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

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

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

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

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

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

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

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

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

3.2 Finite Automata and Regular Expressions

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

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

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

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

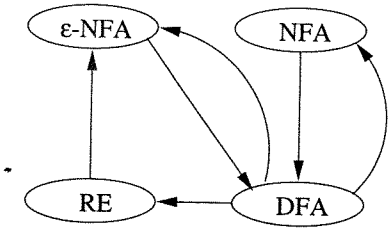

Figure 3.1: Plan for showing the equivalence of four different notations for regular languages

3.2.1 From DFA's to Regular Expressions

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

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

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

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

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

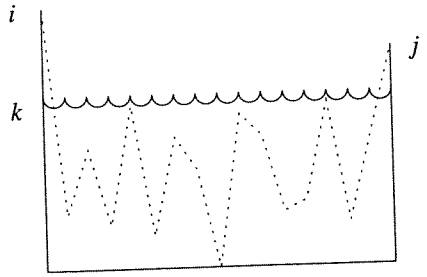

Figure 3.2: A path whose label is in the language of regular expression $R_{ij}^{(k)}$

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.

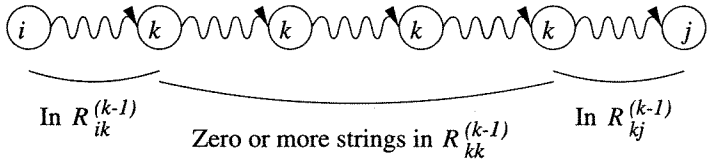

Figure 3.3: A path from i to j can be broken into segments at each point where it goes through state k

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.

This expression is $\mathbf{1^*0(0+1)^*}$. 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,

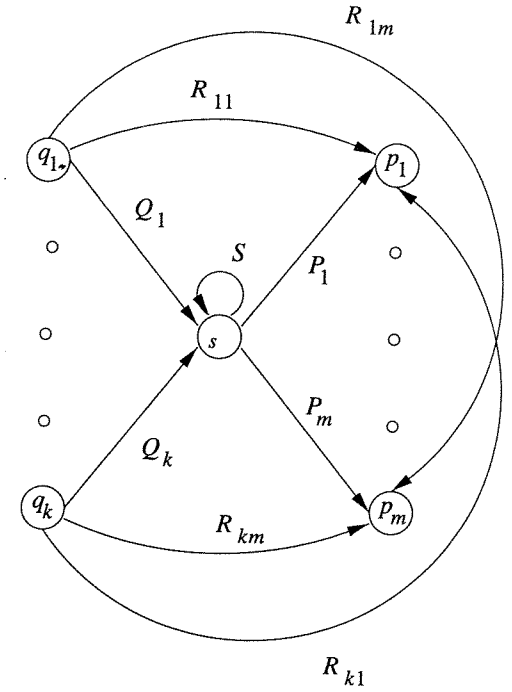

Figure 3.7: A state s about to be eliminated

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 $Q_iS^*P_j$. 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

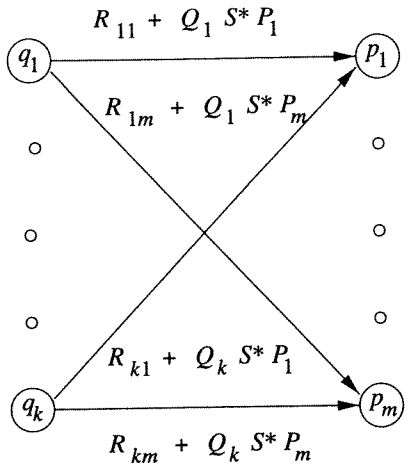

Figure 3.8: Result of eliminating state s from Fig. 3.7

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 $L(U)$.

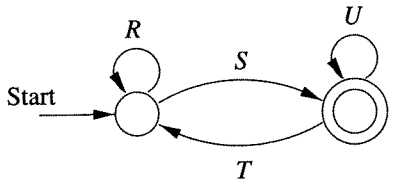

Figure 3.9: A generic two-state automaton

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^*.

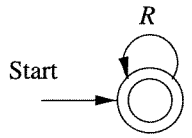

Figure 3.10: A generic one-state automaton

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).

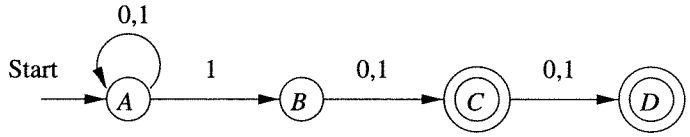

Figure 3.11: An NFA accepting strings that have a 1 either two or three positions from the end

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.

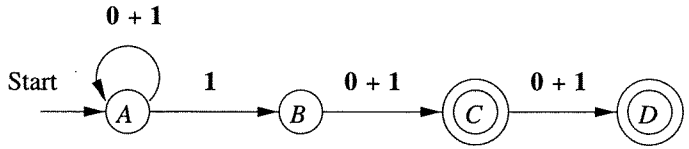

Figure 3.12: The automaton of Fig. 3.11 with regular-expression labels

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

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

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

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

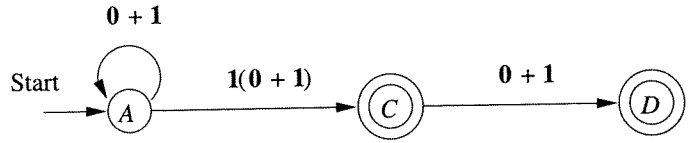

Figure 3.13: Eliminating state B

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.

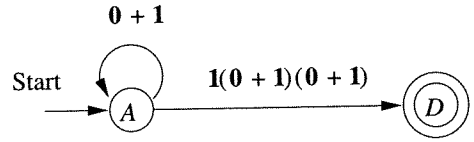

Figure 3.14: A two-state automaton with states A and D

In terms of the generic two-state automaton of Fig. 3.9, the regular expressions from Fig. 3.14 are: $R = \mathbf{0}+\mathbf{1}$, $S = \mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$, $T = \emptyset$, and $U = \emptyset$. The expression U^* can be replaced by ϵ, i.e., eliminated in a concatenation; the justification is that $\emptyset^* = \epsilon$, as we discussed above. Also, the expression SU^*T is equivalent to $\emptyset$, since T, one of the terms of the concatenation, is $\emptyset$. The generic expression $(R + SU^*T)^*SU^*$ thus simplifies in this case to R^*S, or $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$. In informal terms, the language of this expression is any string ending in 1, followed by two symbols that are each either 0 or 1. That language is one portion of the strings accepted by the automaton of Fig. 3.11: those strings whose third position from the end has a 1.

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

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.

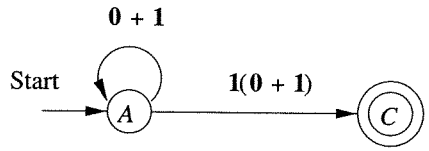

Figure 3.15: Two-state automaton resulting from the elimination of D

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 $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$. This expression represents the other type of string the automaton accepts: those with a 1 in the second position from the end.

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

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

□

3.2.3 Converting Regular Expressions to Automata

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

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

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.

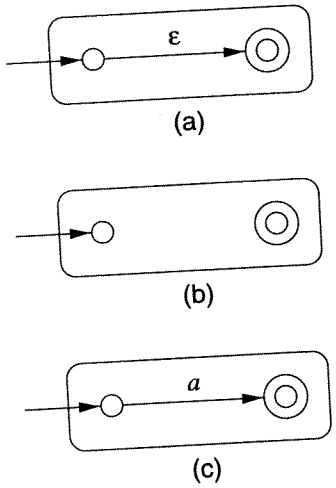

Figure 3.16: The basis of the construction of an automaton from a regular expression

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

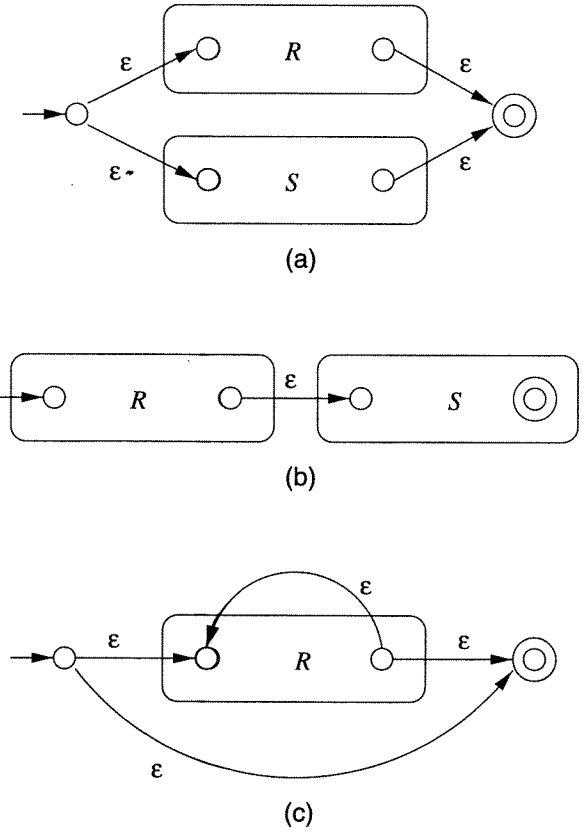

Figure 3.17: The inductive step **in** the regular-expression-to-ϵ-NFA construction

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 expressio**n;** 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 $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})$ to an ϵ-NFA. Our first step is to construct an automaton for $\mathbf{0} + \mathbf{1}$. We use two automata constructed according to Fig. 3.16(c), one with label **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 $\mathbf{0} + \mathbf{1}$. Again, we must create a copy of the automaton of Fig. 3.18(a); we must not use the same copy that became part of Fig. 3.18(b). The complete automaton is shown in

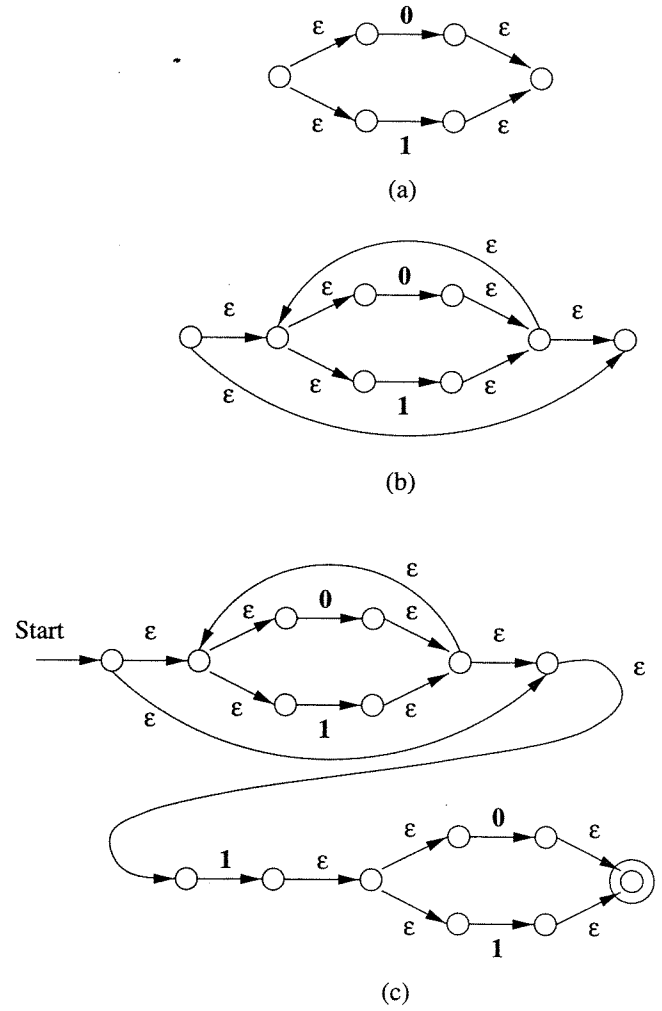

Figure 3.18: Automata constructed for Example 3.8

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) $\mathbf{01}^*$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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