4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon. \end{cases}$$

1.3

REGULAR EXPRESSIONS

In arithmetic, we can use the operations $+$ and $\times$ to build up expressions such as

$$(5+3) \times 4\,.$$

Similarly, we can use the regular operations to build up expressions describing languages, which are called ***regular expressions***. An example is:

$$(\mathtt{0} \cup \mathtt{1})\mathtt{0}^*.$$

The value of the arithmetic expression is the number 32. The value of a regular expression is a language. In this case, the value is the language consisting of all strings starting with a 0 or a 1 followed by any number of 0s. We get this result by dissecting the expression into its parts. First, the symbols 0 and 1 are shorthand for the sets $\{\mathtt{0}\}$ and $\{\mathtt{1}\}$. So $(\mathtt{0} \cup \mathtt{1})$ means $(\{\mathtt{0}\} \cup \{\mathtt{1}\})$. The value of this part is the language $\{\mathtt{0},\mathtt{1}\}$. The part $\mathtt{0}^*$ means $\{\mathtt{0}\}^*$, and its value is the language consisting of all strings containing any number of 0s. Second, like the $\times$ symbol in algebra, the concatenation symbol $\circ$ often is implicit in regular expressions. Thus $(\mathtt{0} \cup \mathtt{1})\mathtt{0}^*$ actually is shorthand for $(\mathtt{0} \cup \mathtt{1}) \circ \mathtt{0}^*$. The concatenation attaches the strings from the two parts to obtain the value of the entire expression.

Regular expressions have an important role in computer science applications. In applications involving text, users may want to search for strings that satisfy certain patterns. Regular expressions provide a powerful method for describing such patterns. Utilities such as awk and grep in UNIX, modern programming languages such as Perl, and text editors all provide mechanisms for the description of patterns by using regular expressions.

EXAMPLE 1.51

Another example of a regular expression is

$$(\mathtt{0} \cup \mathtt{1})^*.$$

It starts with the language $(\mathtt{0} \cup \mathtt{1})$ and applies the $*$ operation. The value of this expression is the language consisting of all possible strings of 0s and 1s. If $\Sigma = \{\mathtt{0},\mathtt{1}\}$, we can write Σ as shorthand for the regular expression $(\mathtt{0} \cup \mathtt{1})$. More generally, if Σ is any alphabet, the regular expression Σ describes the language consisting of all strings of length 1 over this alphabet, and Σ^* describes the language consisting of all strings over that alphabet. Similarly, $\Sigma^*\mathtt{1}$ is the language that contains all strings that end in a 1. The language $(\mathtt{0}\Sigma^*) \cup (\Sigma^*\mathtt{1})$ consists of all strings that start with a 0 or end with a 1. ■

In arithmetic, we say that $\times$ has precedence over $+$ to mean that when there is a choice, we do the $\times$ operation first. Thus in $2+3\times4$, the 3×4 is done before the addition. To have the addition done first, we must add parentheses to obtain $(2+3)\times4$. In regular expressions, the star operation is done first, followed by concatenation, and finally union, unless parentheses change the usual order.

FORMAL DEFINITION OF A REGULAR EXPRESSION

DEFINITION 1.52

Say that R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

In items 1 and 2, the regular expressions a and ε represent the languages $\{a\}$ and $\{\varepsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the star of the language R_1, respectively.

Don't confuse the regular expressions ε and $\emptyset$. The expression ε represents the language containing a single string—namely, the empty string—whereas $\emptyset$ represents the language that doesn't contain any strings.

Seemingly, we are in danger of defining the notion of a regular expression in terms of itself. If true, we would have a ***circular definition***, which would be invalid. However, R_1 and R_2 always are smaller than R. Thus we actually are defining regular expressions in terms of smaller regular expressions and thereby avoiding circularity. A definition of this type is called an ***inductive definition***.

Parentheses in an expression may be omitted. If they are, evaluation is done in the precedence order: star, then concatenation, then union.

For convenience, we let R^+ be shorthand for RR^*. In other words, whereas R^* has all strings that are 0 or more concatenations of strings from R, the language R^+ has all strings that are 1 or more concatenations of strings from R. So $R^+ \cup \varepsilon = R^*$. In addition, we let R^k be shorthand for the concatenation of k R's with each other.

When we want to distinguish between a regular expression R and the language that it describes, we write $L(R)$ to be the language of R.

EXAMPLE 1.53

In the following instances, we assume that the alphabet Σ is $\{\mathtt{0},\mathtt{1}\}$.

1. $\mathtt{0}^*\mathtt{1}\mathtt{0}^* = \{w \mid w \text{ contains a single } \mathtt{1}\}$.
2. $\Sigma^*\mathtt{1}\Sigma^* = \{w \mid w \text{ has at least one } \mathtt{1}\}$.
3. $\Sigma^*\mathtt{001}\Sigma^* = \{w \mid w \text{ contains the string } \mathtt{001} \text{ as a substring}\}$.
4. $\mathtt{1}^*(\mathtt{01}^+)^* = \{w \mid \text{every } \mathtt{0} \text{ in } w \text{ is followed by at least one } \mathtt{1}\}$.
5. $(\Sigma\Sigma)^* = \{w \mid w \text{ is a string of even length}\}$.[5]
6. $(\Sigma\Sigma\Sigma)^* = \{w \mid \text{the length of } w \text{ is a multiple of } 3\}$.
7. $\mathtt{01} \cup \mathtt{10} = \{\mathtt{01}, \mathtt{10}\}$.
8. $\mathtt{0}\Sigma^*\mathtt{0} \cup \mathtt{1}\Sigma^*\mathtt{1} \cup \mathtt{0} \cup \mathtt{1} = \{w \mid w \text{ starts and ends with the same symbol}\}$.
9. $(\mathtt{0} \cup \varepsilon)\mathtt{1}^* = \mathtt{01}^* \cup \mathtt{1}^*$.
   The expression $\mathtt{0} \cup \varepsilon$ describes the language $\{\mathtt{0}, \varepsilon\}$, so the concatenation operation adds either $\mathtt{0}$ or ε before every string in $\mathtt{1}^*$.
10. $(\mathtt{0} \cup \varepsilon)(\mathtt{1} \cup \varepsilon) = \{\varepsilon, \mathtt{0}, \mathtt{1}, \mathtt{01}\}$.
11. $\mathtt{1}^*\emptyset = \emptyset$.
    Concatenating the empty set to any set yields the empty set.
12. $\emptyset^* = \{\varepsilon\}$.
    The star operation puts together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, giving only the empty string.

■

[5]The ***length*** of a string is the number of symbols that it contains.

If we let R be any regular expression, we have the following identities. They are good tests of whether you understand the definition.

$R \cup \emptyset = R$.
Adding the empty language to any other language will not change it.

$R \circ \varepsilon = R$.
Joining the empty string to any string will not change it.

However, exchanging $\emptyset$ and ε in the preceding identities may cause the equalities to fail.

$R \cup \varepsilon$ may not equal R.
For example, if $R = \mathtt{0}$, then $L(R) = \{\mathtt{0}\}$ but $L(R \cup \varepsilon) = \{\mathtt{0}, \varepsilon\}$.

$R \circ \emptyset$ may not equal R.
For example, if $R = \mathtt{0}$, then $L(R) = \{\mathtt{0}\}$ but $L(R \circ \emptyset) = \emptyset$.

Regular expressions are useful tools in the design of compilers for programming languages. Elemental objects in a programming language, called ***tokens***, such as the variable names and constants, may be described with regular expressions. For example, a numerical constant that may include a fractional part and/or a sign may be described as a member of the language

$$\big(\mathtt{+} \cup \mathtt{-} \cup \varepsilon\big)\,\big(D^{\mathtt{+}} \cup D^{\mathtt{+}}\,\mathtt{.}\,D^{*} \cup D^{*}\,\mathtt{.}\,D^{\mathtt{+}}\big)$$

where $D = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}\}$ is the alphabet of decimal digits. Examples of generated strings are: `72`, `3.14159`, `+7.`, and `-.01`.

Once the syntax of a programming language has been described with a regular expression in terms of its tokens, automatic systems can generate the ***lexical analyzer***, the part of a compiler that initially processes the input program.

EQUIVALENCE WITH FINITE AUTOMATA

Regular expressions and finite automata are equivalent in their descriptive power. This fact is surprising because finite automata and regular expressions superficially appear to be rather different. However, any regular expression can be converted into a finite automaton that recognizes the language it describes, and vice versa. Recall that a regular language is one that is recognized by some finite automaton.

THEOREM 1.54

A language is regular if and only if some regular expression describes it.

This theorem has two directions. We state and prove each direction as a separate lemma.

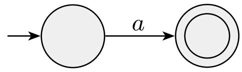
LEMMA 1.55 ....

If a language is described by a regular expression, then it is regular.

PROOF IDEA Say that we have a regular expression R describing some language A. We show how to convert R into an NFA recognizing A. By Corollary 1.40, if an NFA recognizes A then A is regular.

PROOF Let's convert R into an NFA N. We consider the six cases in the formal definition of regular expressions.

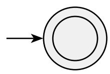
1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.



Note that this machine fits the definition of an NFA but not that of a DFA because it has some states with no exiting arrow for each possible input symbol. Of course, we could have presented an equivalent DFA here; but an NFA is all we need for now, and it is easier to describe.

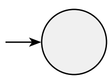
Formally, $N = \big(\{q_1, q_2\},\ \Sigma,\ \delta,\ q_1,\ \{q_2\}\big)$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.

2. $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$, and the following NFA recognizes $L(R)$.



Formally, $N = \big(\{q_1\}, \Sigma, \delta, q_1, \{q_1\}\big)$, where $\delta(r, b) = \emptyset$ for any r and b.

3. $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$.



Formally, $N = \big(\{q\}, \Sigma, \delta, q, \emptyset\big)$, where $\delta(r, b) = \emptyset$ for any r and b.

4. $R = R_1 \cup R_2$.
5. $R = R_1 \circ R_2$.
6. $R = R_1^*$.

For the last three cases, we use the constructions given in the proofs that the class of regular languages is closed under the regular operations. In other words, we construct the NFA for R from the NFAs for R_1 and R_2 (or just R_1 in case 6) and the appropriate closure construction.

....

That ends the first part of the proof of Theorem 1.54, giving the easier direction of the if and only if condition. Before going on to the other direction, let's consider some examples whereby we use this procedure to convert a regular expression to an NFA.

EXAMPLE 1.56

We convert the regular expression $(\mathtt{ab} \cup \mathtt{a})^*$ to an NFA in a sequence of stages. We build up from the smallest subexpressions to larger subexpressions until we have an NFA for the original expression, as shown in the following diagram. Note that this procedure generally doesn't give the NFA with the fewest states. In this example, the procedure gives an NFA with eight states, but the smallest equivalent NFA has only two states. Can you find it?

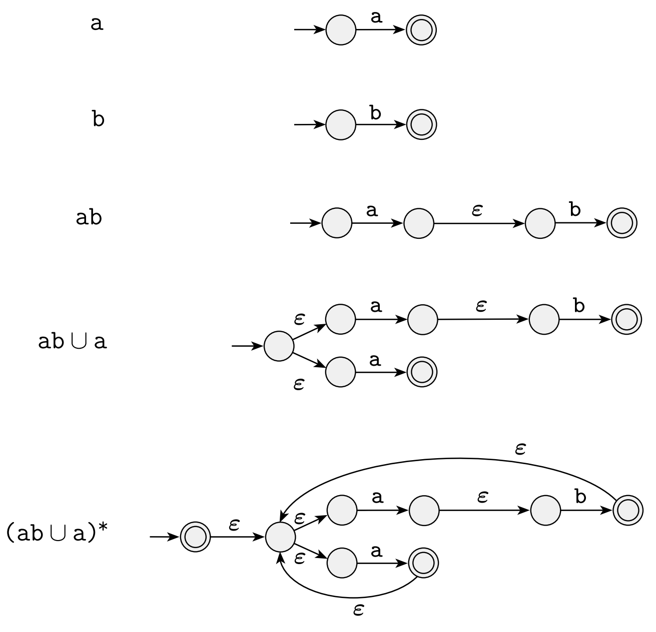


FIGURE 1.57
Building an NFA from the regular expression $(\mathtt{ab} \cup \mathtt{a})^*$

■

EXAMPLE 1.58

In Figure 1.59, we convert the regular expression $(\mathtt{a} \cup \mathtt{b})^*\mathtt{aba}$ to an NFA. A few of the minor steps are not shown.

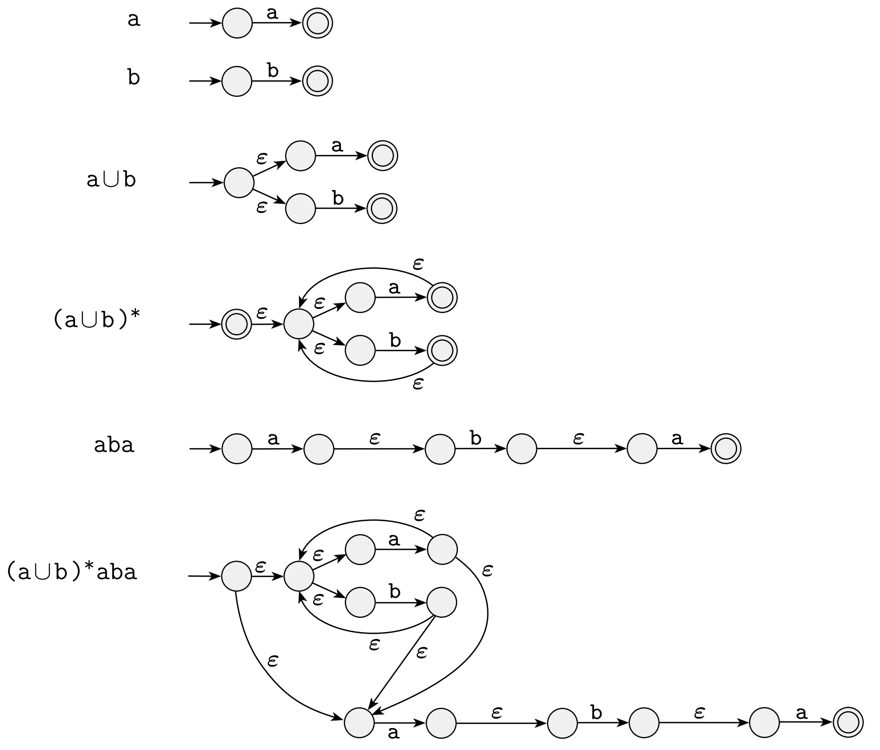


FIGURE 1.59
Building an NFA from the regular expression $(\mathtt{a} \cup \mathtt{b})^*\mathtt{aba}$

Now let's turn to the other direction of the proof of Theorem 1.54.

LEMMA 1.60

If a language is regular, then it is described by a regular expression.

PROOF IDEA We need to show that if a language A is regular, a regular expression describes it. Because A is regular, it is accepted by a DFA. We describe a procedure for converting DFAs into equivalent regular expressions.

We break this procedure into two parts, using a new type of finite automaton called a ***generalized nondeterministic finite automaton***, GNFA. First we show how to convert DFAs into GNFAs, and then GNFAs into regular expressions.

Generalized nondeterministic finite automata are simply nondeterministic finite automata wherein the transition arrows may have any regular expressions as labels, instead of only members of the alphabet or ε. The GNFA reads blocks of symbols from the input, not necessarily just one symbol at a time as in an ordinary NFA. The GNFA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constitute a string described by the regular expression on that arrow. A GNFA is nondeterministic and so may have several different ways to process the same input string. It accepts its input if its processing can cause the GNFA to be in an accept state at the end of the input. The following figure presents an example of a GNFA.

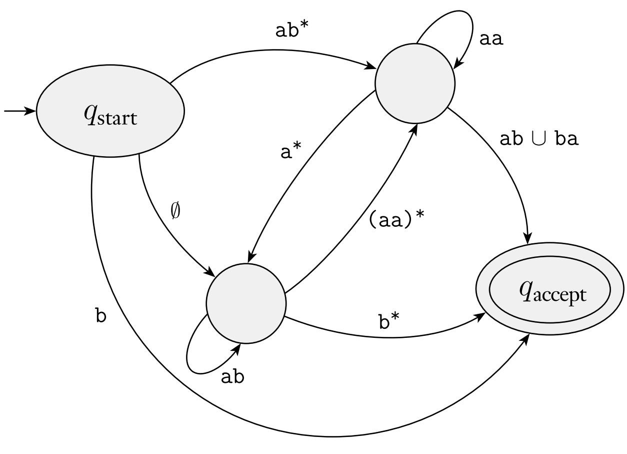


FIGURE 1.61
A generalized nondeterministic finite automaton

For convenience, we require that GNFAs always have a special form that meets the following conditions.

- The start state has transition arrows going to every other state but no arrows coming in from any other state.
- There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
- Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.

We can easily convert a DFA into a GNFA in the special form. We simply add a new start state with an ε arrow to the old start state and a new accept state with ε arrows from the old accept states. If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), we replace each with a single arrow whose label is the union of the previous labels. Finally, we add arrows labeled $\emptyset$ between states that had no arrows. This last step won't change the language recognized because a transition labeled with $\emptyset$ can never be used. From here on we assume that all GNFAs are in the special form.

Now we show how to convert a GNFA into a regular expression. Say that the GNFA has k states. Then, because a GNFA must have a start and an accept state and they must be different from each other, we know that $k \geq 2$. If $k > 2$, we construct an equivalent GNFA with $k - 1$ states. This step can be repeated on the new GNFA until it is reduced to two states. If $k = 2$, the GNFA has a single arrow that goes from the start state to the accept state. The label of this arrow is the equivalent regular expression. For example, the stages in converting a DFA with three states to an equivalent regular expression are shown in the following figure.

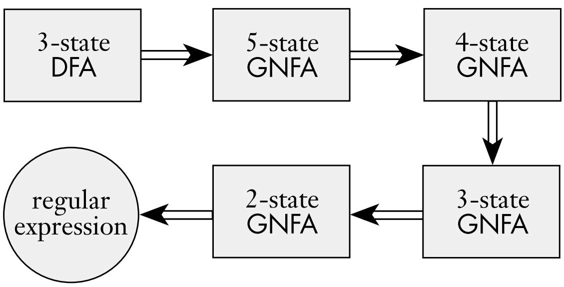


FIGURE 1.62
Typical stages in converting a DFA to a regular expression

The crucial step is constructing an equivalent GNFA with one fewer state when $k > 2$. We do so by selecting a state, ripping it out of the machine, and repairing the remainder so that the same language is still recognized. Any state will do, provided that it is not the start or accept state. We are guaranteed that such a state will exist because $k > 2$. Let's call the removed state q_{rip}.

After removing q_{rip} we repair the machine by altering the regular expressions that label each of the remaining arrows. The new labels compensate for the absence of q_{rip} by adding back the lost computations. The new label going from a state q_i to a state q_j is a regular expression that describes all strings that would take the machine from q_i to q_j either directly or via q_{rip}. We illustrate this approach in Figure 1.63.

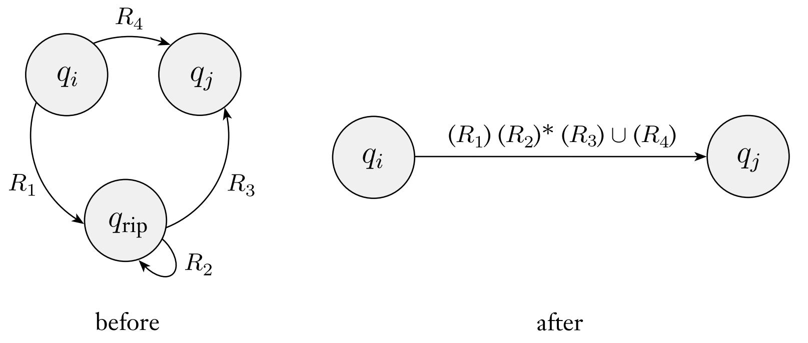


FIGURE 1.63
Constructing an equivalent GNFA with one fewer state

In the old machine, if

1. q_i goes to q_{rip} with an arrow labeled R_1,
2. q_{rip} goes to itself with an arrow labeled R_2,
3. q_{rip} goes to q_j with an arrow labeled R_3, and
4. q_i goes to q_j with an arrow labeled R_4,

then in the new machine, the arrow from q_i to q_j gets the label

$$(R_1)(R_2)^*(R_3) \cup (R_4).$$

We make this change for each arrow going from any state q_i to any state q_j, including the case where $q_i = q_j$. The new machine recognizes the original language.

PROOF Let's now carry out this idea formally. First, to facilitate the proof, we formally define the new type of automaton introduced. A GNFA is similar to a nondeterministic finite automaton except for the transition function, which has the form

$$\delta\colon \big(Q - \{q_{\text{accept}}\}\big) \times \big(Q - \{q_{\text{start}}\}\big) \longrightarrow \mathcal{R}.$$

The symbol $\mathcal{R}$ is the collection of all regular expressions over the alphabet Σ, and q_{start} and q_{accept} are the start and accept states. If $\delta(q_i, q_j) = R$, the arrow from state q_i to state q_j has the regular expression R as its label. The domain of the transition function is $\big(Q - \{q_{\text{accept}}\}\big) \times \big(Q - \{q_{\text{start}}\}\big)$ because an arrow connects every state to every other state, except that no arrows are coming from q_{accept} or going to q_{start}.

DEFINITION 1.64

A ***generalized nondeterministic finite automaton*** is a 5-tuple, $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, where

1. Q is the finite set of states,
2. Σ is the input alphabet,
3. $\delta\colon \big(Q - \{q_{\text{accept}}\}\big) \times \big(Q - \{q_{\text{start}}\}\big) \longrightarrow \mathcal{R}$ is the transition function,
4. q_{start} is the start state, and
5. q_{accept} is the accept state.

A GNFA accepts a string w in Σ^* if $w = w_1 w_2 \cdots w_k$, where each w_i is in Σ^* and a sequence of states $q_0, q_1, \ldots, q_k$ exists such that

1. $q_0 = q_{\text{start}}$ is the start state,
2. $q_k = q_{\text{accept}}$ is the accept state, and
3. for each i, we have $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$; in other words, R_i is the expression on the arrow from q_{i-1} to q_i.

Returning to the proof of Lemma 1.60, we let M be the DFA for language A. Then we convert M to a GNFA G by adding a new start state and a new accept state and additional transition arrows as necessary. We use the procedure CONVERT(G), which takes a GNFA and returns an equivalent regular expression. This procedure uses ***recursion***, which means that it calls itself. An infinite loop is avoided because the procedure calls itself only to process a GNFA that has one fewer state. The case where the GNFA has two states is handled without recursion.

CONVERT(G):

1. Let k be the number of states of G.
2. If $k = 2$, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression R.
   Return the expression R.
3. If $k > 2$, we select any state $q_{\text{rip}} \in Q$ different from q_{start} and q_{accept} and let G' be the GNFA $(Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$, where

   $$Q' = Q - \{q_{\text{rip}}\},$$

   and for any $q_i \in Q' - \{q_{\text{accept}}\}$ and any $q_j \in Q' - \{q_{\text{start}}\}$, let

   $$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4),$$

   for $R_1 = \delta(q_i, q_{\text{rip}})$, $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R_3 = \delta(q_{\text{rip}}, q_j)$, and $R_4 = \delta(q_i, q_j)$.
4. Compute CONVERT(G') and return this value.

Next we prove that CONVERT returns a correct value.

CLAIM 1.65

For any GNFA G, CONVERT(G) is equivalent to G.

We prove this claim by induction on k, the number of states of the GNFA.

Basis: Prove the claim true for $k = 2$ states. If G has only two states, it can have only a single arrow, which goes from the start state to the accept state. The regular expression label on this arrow describes all the strings that allow G to get to the accept state. Hence this expression is equivalent to G.

Induction step: Assume that the claim is true for $k - 1$ states and use this assumption to prove that the claim is true for k states. First we show that G and G' recognize the same language. Suppose that G accepts an input w. Then in an accepting branch of the computation, G enters a sequence of states:

$$q_{\text{start}}, q_1, q_2, q_3, \ldots, q_{\text{accept}}.$$

If none of them is the removed state q_{rip}, clearly G' also accepts w. The reason is that each of the new regular expressions labeling the arrows of G' contains the old regular expression as part of a union.

If q_{rip} does appear, removing each run of consecutive q_{rip} states forms an accepting computation for G'. The states q_i and q_j bracketing a run have a new regular expression on the arrow between them that describes all strings taking q_i to q_j via q_{rip} on G. So G' accepts w.

Conversely, suppose that G' accepts an input w. As each arrow between any two states q_i and q_j in G' describes the collection of strings taking q_i to q_j in G, either directly or via q_{rip}, G must also accept w. Thus G and G' are equivalent.

The induction hypothesis states that when the algorithm calls itself recursively on input G', the result is a regular expression that is equivalent to G' because G' has $k - 1$ states. Hence this regular expression also is equivalent to G, and the algorithm is proved correct.

This concludes the proof of Claim 1.65, Lemma 1.60, and Theorem 1.54.

EXAMPLE 1.66

In this example, we use the preceding algorithm to convert a DFA into a regular expression. We begin with the two-state DFA in Figure 1.67(a).

In Figure 1.67(b), we make a four-state GNFA by adding a new start state and a new accept state, called s and a instead of q_{start} and q_{accept} so that we can draw them conveniently. To avoid cluttering up the figure, we do not draw the arrows labeled $\emptyset$, even though they are present. Note that we replace the label a, b on the self-loop at state 2 on the DFA with the label $\mathtt{a} \cup \mathtt{b}$ at the corresponding point on the GNFA. We do so because the DFA's label represents two transitions, one for a and the other for b, whereas the GNFA may have only a single transition going from 2 to itself.

In Figure 1.67(c), we remove state 2 and update the remaining arrow labels. In this case, the only label that changes is the one from 1 to a. In part (b) it was $\emptyset$, but in part (c) it is $\mathtt{b}(\mathtt{a} \cup \mathtt{b})^*$. We obtain this result by following step 3 of the CONVERT procedure. State q_i is state 1, state q_j is a, and q_{rip} is 2, so $R_1 = \mathtt{b}$, $R_2 = \mathtt{a} \cup \mathtt{b}$, $R_3 = \varepsilon$, and $R_4 = \emptyset$. Therefore, the new label on the arrow from 1 to a is $(\mathtt{b})(\mathtt{a} \cup \mathtt{b})^*(\varepsilon) \cup \emptyset$. We simplify this regular expression to $\mathtt{b}(\mathtt{a} \cup \mathtt{b})^*$.

In Figure 1.67(d), we remove state 1 from part (c) and follow the same procedure. Because only the start and accept states remain, the label on the arrow joining them is the regular expression that is equivalent to the original DFA.

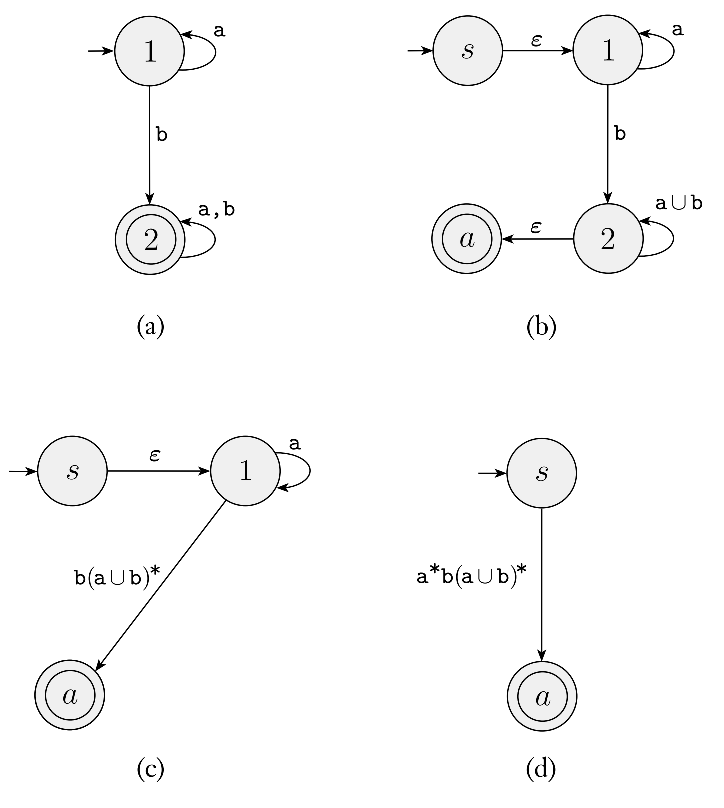


FIGURE 1.67
Converting a two-state DFA to an equivalent regular expression

EXAMPLE 1.68

In this example, we begin with a three-state DFA. The steps in the conversion are shown in the following figure.

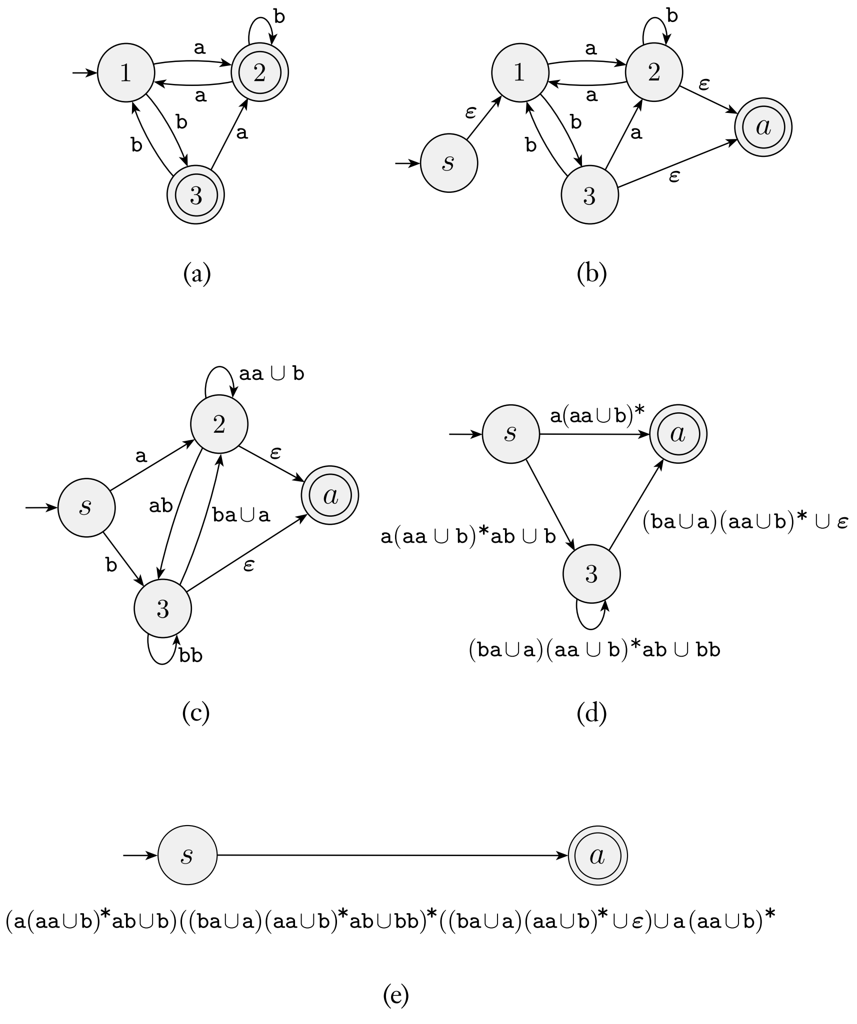


FIGURE 1.69
Converting a three-state DFA to an equivalent regular expression