in Σ. In this case, $L(E) = \{a\}$, so $h\big(L(E)\big) = \{h(a)\}$. Also, $h(E)$ is the regular expression that is the string of symbols $h(a)$. Thus, $L\big(h(E)\big)$ is also $\{h(a)\}$, and we conclude $L\big(h(E)\big) = h\big(L(E)\big)$.

INDUCTION: There are three cases, each of them simple. We shall prove only the union case, where $E = F+G$. The way we apply homomorphisms to regular expressions assures us that $h(E) = h(F + G) = h(F) + h(G)$. We also know that $L(E) = L(F) \cup L(G)$ and

$$L\big(h(E)\big) = L\big(h(F) + h(G)\big) = L\big(h(F)\big) \cup L\big(h(G)\big) \tag{4.2}$$

by the definition of what "+" means in regular expressions. Finally,

$$h\big(L(E)\big) = h\big(L(F) \cup L(G)\big) = h\big(L(F)\big) \cup h\big(L(G)\big) \tag{4.3}$$

because h is applied to a language by application to each of its strings individually. Now we may invoke the inductive hypothesis to assert that $L\big(h(F)\big) = h\big(L(F)\big)$ and $L\big(h(G)\big) = h\big(L(G)\big)$. Thus, the final expressions in (4.2) and (4.3) are equivalent, and therefore so are their respective first terms; that is, $L\big(h(E)\big) = h\big(L(E)\big)$.

We shall not prove the cases where expression E is a concatenation or closure; the ideas are similar to the above in both cases. The conclusion is that $L\big(h(R)\big)$ is indeed $h\big(L(R)\big)$; i.e., applying the homomorphism h to the regular expression for language L results in a regular expression that defines the language $h(L)$. □

4.2.4 Inverse Homomorphisms

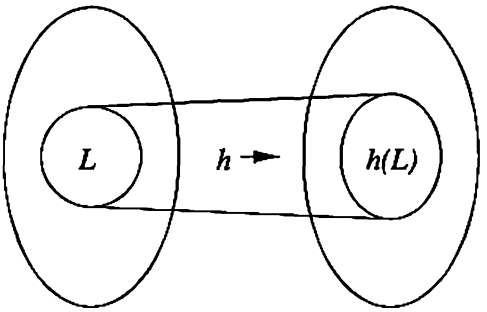
Homomorphisms may also be applied "backwards," and in this mode they also preserve regular languages. That is, suppose h is a homomorphism from some alphabet Σ to strings in another (possibly the same) alphabet T.[2] Let L be a language over alphabet T. Then $h^{-1}(L)$, read "h inverse of L," is the set of strings w in Σ^* such that $h(w)$ is in L. Figure 4.5 suggests the effect of a homomorphism on a language L in part (a), and the effect of an inverse homomorphism in part (b).

Example 4.15: Let L be the language of regular expression $(\mathbf{00} + \mathbf{1})^*$. That is, L consists of all strings of 0's and 1's such that all the 0's occur in adjacent pairs. Thus, 0010011 and 10000111 are in L, but 000 and 10100 are not.

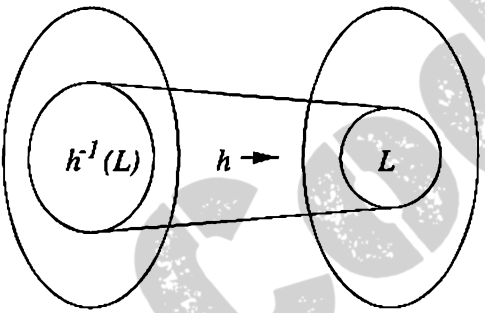
Let h be the homomorphism defined by $h(a) = 01$ and $h(b) = 10$. We claim that $h^{-1}(L)$ is the language of regular expression $(\mathbf{ba})^*$, that is, all strings of repeating ba pairs. We shall prove that $h(w)$ is in L if and only if w is of the form $baba\cdots ba$.

(If) Suppose w is n repetitions of ba for some $n \geq 0$. Note that $h(ba) = 1001$, so $h(w)$ is n repetitions of 1001. Since 1001 is composed of two 1's and a pair of 0's, we know that 1001 is in L. Therefore any repetition of 1001 is also formed from 1 and 00 segments and is in L. Thus, $h(w)$ is in L.

[2] That "T" should be thought of as a Greek capital tau, the letter following sigma.



(a)



(b)

Figure 4.5: A homomorphism applied in the forward and inverse direction

(Only-if) Now, we must assume that $h(w)$ is in L and show that w is of the form $baba\cdots ba$. There are four conditions under which a string is *not* of that form, and we shall show that if any of them hold then $h(w)$ is not in L. That is, we prove the contrapositive of the statement we set out to prove.

1. If w begins with a, then $h(w)$ begins with 01. It therefore has an isolated 0, and is not in L.

2. If w ends in b, then $h(w)$ ends in 10, and again there is an isolated 0 in $h(w)$.

3. If w has two consecutive a's, then $h(w)$ has a substring 0101. Here too, there is an isolated 0 in w.

4. Likewise, if w has two consecutive b's, then $h(w)$ has substring 1010 and has an isolated 0.

Thus, whenever one of the above cases hold, $h(w)$ is not in L. However, unless at least one of items (1) through (4) hold, then w is of the form $baba\cdots ba$.

To see why, assume none of (1) through (4) hold. Then (1) tells us w must begin with b, and (2) tells us w ends with a. Statements (3) and (4) tell us that a's and b's must alternate in w. Thus, the logical "OR" of (1) through (4) is equivalent to the statement "w is not of the form $baba\cdots ba$." We have proved that the "OR" of (1) through (4) implies $h(w)$ is not in L. That statement is the contrapositive of the statement we wanted: "if $h(w)$ is in L, then w is of the form $baba\cdots ba$." □

We shall next prove that the inverse homomorphism of a regular language is also regular, and then show how the theorem can be used.

Theorem 4.16: If h is a homomorphism from alphabet Σ to alphabet T, and L is a regular language over T, then $h^{-1}(L)$ is also a regular language.

PROOF: The proof starts with a DFA A for L. We construct from A and h a DFA for $h^{-1}(L)$ using the plan suggested by Fig. 4.6. This DFA uses the states of A but translates the input symbol according to h before deciding on the next state.

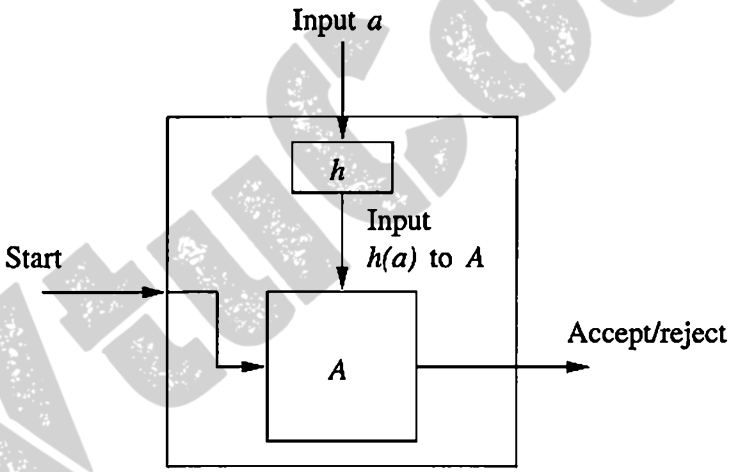


Figure 4.6: The DFA for $h^{-1}(L)$ applies h to its input, and then simulates the DFA for L

Formally, let L be $L(A)$, where DFA $A = (Q, T, \delta, q_0, F)$. Define a DFA

$$B = (Q, \Sigma, \gamma, q_0, F)$$

where transition function γ is constructed by the rule $\gamma(q, a) = \hat{\delta}\big(q, h(a)\big)$. That is, the transition B makes on input a is the result of the sequence of transitions that A makes on the string of symbols $h(a)$. Remember that $h(a)$ could be ϵ, it could be one symbol, or it could be many symbols, but $\hat{\delta}$ is properly defined to take care of all these cases.

It is an easy induction on $|w|$ to show that $\hat{\gamma}(q_0, w) = \hat{\delta}\big(q_0, h(w)\big)$. Since the accepting states of A and B are the same, B accepts w if and only if A accepts $h(w)$. Put another way, B accepts exactly those strings w that are in $h^{-1}(L)$. □

Example 4.17: In this example we shall use inverse homomorphism and several other closure properties of regular sets to prove an odd fact about finite automata. Suppose we required that a DFA visit every state at least once when accepting its input. More precisely, suppose $A = (Q, \Sigma, \delta, q_0, F)$ is a DFA, and we are interested in the language L of all strings w in Σ^* such that $\hat{\delta}(q_0, w)$ is in F, and also for every state q in Q there is some prefix x_q of w such that $\hat{\delta}(q_0, x_q) = q$. Is L regular? We can show it is, but the construction is complex.

First, start with the language M that is $L(A)$, i.e., the set of strings that A accepts in the usual way, without regard to what states it visits during the processing of its input. Note that $L \subseteq M$, since the definition of L puts an additional condition on the strings of $L(A)$. Our proof that L is regular begins by using an inverse homomorphism to, in effect, place the states of A into the input symbols. More precisely, let us define a new alphabet T consisting of symbols that we may think of as triples $[paq]$, where:

1. p and q are states in Q,
2. a is a symbol in Σ, and
3. $\delta(p, a) = q$.

That is, we may think of the symbols in T as representing transitions of the automaton A. It is important to see that the notation $[paq]$ is our way of expressing a single symbol, not the concatenation of three symbols. We could have given it a single letter as a name, but then its relationship to p, q, and a would be hard to describe.

Now, define the homomorphism $h([paq]) = a$ for all p, a, and q. That is, h removes the state components from each of the symbols of T and leaves only the symbol from Σ. Our first step in showing L is regular is to construct the language $L_1 = h^{-1}(M)$. Since M is regular, so is L_1 by Theorem 4.16. The strings of L_1 are just the strings of M with a pair of states, representing a transition, attached to each symbol.

As a very simple illustration, consider the two-state automaton of Fig. 4.4(a). The alphabet Σ is $\{0, 1\}$, and the alphabet T consists of the four symbols $[p0q]$, $[q0q]$, $[p1p]$, and $[q1q]$. For instance, there is a transition from state p to q on input 0, so $[p0q]$ is one of the symbols of T. Since 101 is a string accepted by the automaton, h^{-1} applied to this string will give us $2^3 = 8$ strings, of which $[p1p][p0q][q1q]$ and $[q1q][q0q][p1p]$ are two examples.

We shall now construct L from L_1 by using a series of further operations that preserve regular languages. Our first goal is to eliminate all those strings of L_1 that deal incorrectly with states. That is, we can think of a symbol like

[paq] as saying the automaton was in state p, read input a, and thus entered state q. The sequence of symbols must satisfy three conditions if it is to be deemed an accepting computation of A:

1. The first state in the first symbol must be q_0, the start state of A.
2. Each transition must pick up where the previous one left off. That is, the first state in one symbol must equal the second state of the previous symbol.
3. The second state of the last symbol must be in F. This condition in fact will be guaranteed once we enforce (1) and (2), since we know that every string in L_1 came from a string accepted by A.

The plan of the construction of L is shown in Fig. 4.7.

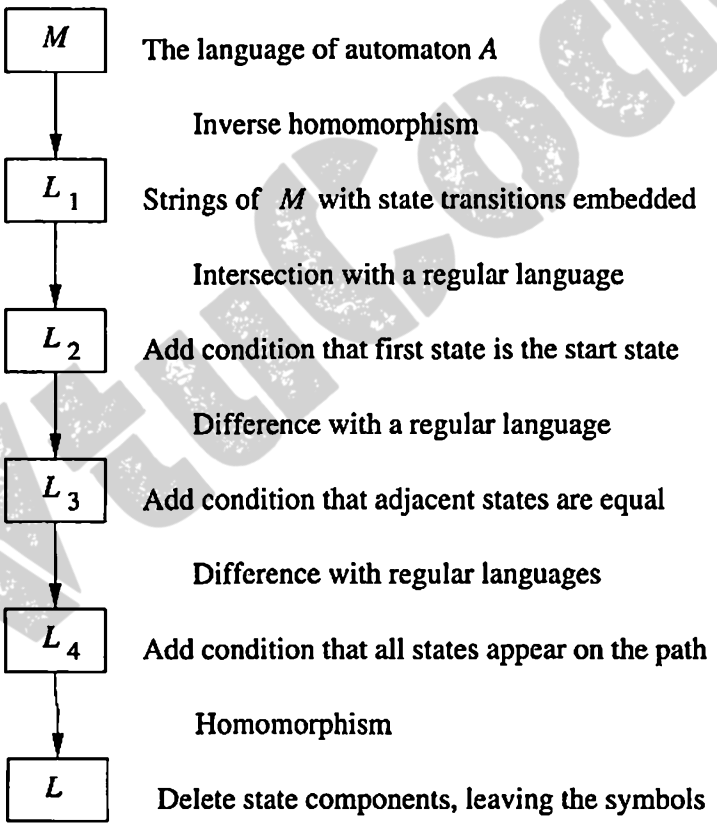


Figure 4.7: Constructing language L from language M by applying operations that preserve regularity of languages

We enforce (1) by intersecting L_1 with the set of strings that begin with a symbol of the form [q_0aq] for some symbol a and state q. That is, let E_1 be the

expression $[q_0a_1q_1] + [q_0a_2q_2] + \cdots$, where the pairs a_iq_i range over all pairs in $\Sigma \times Q$ such that $\delta(q_0, a_i) = q_i$. Then let $L_2 = L_1 \cap L(E_1T^*)$. Since E_1T^* is a regular expression denoting all strings in T^* that begin with the start state (treat T in the regular expression as the sum of its symbols), L_2 is all strings that are formed by applying h^{-1} to language M and that have the start state as the first component of its first symbol; i.e., it meets condition (1).

To enforce condition (2), it is easier to subtract from L_2 (using the set-difference operation) all those strings that violate it. Let E_2 be the regular expression consisting of the sum (union) of the concatenation of all pairs of symbols that fail to match; that is, pairs of the form $[paq][rbs]$ where $q \neq r$. Then $T^*E_2T^*$ is a regular expression denoting all strings that fail to meet condition (2).

We may now define $L_3 = L_2 - L(T^*E_2T^*)$. The strings of L_3 satisfy condition (1) because strings in L_2 must begin with the start symbol. They satisfy condition (2) because the subtraction of $L(T^*E_2T^*)$ removes any string that violates that condition. Finally, they satisfy condition (3), that the last state is accepting, because we started with only strings in M, all of which lead to acceptance by A. The effect is that L_3 consists of the strings in M with the states of the accepting computation of that string embedded as part of each symbol. Note that L_3 is regular because it is the result of starting with the regular language M, and applying operations — inverse homomorphism, intersection, and set difference — that yield regular sets when applied to regular sets.

Recall that our goal was to accept only those strings in M that visited every state in their accepting computation. We may enforce this condition by additional applications of the set-difference operator. That is, for each state q, let E_q be the regular expression that is the sum of all the symbols in T such that q appears in neither its first or last position. If we subtract $L(E_q^*)$ from L_3 we have those strings that are an accepting computation of A and that visit state q at least once. If we subtract from L_3 all the languages $L(E_q^*)$ for q in Q, then we have the accepting computations of A that visit all the states. Call this language L_4. By Theorem 4.10 we know L_4 is also regular.

Our final step is to construct L from L_4 by getting rid of the state components. That is, $L = h(L_4)$. Now, L is the set of strings in Σ^* that are accepted by A and that visit each state of A at least once during their acceptance. Since regular languages are closed under homomorphisms, we conclude that L is regular. □

4.2.5 Exercises for Section 4.2

Exercise 4.2.1: Suppose h is the homomorphism from the alphabet $\{0, 1, 2\}$ to the alphabet $\{a, b\}$ defined by: $h(0) = a$; $h(1) = ab$, and $h(2) = ba$.

* a) What is $h(0120)$?

b) What is $h(21120)$?

* c) If L is the language $L(\mathbf{01^*2})$, what is $h(L)$?

d) If L is the language $L(\mathbf{0+12})$, what is $h(L)$?

* e) Suppose L is the language $\{ababa\}$, that is, the language consisting of only the one string $ababa$. What is $h^{-1}(L)$?

! f) If L is the language $L(\mathbf{a(ba)^*})$, what is $h^{-1}(L)$?

*! **Exercise 4.2.2:** If L is a language, and a is a symbol, then L/a, the *quotient* of L and a, is the set of strings w such that wa is in L. For example, if $L = \{a, aab, baa\}$, then $L/a = \{\epsilon, ba\}$. Prove that if L is regular, so is L/a. *Hint*: Start with a DFA for L and consider the set of accepting states.

! **Exercise 4.2.3:** Lf L is a language, and a is a symbol, then $a\backslash L$ is the set of strings w such that aw is in L. For example, if $L = \{a, aab, baa\}$, then $a\backslash L = \{\epsilon, ab\}$. Prove that if L is regular, so is $a\backslash L$. *Hint*: Remember that the regular languages are closed under reversal and under the quotient operation of Exercise 4.2.2.

! **Exercise 4.2.4:** Which of the following identities are true?

a) $(L/a)a = L$ (the left side represents the concatenation of the languages L/a and $\{a\}$).

b) $a(a\backslash L) = L$ (again, concatenation with $\{a\}$, this time on the left, is intended).

c) $(La)/a = L$.

d) $a\backslash(aL) = L$.

Exercise 4.2.5: The operation of Exercise 4.2.3 is sometimes viewed as a "derivative," and $a\backslash L$ is written $\frac{dL}{da}$. These derivatives apply to regular expressions in a manner similar to the way ordinary derivatives apply to arithmetic expressions. Thus, if R is a regular expression, we shall use $\frac{dR}{da}$ to mean the same as $\frac{dL}{da}$, if $L = L(R)$.

a) Show that $\frac{d(R+S)}{da} = \frac{dR}{da} + \frac{dS}{da}$.

*! b) Give the rule for the "derivative" of RS. *Hint*: You need to consider two cases: if $L(R)$ does or does not contain ϵ. This rule is not quite the same as the "product rule" for ordinary derivatives, but is similar.

! c) Give the rule for the "derivative" of a closure, i.e., $\frac{d(R^*)}{da}$.

d) Use the rules from (a)-(c) to find the "derivatives" of regular expression $(\mathbf{0+1})^*\mathbf{011}$ with respect to 0 and 1.

* e) Characterize those languages L for which $\frac{dL}{d0} = \emptyset$.

*! f) Characterize those languages L for which $\frac{dL}{d0} = L$.

! **Exercise 4.2.6:** Show that the regular languages are closed under the following operations:

a) $min(L) = \{w \mid w \text{ is in } L, \text{ but no proper prefix of } w \text{ is in } L\}$.

b) $max(L) = \{w \mid w \text{ is in } L \text{ and for no } x \text{ other than } \epsilon \text{ is } wx \text{ in } L\}$.

c) $init(L) = \{w \mid \text{for some } x, wx \text{ is in } L\}$.

Hint: Like Exercise 4.2.2, it is easiest to start with a DFA for L and perform a construction to get the desired language.

! **Exercise 4.2.7:** If $w = a_1a_2\cdots a_n$ and $x = b_1b_2\cdots b_m$ are strings of the same length, define $alt(w,x)$ to be the string in which the symbols of w and x alternate, starting with w, that is, $a_1b_1a_2b_2\cdots a_nb_n$. If L and M are languages, define $alt(L,M)$ to be the set of strings of the form $alt(w,x)$, where w is any string in L and x is any string in M of the same length. Prove that if L and M are regular, so is $alt(L,M)$.

*!! **Exercise 4.2.8:** Let L be a language. Define $half(L)$ to be the set of first halves of strings in L, that is, $\{w \mid \text{for some } x \text{ such that } |x| = |w|, \text{ we have } wx \text{ in } L\}$. For example, if $L = \{\epsilon, 0010, 011, 010110\}$ then $half(L) = \{\epsilon, 00, 010\}$. Notice that odd-length strings do not contribute to $half(L)$. Prove that if L is a regular language, so is $half(L)$.

!! **Exercise 4.2.9:** We can generalize Exercise 4.2.8 to a number of functions that determine how much of the string we take. If f is a function of integers, define $f(L)$ to be $\{w \mid \text{for some } x, \text{ with } |x| = f(|w|), \text{ we have } wx \text{ in } L\}$. For instance, the operation *half* corresponds to f being the identity function $f(n) = n$, since $half(L)$ is defined by having $|x| = |w|$. Show that if L is a regular language, then so is $f(L)$, if f is one of the following functions:

a) $f(n) = 2n$ (i.e., take the first thirds of strings).

b) $f(n) = n^2$ (i.e., the amount we take has length equal to the square root of what we do not take.

c) $f(n) = 2^n$ (i.e., what we take has length equal to the logarithm of what we leave).

!! **Exercise 4.2.10:** Suppose that L is any language, not necessarily regular, whose alphabet is $\{0\}$; i.e., the strings of L consist of 0's only. Prove that L^* is regular. *Hint*: At first, this theorem sounds preposterous. However, an example will help you see why it is true. Consider the language $L = \{0^i \mid i \text{ is prime}\}$, which we know is not regular by Example 4.3. Strings 00 and 000 are in L, since 2 and 3 are both primes. Thus, if $j \geq 2$, we can show 0^j is in L^*. If j is even, use $j/2$ copies of 00, and if j is odd, use one copy of 000 and $(j-3)/2$ copies of 00. Thus, $L^* = \mathbf{000}^*$.

!! **Exercise 4.2.11:** Show that the regular languages are closed under the following operation: $cycle(L) = \{w \mid$ we can write w as $w = xy$, such that yx is in $L\}$. For example, if $L = \{01, 011\}$, then $cycle(L) = \{01, 10, 011, 110, 101\}$. *Hint*: Start with a DFA for L and construct an ϵ-NFA for $cycle(L)$.

!! **Exercise 4.2.12:** Let $w_1 = a_0a_0a_1$, and $w_i = w_{i-1}w_{i-1}a_i$ for all $i > 1$. For instance, $w_3 = a_0a_0a_1a_0a_0a_1a_2a_0a_0a_1a_0a_0a_1a_2a_3$. The shortest regular expression for the language $L_n = \{w_n\}$, i.e., the language consisting of the one string w_n, is the string w_n itself, and the length of this expression is $2^{n+1} - 1$. However, if we allow the intersection operator, we can write an expression for L_n whose length is $O(n^2)$. Find such an expression. *Hint*: Find n languages, each with regular expressions of length $O(n)$, whose intersection is L_n.

! **Exercise 4.2.13:** We can use closure properties to help prove certain languages are not regular. Start with the fact that the language

$$L_{0n1n} = \{0^n1^n \mid n \geq 0\}$$

is not a regular set. Prove the following languages not to be regular by transforming them, using operations known to preserve regularity, to L_{0n1n}:

* a) $\{0^i1^j \mid i \neq j\}$.

b) $\{0^n1^m2^{n-m} \mid n \geq m \geq 0\}$.

Exercise 4.2.14: In Theorem 4.8, we described the "product construction" that took two DFA's and constructed one DFA whose language is the intersection of the languages of the first two.

a) Show how to perform the product construction on NFA's (without ϵ-transitions).

! b) Show how to perform the product construction on ϵ-NFA's.

* c) Show how to modify the product construction so the resulting DFA accepts the difference of the languages of the two given DFA's.

d) Show how to modify the product construction so the resulting DFA accepts the union of the languages of the two given DFA's.

Exercise 4.2.15: In the proof of Theorem 4.8 we claimed that it could be proved by induction on the length of w that

$$\hat{\delta}\big((q_L, q_M), w\big) = \big(\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w)\big)$$

Give this inductive proof.

Exercise 4.2.16: Complete the proof of Theorem 4.14 by considering the cases where expression E is a concatenation of two subexpressions and where E is the closure of an expression.

Exercise 4.2.17: In Theorem 4.16, we omitted a proof by induction on the length of w that $\hat{\gamma}(q_0, w) = \hat{\delta}\big(q_0, h(w)\big)$. Prove this statement.

4.4 Equivalence and Minimization of Automata

In contrast to the previous questions — emptiness and membership — whose algorithms were rather simple, the question of whether two descriptions of two regular languages actually define the same language involves considerable intellectual mechanics. In this section we discuss how to test whether two descriptors for regular languages are *equivalent*, in the sense that they define the same language. An important consequence of this test is that there is a way to minimize a DFA. That is, we can take any DFA and find an equivalent DFA that has the minimum number of states. In fact, this DFA is essentially unique: given any two minimum-state DFA's that are equivalent, we can always find a way to rename the states so that the two DFA's become the same.

4.4.1 Testing Equivalence of States

We shall begin by asking a question about the states of a single DFA. Our goal is to understand when two distinct states p and q can be replaced by a single state that behaves like both p and q. We say that states p and q are *equivalent* if:

- For all input strings w, $\hat{\delta}(p, w)$ is an accepting state if and only if $\hat{\delta}(q, w)$ is an accepting state.

Less formally, it is impossible to tell the difference between equivalent states p and q merely by starting in one of the states and asking whether or not a given input string leads to acceptance when the automaton is started in this (unknown) state. Note we do *not* require that $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ are the *same* state, only that either both are accepting or both are nonaccepting.

If two states are not equivalent, then we say they are *distinguishable*. That is, state p is distinguishable from state q if there is at least one string w such that one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting, and the other is not accepting.

Example 4.18: Consider the DFA of Fig. 4.8, whose transition function we shall refer to as δ in this example. Certain pairs of states are obviously not equivalent. For example, C and G are not equivalent because one is accepting and the other is not. That is, the empty string distinguishes these two states, because $\hat{\delta}(C, \epsilon)$ is accepting and $\hat{\delta}(G, \epsilon)$ is not.

Consider states A and G. String ϵ doesn't distinguish them, because they are both nonaccepting states. String 0 doesn't distinguish them because they go to

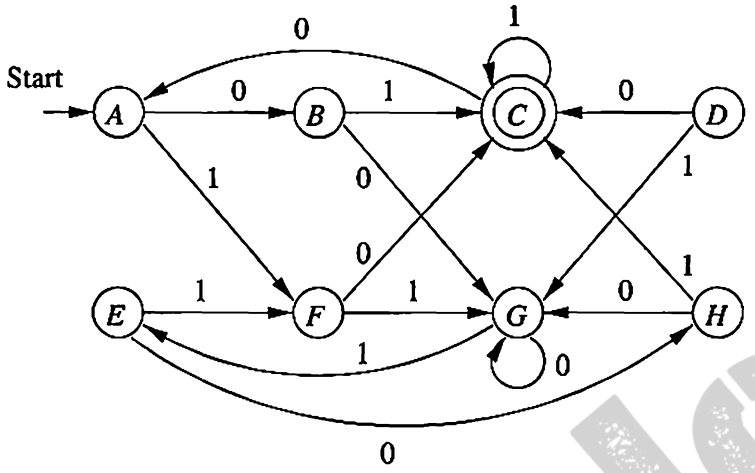


Figure 4.8: An automaton with equivalent states

states B and G, respectively on input 0, and both these states are nonaccepting. Likewise, string 1 doesn't distinguish A from G, because they go to F and E, respectively, and both are nonaccepting. However, 01 distinguishes A from G, because $\hat{\delta}(A, 01) = C$, $\hat{\delta}(G, 01) = E$, C is accepting, and E is not. Any input string that takes A and G to states only one of which is accepting is sufficient to prove that A and G are not equivalent.

In contrast, consider states A and E. Neither is accepting, so ϵ does not distinguish them. On input 1, they both go to state F. Thus, no input string that begins with 1 can distinguish A from E, since for any string x, $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x)$.

Now consider the behavior of states A and E on inputs that begin with 0. They go to states B and H, respectively. Since neither is accepting, string 0 by itself does not distinguish A from E. However, B and H are no help. On input 1 they both go to C, and on input 0 they both go to G. Thus, all inputs that begin with 0 will fail to distinguish A from E. We conclude that no input string whatsoever will distinguish A from E; i.e., they are equivalent states. □

To find states that are equivalent, we make our best efforts to find pairs of states that are distinguishable. It is perhaps surprising, but true, that if we try our best, according to the algorithm to be described below, then any pair of states that we do not find distinguishable are equivalent. The algorithm, which we refer to as the *table-filling algorithm*, is a recursive discovery of distinguishable pairs in a DFA $A = (Q, \Sigma, \delta, q_0, F)$.

BASIS: If p is an accepting state and q is nonaccepting, then the pair $\{p, q\}$ is distinguishable.

INDUCTION: Let p and q be states such that for some input symbol a, $r = \delta(p, a)$ and $s = \delta(q, a)$ are a pair of states known to be distinguishable. Then

$\{p, q\}$ is a pair of distinguishable states. The reason this rule makes sense is that there must be some string w that distinguishes r from s; that is, exactly one of $\hat{\delta}(r, w)$ and $\hat{\delta}(s, w)$ is accepting. Then string aw must distinguish p from q, since $\hat{\delta}(p, aw)$ and $\hat{\delta}(q, aw)$ is the same pair of states as $\hat{\delta}(r, w)$ and $\hat{\delta}(s, w)$.

Example 4.19: Let us execute the table-filling algorithm on the DFA of Fig 4.8. The final table is shown in Fig. 4.9, where an x indicates pairs of distinguishable states, and the blank squares indicate those pairs that have been found equivalent. Initially, there are no x's in the table.

B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x
	A	B	C	D	E	F	G

Figure 4.9: Table of state inequivalences

For the basis, since C is the only accepting state, we put x's in each pair that involves C. Now that we know some distinguishable pairs, we can discover others. For instance, since $\{C, H\}$ is distinguishable, and states E and F go to H and C, respectively, on input 0, we know that $\{E, F\}$ is also a distinguishable pair. In fact, all the x's in Fig. 4.9 with the exception of the pair $\{A, G\}$ can be discovered simply by looking at the transitions from the pair of states on either 0 or on 1, and observing that (for one of those inputs) one state goes to C and the other does not. We can show $\{A, G\}$ is distinguishable on the next round, since on input 1 they go to F and E, respectively, and we already established that the pair $\{E, F\}$ is distinguishable.

However, then we can discover no more distinguishable pairs. The three remaining pairs, which are therefore equivalent pairs, are $\{A, E\}$, $\{B, H\}$, and $\{D, F\}$. For example, consider why we can not infer that $\{A, E\}$ is a distinguishable pair. On input 0, A and E go to B and H, respectively, and $\{B, H\}$ has not yet been shown distinguishable. On input 1, A and E both go to F, so there is no hope of distinguishing them that way. The other two pairs, $\{B, H\}$ and $\{D, F\}$ will never be distinguished because they each have identical transitions on 0 and identical transitions on 1. Thus, the table-filling algorithm stops with the table as shown in Fig. 4.9, which is the correct determination of equivalent and distinguishable states. □

Theorem 4.20: If two states are not distinguished by the table-filling algorithm, then the states are equivalent.

PROOF: Let us again assume we are talking of the DFA $A = (Q, \Sigma, \delta, q_0, F)$. Suppose the theorem is false; that is, there is at least one pair of states $\{p, q\}$ such that

1. States p and q are distinguishable, in the sense that there is some string w such that exactly one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting, and yet
2. The table-filling algorithm does not find p and q to be distinguished.

Call such a pair of states a *bad pair*.

If there are bad pairs, then there must be some that are distinguished by the shortest strings among all those strings that distinguish bad pairs. Let $\{p, q\}$ be one such bad pair, and let $w = a_1 a_2 \cdots a_n$ be a string as short as any that distinguishes p from q. Then exactly one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting.

Observe first that w cannot be ϵ, since if ϵ distinguishes a pair of states, then that pair is marked by the basis part of the table-filling algorithm. Thus, $n \geq 1$.

Consider the states $r = \delta(p, a_1)$ and $s = \delta(q, a_1)$. States r and s are distinguished by the string $a_2 a_3 \cdots a_n$, since this string takes r and s to the states $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$. However, the string distinguishing r from s is shorter than any string that distinguishes a bad pair. Thus, $\{r, s\}$ cannot be a bad pair. Rather, the table-filling algorithm must have discovered that they are distinguishable.

But the inductive part of the table-filling algorithm will not stop until it has also inferred that p and q are distinguishable, since it finds that $\delta(p, a_1) = r$ is distinguishable from $\delta(q, a_1) = s$. We have contradicted our assumption that bad pairs exist. If there are no bad pairs, then every pair of distinguishable states is distinguished by the table-filling algorithm, and the theorem is true. □

4.4.2 Testing Equivalence of Regular Languages

The table-filling algorithm gives us an easy way to test if two regular languages are the same. Suppose languages L and M are each represented in some way, e.g., one by a regular expression and one by an NFA. Convert each representation to a DFA. Now, imagine one DFA whose states are the union of the states of the DFA's for L and M. Technically, this DFA has two start states, but actually the start state is irrelevant as far as testing state equivalence is concerned, so make any state the lone start state.

Now, test if the start states of the two original DFA's are equivalent, using the table-filling algorithm. If they are equivalent, then $L = M$, and if not, then $L \neq M$.

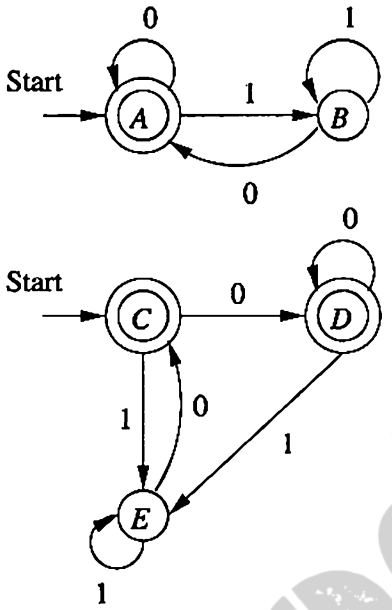


Figure 4.10: Two equivalent DFA's

Example 4.21: Consider the two DFA's in Fig. 4.10. Each DFA accepts the empty string and all strings that end in 0; that is the language of regular expression $\epsilon + (0 + 1)^*0$. We can imagine that Fig. 4.10 represents a single DFA, with five states A through E. If we apply the table-filling algorithm to that automaton, the result is as shown in Fig. 4.11.

B	x			
C		x		
D		x		
E	x		x	x
	A	B	C	D

Figure 4.11: The table of distinguishabilities for Fig. 4.10

To see how the table is filled out, we start by placing x's in all pairs of states where exactly one of the states is accepting. It turns out that there is no more to do. The four remaining pairs, $\{A, C\}$, $\{A, D\}$, $\{C, D\}$, and $\{B, E\}$ are all equivalent pairs. You should check that no more distinguishable pairs are discovered in the inductive part of the table-filling algorithm. For instance, with the table as in Fig. 4.11, we cannot distinguish the pair $\{A, D\}$ because on 0 they go to themselves, and on 1 they go to the pair $\{B, E\}$, which has

not yet been distinguished. Since A and C are found equivalent by this test, and those states were the start states of the two original automata, we conclude that these DFA's do accept the same language. □

The time to fill out the table, and thus to decide whether two states are equivalent is polynomial in the number of states. If there are n states, then there are $\binom{n}{2}$, or $n(n-1)/2$ pairs of states. In one round, we consider all pairs of states, to see if one of their successor pairs has been found distinguishable, so a round surely takes no more than $O(n^2)$ time. Moreover, if on some round, no additional x's are placed in the table, then the algorithm ends. Thus, there can be no more than $O(n^2)$ rounds, and $O(n^4)$ is surely an upper bound on the running time of the table-filling algorithm.

However, a more careful algorithm can fill the table in $O(n^2)$ time. The idea is to initialize, for each pair of states $\{r,s\}$, a list of those pairs $\{p,q\}$ that "depend on" $\{r,s\}$. That is, if $\{r,s\}$ is found distinguishable, then $\{p,q\}$ is distinguishable. We create the lists initially by examining each pair of states $\{p,q\}$, and for each of the fixed number of input symbols a, we put $\{p,q\}$ on the list for the pair of states $\{\delta(p,a),\delta(q,a)\}$, which are the successor states for p and q on input a.

If we ever find $\{r,s\}$ to be distinguishable, then we go down the list for $\{r,s\}$. For each pair on that list that is not already distinguishable, we make that pair distinguishable, and we put the pair on a queue of pairs whose lists we must check similarly.

The total work of this algorithm is proportional to the sum of the lengths of the lists, since we are at all times either adding something to the lists (initialization) or examining a member of the list for the first and last time (when we go down the list for a pair that has been found distinguishable). Since the size of the input alphabet is considered a constant, each pair of states is put on $O(1)$ lists. As there are $O(n^2)$ pairs, the total work is $O(n^2)$.

4.4.3 Minimization of DFA's

Another important consequence of the test for equivalence of states is that we can "minimize" DFA's. That is, for each DFA we can find an equivalent DFA that has as few states as any DFA accepting the same language. Moreover, except for our ability to call the states by whatever names we choose, this minimum-state DFA is unique for the language. The algorithm is as follows:

1. First, eliminate any state that cannot be reached from the start state.
2. Then, partition the remaining states into blocks, so that all states in the same block are equivalent, and no pair of states from different blocks are equivalent. Theorem 4.24, below, shows that we can always make such a partition.

Example 4.22: Consider the table of Fig. 4.9, where we determined the state equivalences and distinguishabilities for the states of Fig. 4.8. The partition

of the states into equivalent blocks is $(\{A,E\}, \{B,H\}, \{C\}, \{D,F\}, \{G\})$. Notice that the three pairs of states that are equivalent are each placed in a block together, while the states that are distinguishable from all the other states are each in a block alone.

For the automaton of Fig. 4.10, the partition is $(\{A,C,D\}, \{B,E\})$. This example shows that we can have more than two states in a block. It may appear fortuitous that A, C, and D can all live together in a block, because every pair of them is equivalent, and none of them is equivalent to any other state. However, as we shall see in the next theorem to be proved, this situation is guaranteed by our definition of "equivalence" for states. □

Theorem 4.23: The equivalence of states is transitive. That is, if in some DFA $A = (Q, \Sigma, \delta, q_0, F)$ we find that states p and q are equivalent, and we also find that q and r are equivalent, then it must be that p and r are equivalent.

PROOF: Note that transitivity is a property we expect of any relationship called "equivalence." However, simply calling something "equivalence" doesn't make it transitive; we must prove that the name is justified.

Suppose that the pairs $\{p,q\}$ and $\{q,r\}$ are equivalent, but pair $\{p,r\}$ is distinguishable. Then there is some input string w such that exactly one of $\hat{\delta}(p,w)$ and $\hat{\delta}(r,w)$ is an accepting state. Suppose, by symmetry, that $\hat{\delta}(p,w)$ is the accepting state.

Now consider whether $\hat{\delta}(q,w)$ is accepting or not. If it is accepting, then $\{q,r\}$ is distinguishable, since $\hat{\delta}(q,w)$ is accepting, and $\hat{\delta}(r,w)$ is not. If $\hat{\delta}(q,w)$ is nonaccepting, then $\{p,q\}$ is distinguishable for a similar reason. We conclude by contradiction that $\{p,r\}$ was not distinguishable, and therefore this pair is equivalent. □

We can use Theorem 4.23 to justify the obvious algorithm for partitioning states. For each state q, construct a block that consists of q and all the states that are equivalent to q. We must show that the resulting blocks are a partition; that is, no state is in two distinct blocks.

First, observe that all states in any block are mutually equivalent. That is, if p and r are two states in the block of states equivalent to q, then p and r are equivalent to each other, by Theorem 4.23.

Suppose that there are two overlapping, but not identical blocks. That is, there is a block B that includes states p and q, and another block C that includes p but not q. Since p and q are in a block together, they are equivalent. Consider how the block C was formed. If it was the block generated by p, then q would be in C, because those states are equivalent. Thus, it must be that there is some third state s that generated block C; i.e., C is the set of states equivalent to s.

We know that p is equivalent to s, because p is in block C. We also know that p is equivalent to q because they are together in block B. By the transitivity of Theorem 4.23, q is equivalent to s. But then q belongs in block C, a contradiction. We conclude that equivalence of states partitions the states; that is, two

states either have the same set of equivalent states (including themselves), or their equivalent states are disjoint. To conclude the above analysis:

Theorem 4.24: If we create for each state q of a DFA a *block* consisting of q and all the states equivalent to q, then the different blocks of states form a *partition* of the set of states.[5] That is, each state is in exactly one block. All members of a block are equivalent, and no pair of states chosen from different blocks are equivalent. □

We are now able to state succinctly the algorithm for minimizing a DFA $A = (Q, \Sigma, \delta, q_0, F)$.

1. Use the table-filling algorithm to find all the pairs of equivalent states.
2. Partition the set of states Q into blocks of mutually equivalent states by the method described above.
3. Construct the minimum-state equivalent DFA B by using the blocks as its states. Let γ be the transition function of B. Suppose S is a set of equivalent states of A, and a is an input symbol. Then there must exist one block T of states such that for all states q in S, $\delta(q, a)$ is a member of block T. For if not, then input symbol a takes two states p and q of S to states in different blocks, and those states are distinguishable by Theorem 4.24. That fact lets us conclude that p and q are not equivalent, and they did not both belong in S. As a consequence, we can let $\gamma(S, a) = T$. In addition:
   (a) The start state of B is the block containing the start state of A.
   (b) The set of accepting states of B is the set of blocks containing accepting states of A. Note that if one state of a block is accepting, then all the states of that block must be accepting. The reason is that any accepting state is distinguishable from any nonaccepting state, so you can't have both accepting and nonaccepting states in one block of equivalent states.

Example 4.25: Let us minimize the DFA from Fig. 4.8. We established the blocks of the state partition in Example 4.22. Figure 4.12 shows the minimum-state automaton. Its five states correspond to the five blocks of equivalent states for the automaton of Fig. 4.8.

The start state is $\{A, E\}$, since A was the start state of Fig. 4.8. The only accepting state is $\{C\}$, since C is the only accepting state of Fig. 4.8. Notice that the transitions of Fig. 4.12 properly reflect the transitions of Fig. 4.8. For instance, Fig. 4.12 has a transition on input 0 from $\{A, E\}$ to $\{B, H\}$. That

[5] You should remember that the same block may be formed several times, starting from different states. However, the partition consists of the *different* blocks, so this block appears only once in the partition.

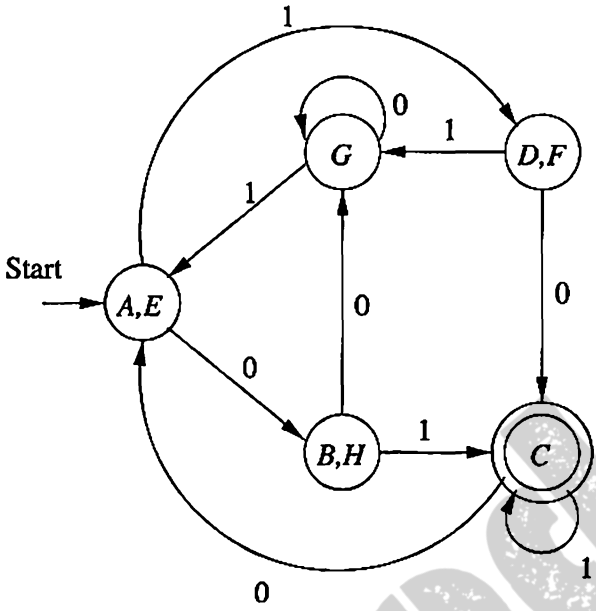


Figure 4.12: Minimum-state DFA equivalent to Fig. 4.8

makes sense, because in Fig. 4.8, A goes to B on input 0, and E goes to H. Likewise, on input 1, $\{A, E\}$ goes to $\{D, F\}$. If we examine Fig. 4.8, we find that both A and E go to F on input 1, so the selection of the successor of $\{A, E\}$ on input 1 is also correct. Note that the fact neither A nor E goes to D on input 1 is not important. You may check that all of the other transitions are also proper. □

4.4.4 Why the Minimized DFA Can't Be Beaten

Suppose we have a DFA A, and we minimize it to construct a DFA M, using the partitioning method of Theorem 4.24. That theorem shows that we can't group the states of A into fewer groups and still have an equivalent DFA. However, could there be another DFA N, unrelated to A, that accepts the same language as A and M, yet has fewer states than M? We can prove by contradiction that N does not exist.

First, run the state-distinguishability process of Section 4.4.1 on the states of M and N together, as if they were one DFA. We may assume that the states of M and N have no names in common, so the transition function of the combined automaton is the union of the transition rules of M and N, with no interaction. States are accepting in the combined DFA if and only if they are accepting in the DFA from which they come.

The start states of M and N are indistinguishable because $L(M) = L(N)$. Further, if $\{p, q\}$ are indistinguishable, then their successors on any one input

Minimizing the States of an NFA

You might imagine that the same state-partition technique that minimizes the states of a DFA could also be used to find a minimum-state NFA equivalent to a given NFA or DFA. While we can, by a process of exhaustive enumeration, find an NFA with as few states as possible accepting a given regular language, we cannot simply group the states of some given NFA for the language.

An example is in Fig. 4.13. None of the three states are equivalent. Surely accepting state B is distinguishable from nonaccepting states A and C. However, A and C are distinguishable by input 0. The successors of C are A alone, which does not include an accepting state, while the successors of A are $\{A, B\}$, which does include an accepting state. Thus, grouping equivalent states does not reduce the number of states of Fig. 4.13.

However, we can find a smaller NFA for the same language if we simply remove state C. Note that A and B alone accept all strings ending in 0, while adding state C does not allow us to accept any other strings.

symbol are also indistinguishable. The reason is that if we could distinguish the successors, then we could distinguish p from q.

Neither M nor N could have an inaccessible state, or else we could eliminate that state and have an even smaller DFA for the same language. Thus, every state of M is indistinguishable from at least one state of N. To see why, suppose p is a state of M. Then there is some string $a_1 a_2 \cdots a_k$ that takes the start state of M to state p. This string also takes the start state of N to some state q. Since we know the start states are indistinguishable, we also know that their successors under input symbol a_1 are indistinguishable. Then, the successors of those states on input a_2 are indistinguishable, and so on, until we conclude

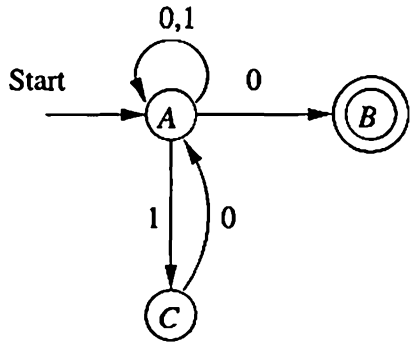


Figure 4.13: An NFA that cannot be minimized by state equivalence

that p and q are indistinguishable.

Since N has fewer states than M, there are two states of M that are indistinguishable from the same state of N, and therefore indistinguishable from each other. But M was designed so that all its states *are* distinguishable from each other. We have a contradiction, so the assumption that N exists is wrong, and M in fact has as few states as any equivalent DFA for A. Formally, we have proved:

Theorem 4.26: If A is a DFA, and M the DFA constructed from A by the algorithm described in the statement of Theorem 4.24, then M has as few states as any DFA equivalent to A. □

In fact we can say something even stronger than Theorem 4.26. There must be a one-to-one correspondence between the states of any other minimum-state N and the DFA M. The reason is that we argued above how each state of M must be equivalent to one state of N, and no state of M can be equivalent to two states of N. We can similarly argue that no state of N can be equivalent to two states of M, although each state of N must be equivalent to one of M's states. Thus, the minimum-state DFA equivalent to A is unique except for a possible renaming of the states.

	0	1
$\rightarrow A$	B	A
B	A	C
C	D	B
$*D$	D	A
E	D	F
F	G	E
G	F	G
H	G	D

Figure 4.14: A DFA to be minimized

4.4.5 Exercises for Section 4.4

* **Exercise 4.4.1:** In Fig. 4.14 is the transition table of a DFA.

a) Draw the table of distinguishabilities for this automaton.

b) Construct the minimum-state equivalent DFA.

Exercise 4.4.2: Repeat Exercise 4.4.1 for the DFA of Fig 4.15.

!! **Exercise 4.4.3:** Suppose that p are q are distinguishable states of a given DFA A with n states. As a function of n, what is the tightest upper bound on how long the shortest string that distinguishes p from q can be?

	0	1
$\rightarrow A$	B	E
B	C	F
$*C$	D	H
D	E	H
E	F	I
$*F$	G	B
G	H	B
H	I	C
$*I$	A	E

Figure 4.15: Another DFA to minimize