must create a copy of the automaton of Fig. 3.18(a); we must not use the same copy that became part of Fig. 3.18(b). The complete automaton is shown in Fig. 3.18(c). Note that this ε-NFA, when ε-transitions are removed, looks just like the much simpler automaton of Fig. 3.15 that also accepts the strings that have a 1 in their next-to-last position. □

3.2.4 Exercises for Section 3.2

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>→q₁</td>
<td>q₂</td>
<td>q₁</td>
</tr>
<tr>
<td>q₂</td>
<td>q₃</td>
<td>q₁</td>
</tr>
<tr>
<td>*q₃</td>
<td>q₃</td>
<td>q₂</td>
</tr>
</tbody>
</table>

* 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:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>→q₁</td>
<td>q₂</td>
<td>q₃</td>
</tr>
<tr>
<td>q₂</td>
<td>q₃</td>
<td>q₁</td>
</tr>
<tr>
<td>*q₃</td>
<td>q₂</td>
<td>q₁</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>→*p</td>
<td>s</td>
<td>p</td>
</tr>
<tr>
<td>q</td>
<td>p</td>
<td>s</td>
</tr>
<tr>
<td>r</td>
<td>r</td>
<td>q</td>
</tr>
<tr>
<td>s</td>
<td>q</td>
<td>r</td>
</tr>
</tbody>
</table>
Exercise 3.2.4: Convert the following regular expressions to NFA’s with \( \epsilon \)-transitions.

* a) \( 01^* \).

b) \((0 + 1)01\).

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

Exercise 3.2.5: Eliminate \( \epsilon \)-transitions from your \( \epsilon \)-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 \( \epsilon \)-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 \( \epsilon \)-transition from \( q_f \) to \( q_0 \).

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

c) The automaton constructed from \( A \) by adding an \( \epsilon \)-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 \( \epsilon \)-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 \( \epsilon \)-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 \( \epsilon \)-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 \( A \). \textit{Hint:} Use the technique suggested by the construction of Theorem 3.4.
regular expressions denote the same language until Section 4.4. However, we can use ad-hoc means to decide the equality of the pairs of languages that we actually care about. Recall that if the languages are not the same, then it is sufficient to provide one counterexample: a single string that is in one language but not the other.

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

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

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

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

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

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

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

### 3.4.8 Exercises for Section 3.4

**Exercise 3.4.1:** Verify the following identities involving regular expressions.

* a) \( R + S = S + R \).

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

  c) \( (RS)T = R(ST) \).
Extensions of the Test Beyond Regular Expressions May Fail

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

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

d) \( R(S + T) = RS + RT \).
e) \( (R + S)T = RT + ST \).

\* f) \( (R^*)^* = R^* \).
g) \( (e + R)^* = R^* \).
h) \( (R^* S^*)^* = (R + S)^* \).

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

\* a) \( (R + S)^* = R^* + S^* \).
b) \( (RS + R)^* R = R(SR + R)^* \).

\* c) \( (RS + R)^* RS = (RR^* S)^* \).
d) \( (R + S)^* S = (R^* S)^* \).
e) \( S(RS + S)^* R = RR^* S(RR^* S)^* \).

Exercise 3.4.3: In Example 3.6, we developed the regular expression
\[
(0 + 1)^* 1(0 + 1) + (0 + 1)^* 1(0 + 1)(0 + 1)
\]
Use the distributive laws to develop two different, simpler, equivalent expressions.