

catenation, and Kleene star, we can build numerous languages from our building blocks. To convey this process more precisely, we make the following definition.

A **regular expression** (over an alphabet Σ) is defined as follows:

- \emptyset is a regular expression.
- Each member of Σ is a regular expression.
- If p and q are regular expressions then so is $(p \cup q)$.
- If p and q are regular expressions then so is $(p \circ q)$.
- If p is a regular expression then so is p^* .

As an example, if the alphabet Σ were $\{x, y, z\}$, then $(x \cup (z \circ y))$ would be a regular expression since $(z \circ y)$ would be a regular expression by Rule d, and hence $(x \cup (z \circ y))$ would be a regular expression by Rule c.

Each regular expression r over some alphabet Σ represents a language, denoted by $L(r)$, that is constructed from our building blocks. More precisely, $L(\emptyset)$ is the language \emptyset , and for each $x \in \Sigma$, $L(x)$ is the language $\{x\}$. Furthermore, if p and q are regular expressions, then $L((p \cup q)) = L(p) \cup L(q)$, $L((p \circ q)) = L(p) \circ L(q)$, and $L(p^*) = L(p)^*$. For example, the expression $(x \cup (z \circ y))$ represents the language $\{x, zy\}$. That is, it represents the language generated by forming the union of $\{x\}$ with the language obtained by concatenating $\{z\}$ and $\{y\}$. In a similar manner, the expression $((x \circ y)^* \cup z^*)$ (obtained by applying Rule d to x and y , then Rule e to $(x \circ y)$ and z , and finally Rule c to $(x \circ y)^*$ and z^*) represents the language that consists of strings of zero or more copies of the pattern xy in addition to strings of zero or more z s.

Thus, regular expressions provide yet another means, in addition to transition diagrams, automata, and grammars, of specifying languages. Theorem 1.6 shows why we are interested in languages that are represented by regular expressions.

$$\mathcal{L}_{NFA} \subseteq \mathcal{L}_{RE}$$

THEOREM 1.6

Given an alphabet Σ , the regular languages over Σ are exactly the languages that are represented by regular expressions over Σ .

PROOF

First we must show that the language represented by any regular expression is regular. We have already seen that the language \emptyset is regular as well as any language containing a single string. Moreover, we have seen that the union and concatenation of two regular languages is always regular, and that the Kleene star of any regular language is regular. Thus, this first step of our proof is complete.

Next, we show that any language accepted by a finite automaton can also be represented by a regular expression. We assume that T

is a transition diagram for some finite automaton and show, by induction on the number of states in T that are not initial or accept states, that there is a regular expression that represents the language accepted by T . The diagrams we will consider are slightly more general than traditional transition diagrams in that their arcs can be labeled by regular expressions. (To traverse such an arc requires that the machine read a pattern compatible with the expression.) If the languages accepted by such generalized diagrams can be represented by regular expressions, then any language accepted by a traditional transition diagram for a finite automaton can also be so represented.

Our induction process assumes that T has only one accept state. Otherwise, for each accept state we could make a separate copy of T in which only that state was an accept state, find regular expressions associated with each of these diagrams, and then form the union of these expressions to obtain the desired expression for T .

To begin our induction process, then, let us assume that T is a generalized transition diagram with only one accept state, and that every state in T is an initial state or an accept state. There are two possibilities: Either T contains only one state that is both the initial state and the accept state, or T contains two states of which one is the initial state and the other is the accept state. In the former case, the desired expression is merely the Kleene star of the union of the labels that appear on the arcs in the diagram.

The latter case is a bit more complex. If there are multiple arcs connecting the same states in the same direction, we replace them with a single arc labeled by the regular expression that represents the union of the original labels. At this point T can contain at most four arcs: one from the initial state back to the initial state, one from the initial state to the accept state, one from the accept state back to the accept state, and one from the accept state to the initial state. Let us denote these arcs with the labels r , s , t , and u , respectively (see Figure 1.30).

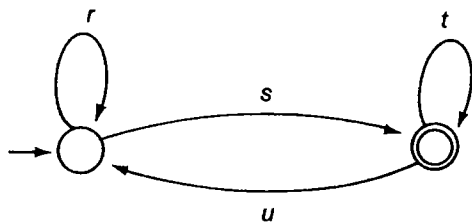


Figure 1.30 The possible arcs in a transition diagram with two states—one an initial state and the other an accept state

simple

easy

If the arc labeled s is not present in the diagram, then the associated regular expression is \emptyset since there would be no way to reach the accept state from the initial state. Otherwise, the structure of the associated regular expression depends on the absence or presence of the arc labeled u . If this arc is not present, then the expression

$$((r^* \circ s) \circ t^*)$$

is the desired regular expression, where r and t are replaced by \emptyset if the corresponding arc is not present in the diagram. If there is in fact an arc from the accept state to the initial state, then the expression

$$(((r^* \circ s) \circ t^*) \circ (u \circ ((r^* \circ s) \circ t^*)))^*$$

is the desired expression, where again r and t are replaced by \emptyset if the corresponding arc is not present in T .

Let us now suppose that $n \in \mathbb{N}$, and that the language accepted by any generalized transition diagram with no more than n states that are not initial or accept states can be represented by a regular expression. For any generalized transition diagram having $n + 1$ states that are not initial or accept states, we proceed as follows (refer to Figure 1.31): Select a state s_0 in T that is not an accept state or an initial state. Remove this state and all its incident arcs from T . For each pair of removed arcs—one leading to s_0 from another state and the other leading away from s_0 to another state, labeled p and q , respectively—draw an arc from the origin of the arc la-

any pair going into s_0

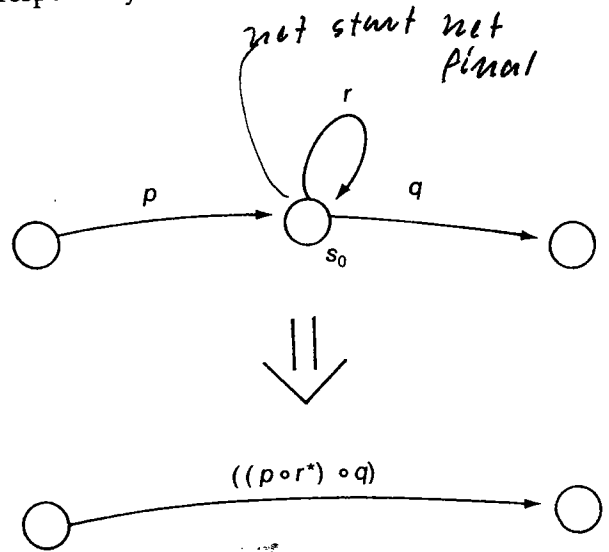


Figure 1.31 Removing a state s_0 from a generalized transition diagram for a finite automaton

beled p to the destination of the arc labeled q , and label this new arc $((p \circ r^*) \circ q)$, where r is either the union of the labels of those arcs that originally went from s_0 to s_0 , or \emptyset if there were no such arcs. The resulting diagram is another generalized transition diagram that will accept the same language as T and has only n states that are not initial or accept states. Thus, by our induction hypothesis, the language in question can be represented by a regular expression.



Theorem 1.6 provides yet another way of characterizing regular languages: They are the languages accepted by deterministic finite automata, the languages accepted by nondeterministic finite automata, the languages generated by regular grammars, and the languages represented by regular expressions.

From this we can conclude that the expressive power of regular expressions is rather limited, but this does not imply that regular expressions are useless. To the contrary, these expressions give insights into the structure of regular languages that may not be apparent in the other characterizations. Regular expressions also provide a relatively concise way of expressing and communicating regular languages. For instance, to describe a regular language by means of a finite automaton requires a definition of the transition function, perhaps in the form of a transition diagram or a transition table; to describe a language using a grammar requires a list of rewrite rules; but, by using a regular expression a language can be described via a single line of type.

A prime example of where this concise notation is advantageous is found in many of today's text editors, which allow one to search a file for a particular pattern that is to be deleted or replaced with another string. In this setting a notation system is required for communicating the form of the target pattern. Although the syntax actually used might differ from that introduced here, the theme of expressing the pattern via the operations of union, concatenation, and Kleene star is fundamental in such applications. As a particular example, when using the UNIX editor *ed*, the command

s/xx/z/*

instructs the system to substitute the first occurrence of one or more x s with a single z . Here, the expression xx^* represents a single x concatenated with the Kleene star of x .

Finally, a consequence of theorems 1.3 and 1.6 is that pattern-matching systems that rely on only union, concatenation, and Kleene star for representing patterns are rather restrictive in the patterns that can be requested. On the other hand, as a testimonial to the scope of regular languages, many very useful systems have been designed with these limitations.