CHAPTER 6

Regular Expressions

Let's now take a different approach to categorizing problems. Instead of focusing on the power of a computing device, let's look at the task that we need to perform. In particular, let's consider problems in which our goal is to match finite or repeating patterns. For example, consider:

- The first step of compiling a program: This step is called lexical analysis. Its job is to break the source code into meaningful units such as keywords, variables, and numbers. For example, the string `void` may be a keyword, while the string `23E-12` should be recognized as a floating point number.
- Filtering email for spam.
- Sorting email into appropriate mailboxes based on sender and/or content words and phrases.
- Searching a complex directory structure by specifying patterns that are known to occur in the file we want.

In this chapter, we will define a simple ***pattern language***. It has limitations. But its strength, as we will soon see, is that we can implement pattern matching for this language using finite state machines.

> In his classic book, *A Pattern Language*, Christopher Alexander described common patterns that can be found in successful buildings, towns and cities. Software engineers read Alexander's work and realized that the same is true of successful programs and systems. Patterns are ubiquitous in our world.

6.1 What is a Regular Expression?

The regular expression language that we are about to describe is built on an alphabet that contains two kinds of symbols:

- A set of special symbols to which we will attach particular meanings when they occur in a regular expression. These symbols are $\emptyset$, $\cup$, ε, (,), *, and $^+$.
- An alphabet Σ, which contains the symbols that regular expressions will match against.

A ***regular expression*** is a string that can be formed according to the following rules:

1. $\emptyset$ is a regular expression.
2. ε is a regular expression.
3. Every element in Σ is a regular expression.
4. Given two regular expressions α and β, $\alpha\beta$ is a regular expression.
5. Given two regular expressions α and β, $\alpha \cup \beta$ is a regular expression.
6. Given a regular expression α, α^* is a regular expression.
7. Given a regular expression α, α^+ is a regular expression.
8. Given a regular expression α, (α) is a regular expression.

So, if we let $\Sigma = \{a, b\}$, the following strings are regular expressions:

$$\emptyset, \varepsilon, a, b, (a \cup b)^*, abba \cup \varepsilon.$$

The language of regular expressions, as we have just defined it, is useful because every regular expression has a meaning (just like every English sentence and every Java program). In the case of regular expressions, the meaning of a string is another language. In other words, every string α (such as abba $\cup \varepsilon$) in the regular expression language has, as its meaning, some new language that contains exactly the strings that match the pattern specified in α.

To make it possible to determine that meaning, we need to describe a semantic interpretation function for regular expressions. Fortunately, the regular expressions language is simple. So designing a compositional semantic interpretation function (as defined in Section 2.2.6) for it is straightforward. As you read the definition that we are about to present, it will become clear why we chose the particular symbol alphabet we did. In particular, you will notice the similarity between the operations that are allowed in regular expressions and the operations that we defined on languages in Section 2.2.

Define the following semantic interpretation function L for the language of regular expressions:

1. $L(\emptyset) = \emptyset$, the language that contains no strings.
2. $L(\varepsilon) = \{\varepsilon\}$, the language that contains just the empty string.
3. For any $c \in \Sigma$, $L(c) = \{c\}$, the language that contains the single, one-character string c.

4. For any regular expressions α and β, $L(\alpha\beta) = L(\alpha)\,L(\beta)$. In other words, to form the meaning of the concatenation of two regular expressions, first determine the meaning of each of the constituents. Both meanings will be languages. Then concatenate the two languages together. Recall that the concatenation of two languages L_1 and L_2 is $\{w = xy$, where $x \in L_1$ and $y \in L_2\}$. Note that, if either $L(\alpha)$ or $L(\beta)$ is equal to $\emptyset$, then the concatenation will also be equal to $\emptyset$.
5. For any regular expressions α and β, $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$. Again we form the meaning of the larger expression by first determining the meaning of each of the constituents. Each of them is a language. The meaning of $\alpha \cup \beta$ then, as suggested by our choice of the character $\cup$ as an operator, is the union of the two constituent languages.
6. For any regular expression α, $L(\alpha^*) = (L(\alpha))^*$, where * is the Kleene star operator defined in Section 2.2.5. So $L(\alpha^*)$ is the language that is formed by concatenating together zero or more strings drawn from $L(\alpha)$.
7. For any regular expression α, $L(\alpha^+) = L(\alpha\alpha^*) = L(\alpha)(L(\alpha))^*$. If $L(\alpha)$ is equal to $\emptyset$, then $L(\alpha^+)$ is also equal to $\emptyset$. Otherwise $L(\alpha^+)$ is the language that is formed by concatenating together one or more strings drawn from $L(\alpha)$.
8. For any regular expression α, $L((\alpha)) = L(\alpha)$. In other words, parentheses have no effect on meaning except to group the constituents in an expression.

If the meaning of a regular expression α is the language L, then we say that α ***defines*** or ***describes*** L.

The definition that we have just given for the regular expression language contains three kinds of rules:

- Rules 1, 3, 4, 5, and 6 give the language its power to define sets, starting with the basic sets defined by rules 1 and 3, and then building larger sets using the operators defined by rules 4, 5, and 6.
- Rule 8 has as its only role grouping other operators.
- Rules 2 and 7 appear to add functionality to the regular expression language. But in fact they don't—they serve only to provide convenient shorthands for languages that can be defined using only rules 1, 3-6, and 8. Let's see why.

First consider rule 2: The language of regular expressions does not need the symbol ε because it has an alternative mechanism for describing $L(\varepsilon)$. Observe that $L(\emptyset^*) = \{w : w$ is formed by concatenating together zero or more strings from $\emptyset\}$. But how many ways are there to concatenate together zero or more strings from $\emptyset$? If we select zero strings to concatenate, we get ε. We cannot select more than zero since there aren't any to choose from. So $L(\emptyset^*) = \{\varepsilon\}$. Thus, whenever we would like to write ε, we could instead write $\emptyset^*$. It is much clearer to write ε, and we shall. But, whenever we wish to make a formal statement about regular expressions or the languages they define, we need not consider rule 2 since we can rewrite any regular expression that contains ε as an equivalent one that contains $\emptyset^*$ instead.

Next consider rule 7: As we showed in the statement of rule 7 itself, the regular expression α^+ is equivalent to the slightly longer regular expression $\alpha\alpha^*$. The form α^+ is a

convenient shortcut, and we will use it. But we need not consider rule 7 in any analysis that we may choose to do of regular expressions or the languages that they generate.

The compositional semantic interpretation function that we just defined lets us map between regular expressions and the languages that they define. We begin by analyzing the smallest subexpressions and then work outward to larger and larger expressions.

EXAMPLE 6.1 Analyzing a Simple Regular Expression

$$\begin{aligned} L((a \cup b)^*b) &= L((a \cup b)^*)L(b) \\ &= (L((a \cup b)))^*L(b) \\ &= (L(a) \cup L(b))^*L(b) \\ &= (\{a\} \cup \{b\})^*\{b\} \\ &= \{a, b\}^*\{b\}. \end{aligned}$$

So the meaning of the regular expression (a ∪ b)*b is the set of all strings over the alphabet {a, b} that end in b.

One straightforward way to read a regular expression and determine its meaning is to imagine it as a procedure that generates strings. Read it left to right and imagine it generating a string left to right. As you are doing that, think of any expression that is enclosed in a Kleene star as a loop that can be executed zero or more times. Each time through the loop, choose any one of the alternatives listed in the expression. So we can read the regular expression of the last example, (a ∪ b)*b, as, "Go through a loop zero or more times, picking a single a or b each time. Then concatenate b." Any string that can be generated by this procedure is in $L((a \cup b)^*b)$.

> Regular expressions can be used to scan text and pick out email addresses. (O.2)

EXAMPLE 6.2 Another Simple Regular Expression

$$\begin{aligned} L(((a \cup b)(a \cup b))a(a \cup b)^*) &= L(((a \cup b)(a \cup b)))L(a)\ L((a \cup b)^*) \\ &= L((a \cup b)(a \cup b))\ \{a\}\ (L((a \cup b)))^* \\ &= L((a \cup b))L((a \cup b))\ \{a\}\ \{a, b\}^* \\ &= \{a, b\}\ \{a, b\}\ \{a\}\ \{a, b\}^* \end{aligned}$$

So the meaning of the regular expression ((a ∪ b)(a ∪ b))a(a ∪ b)* is:

$$\{xay : x \text{ and } y \text{ are strings of a's and b's and } |x| = 2\}.$$

Alternatively, it is the language that contains all strings of a's and b's such that there exists a third character and it is an a.

EXAMPLE 6.3 Given a Language, Find a Regular Expression

Let $L = \{w \in \{a, b\}^* : |w| \textit{ is even}\}$. There are two simple regular expressions both of which define L:

((a ∪ b)(a ∪ b))*	This one can be read as, "Go through a loop zero or more times. Each time through, choose an a or b, then choose a second character (a or b)."
(aa ∪ ab ∪ ba ∪ bb)*	This one can be read as, "Go through a loop zero or more times. Each time through, choose one of the two-character sequences."

From this example, it is clear that the semantic interpretation function we have defined for regular expressions is not one-to-one. In fact, given any language L, if there is one regular expression that defines it, there is an infinite number that do. This is trivially true since, for any regular expression α, the regular expression $\alpha \cup \alpha$ defines the same language α does.

Recall from our discussion in Section 2.2.6 that this is not unusual. Semantic interpretation functions for English and for Java are not one-to-one. The practical consequence of this phenomenon for regular expressions is that, if we are trying to design a regular expression that describes some particular language, there will be more than one right answer. We will generally seek the simplest one that works, both for clarity and to make pattern matching fast.

EXAMPLE 6.4 More than One Regular Expression for a Language

Let $L = \{w \in \{a, b\}^* : w \text{ contains an odd number of a's}\}$. Two equally simple regular expressions that define L are:

b* (ab*ab*)* a b*.

b* a b* (ab*ab*)*.

EXAMPLE 6.4 *(Continued)*

Both of these expressions require that there be a single a somewhere. There can also be other a's, but they must occur in pairs, so the result is an odd number of a's. In the first expression, the last a in the string is viewed as the required "odd a". In the second, the first a plays that role.

The regular expression language that we have just defined provides three operators. We will assign the following precedence order to them (from highest to lowest):

1. Kleene star,
2. concatenation, and
3. union.

So the expression (a ∪ bb*a) will be interpreted as (a ∪ (b(b*a))).

All useful languages have idioms: common phrases that correspond to common meanings. Regular expressions are no exception. In writing them, we will often use the following:

$(\alpha \cup \varepsilon)$	Can be read as "optional α", since the expression can be satisfied either by matching α or by matching the empty string.
(a ∪ b)*	Describes the set of all strings composed of the characters a and b. More generally, given any alphabet $\Sigma = \{c_1, c_2, \ldots, c_n\}$, the language Σ^* is described by the regular expression: $(c_1 \cup c_2 \cup \cdots \cup c_n)^*$.

When writing regular expressions, the details matter. For example:

a* ∪ b* ≠ (a ∪ b)*	The language on the right contains the string ab, while the language on the left does not. Every string in the language on the left contains only a's or only b's.
(ab)* ≠ a*b*	The language on the left contains the string abab, while the language on the right does not. The language on the right contains the string aaabbbb, while the language on the left does not.

The regular expression a* is simply a string. It is different from the language L(a*) = $\{w : w$ is composed of zero or more a's$\}$. However, when no confusion will result, we will use regular expressions to stand for the languages that they describe and we will no longer write the semantic interpretation function explicitly. So we will be able to say things like, "The language a* is infinite."

6.2 Kleene's Theorem

The regular expression language that we have just described is significant for two reasons:

- It is a useful way to define patterns.
- The languages that can be defined with regular expressions are, as the name perhaps suggests, exactly the regular languages. In other words, any language that can be defined by a regular expression can be accepted by some finite state machine. And any language that can be accepted by a finite state machine can be defined by some regular expressions.

In this section, we will state and prove as a theorem the claim that we just made: The class of languages that can be defined with regular expressions is exactly the regular languages. This is the first of several claims of this sort that we will make in this book. In each case, we will assert that some set A is identical to some very different looking set B. The proof strategy that we will use in all of these cases is the same. We will first prove that every element of A is also an element of B. We will then prove that every element of B is also an element of A. Thus, since A and B contain the same elements, they are the same set.

6.2.1 Building an FSM from a Regular Expression

THEOREM 6.1 For Every Regular Expression There is an Equivalent FSM

Theorem: Any language that can be defined with a regular expression can be accepted by some FSM and so is regular.

Proof: The proof is by construction. We will show that, given a regular expression α, we can construct an FSM M such that $L(\alpha) = L(M)$.

We first show that there exists an FSM that corresponds to each primitive regular expression:

- If α is any $c \in \Sigma$, we construct for it the simple FSM shown in Figure 6.1 (a).
- If α is $\varnothing$, we construct for it the simple FSM shown in Figure 6.1 (b).
- Although it's not strictly necessary to consider ε since it has the same meaning as $\varnothing^*$ we'll do so since we don't usually think of it that way. So, if α is ε, we construct for it the simple FSM shown in Figure 6.1 (c).

Next we must show how to build FSMs to accept languages that are defined by regular expressions that exploit the operations of concatenation, union, and Kleene star. Let β and γ be regular expressions that define languages over the alphabet Σ. If $L(\beta)$ is regular, then it is accepted by some FSM $M_1 = (K_1, \Sigma, \delta_1, s_1, A_1)$. If $L(\gamma)$ is regular, then it is accepted by some FSM $M_2 = (K_2, \Sigma, \delta_2, s_2, A_2)$.

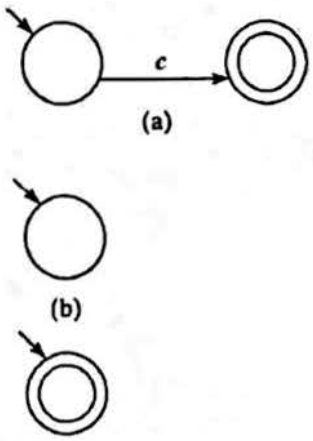


FIGURE 6.1 FSMs for primitive regular expressions.

- If α is the regular expression $\beta \cup \gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular, then we construct $M_3 = (K_3, \Sigma, \delta_3, s_3, A_3)$ such that $L(M_3) = L(\alpha) = L(\beta) \cup L(\gamma)$. If necessary, rename the states of M_1 and M_2 so that $K_1 \cap K_2 = \varnothing$. Create a new start state, s_3, and connect it to the start states of M_1 and M_2 via ε-transitions. M_3 accepts iff either M_1 or M_2 accepts. So $M_3 = (\{s_3\} \cup K_1 \cup K_2, \Sigma, \delta_3, s_3, A_1 \cup A_2)$, where $\delta_3 = \delta_1 \cup \delta_2 \cup \{((s_3, \varepsilon), s_1), ((s_3, \varepsilon), s_2)\}$.
- If α is the regular expression $\beta\gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular, then we construct $M_3 = (K_3, \Sigma, \delta_3, s_3, A_3)$ such that $L(M_3) = L(\alpha) = L(\beta)L(\gamma)$. If necessary, rename the states of M_1 and M_2 so that $K_1 \cap K_2 = \varnothing$. We will build M_3 by connecting every accepting state of M_1 to the start state of M_2 via an ε-transition. M_3 will start in the start state of M_1 and will accept iff M_2 does. So $M_3 = (K_1 \cup K_2, \Sigma, \delta_3, s_1, A_2)$, where $\delta_3 = \delta_1 \cup \delta_2 \cup \{((q, \varepsilon), s_2) : q \in A_1\}$.
- If α is the regular expression β^* and if $L(\beta)$ is regular, then we construct $M_2 = (K_2, \Sigma, \delta_2, s_2, A_2)$ such that $L(M_2) = L(\alpha) = L(\beta)^*$. We will create a new start state s_2 and make it accepting, thus assuring that M_2 accepts ε. (We need a new start state because it is possible that s_1, the start state of M_1, is not an accepting state. If it isn't and if it is reachable via any input string other than ε, then simply making it an accepting state would cause M_2 to accept strings that are not in $(L(M_1))^*$.) We link the new s_2 to s_1 via an ε-transitions. Finally, we create ε-transitions from each of M_1's accepting states back to s_1. So $M_2 = (\{s_2\} \cup K_1, \Sigma, \delta_2, s_2, \{s_2\} \cup A_1)$, where $\delta_2 = \delta_1 \cup \{((s_2, \varepsilon), s_1)\} \cup \{((q, \varepsilon), s_1) : q \in A_1\}$.

Notice that the machines that these constructions build are typically highly nondeterministic because of their use of ε-transitions. They also typically have a large number of unnecessary states. But, as a practical matter, that is not a problem since, given an arbitrary NDFSM M, we have an algorithm that can construct an equivalent DFSM M'. We also have an algorithm that can minimize M'.

Based on the constructions that have just been described, we can define the following algorithm to construct, given a regular expression α, a corresponding (usually nondeterministic) FSM:

regextofsm(α: regular expression) =

Beginning with the primitive subexpressions of α and working outwards until an FSM for all of α has been built do:

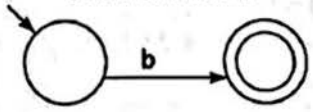
Construct an FSM as described above.

The fact that regular expressions can be transformed into executable finite state machines is important. It means that people can specify programs as regular expressions and then have those expressions "compiled" into efficient processes. For example, hierarchically structured regular expressions, with the same formal power as the regular expressions we have been working with, can be used to describe a lightweight parser for analyzing legacy software. (H.4.1)

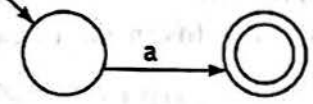
EXAMPLE 6.5 Building an FSM from a Regular Expression

Consider the regular expression (b ∪ ab)*. We use *regextofsm* to build an FSM that accepts the language defined by this regular expression:

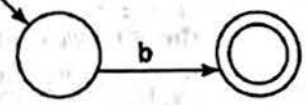
An FSM for b



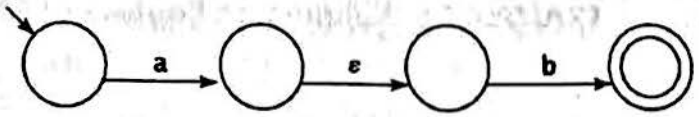
An FSM for a



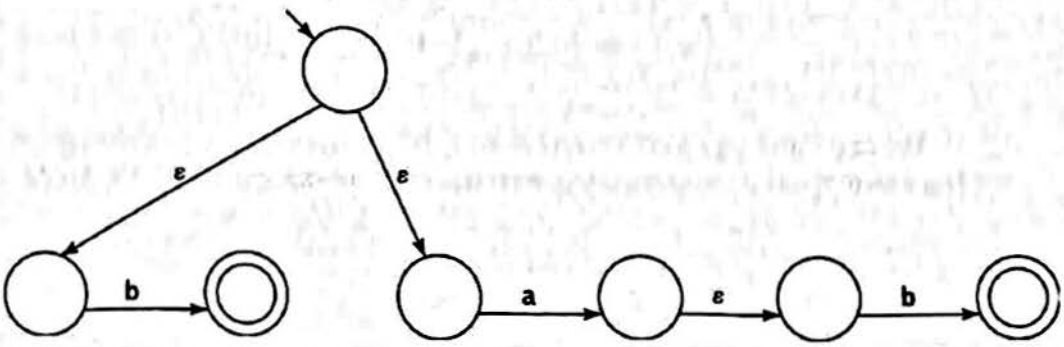
An FSM for b



An FSM for ab:

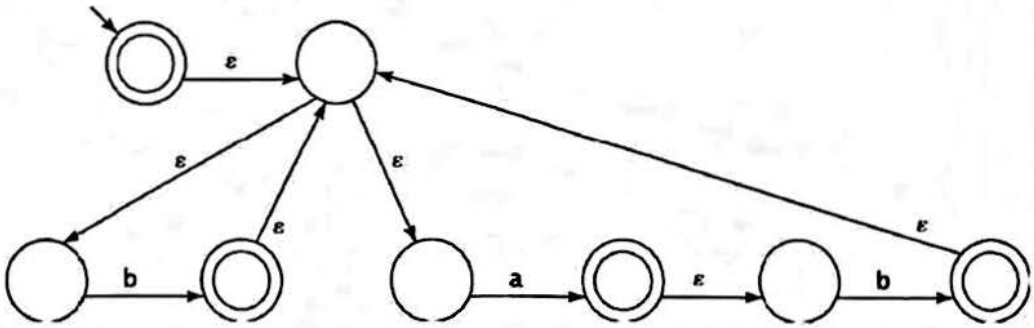


An FSM for (b ∪ ab):



EXAMPLE 6.5 *(Continued)*

An FSM for (b ∪ ab)*

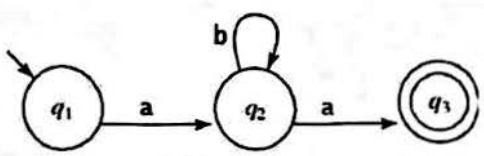


6.2.2 Building a Regular Expression from an FSM

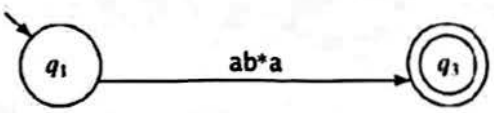
Next we must show that it is possible to go the other direction, namely to build, from an FSM, a corresponding regular expression. The idea behind the algorithm that we are about to present is the following: Instead of limiting the labels on the transitions of an FSM to a single character or ε, we will allow entire regular expressions as labels. The goal of the algorithm is to construct, from an input FSM M, an output machine M' such that M and M' are equivalent and M' has only two states, a start state and a single accepting state. It will also have just one transition, which will go from its start state to its accepting state. The label on that transition will be a regular expression that describes all the strings that could have driven the original machine M from its start state to some accepting state.

EXAMPLE 6.6 Building an Equivalent Machine M

Let M be:



We can build an equivalent machine M' by ripping out q_2 and replacing it by a transition from q_1 to q_3 labeled with the regular expression ab *a. So M' is:



Given an arbitrary FSM M, M' will be built by starting with M and then removing, one at a time, all the states that lie in between the start state and an accepting state. As each such state is removed, the remaining transitions will be modified so that the set of strings that can drive M' from its start state to some accepting state remains unchanged.

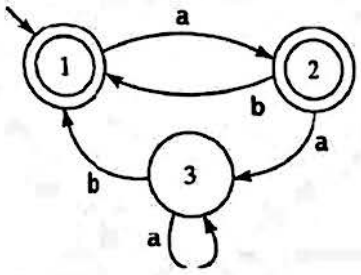
The following algorithm creates a regular expression that defines $L(M)$, provided that step 6 can be executed correctly:

fsmtoregexheuristic(M: FSM) =

1. Remove from M any states that are unreachable from the start state.
2. If M has no accepting states then halt and return the simple regular expression $\emptyset$.
3. If the start state of M is part of a loop (i.e., it has any transitions coming into it), create a new start state s and connect s to M's start state via an ε-transition. This new start state s will have no transitions into it.
4. If there is more than one accepting state of M or if there is just one but there are any transitions out of it, create a new accepting state and connect each of M's accepting states to it via an ε-transition. Remove the old accepting states from the set of accepting states. Note that the new accepting state will have no transitions out from it.
5. If, at this point, M has only one state, then that state is both the start state and the accepting state and M has no transitions. So $L(M) = \{\varepsilon\}$. Halt and return the simple regular expression ε.
6. Until only the start state and the accepting state remain do:
   - 6.1. Select some state *rip* of M. Any state except the start state or the accepting state may be chosen.
   - 6.2. Remove *rip* from M.
   - 6.3. Modify the transitions among the remaining states so that M accepts the same strings. The labels on the rewritten transitions may be any regular expression.
7. Return the regular expression that labels the one remaining transition from the start state to the accepting state.

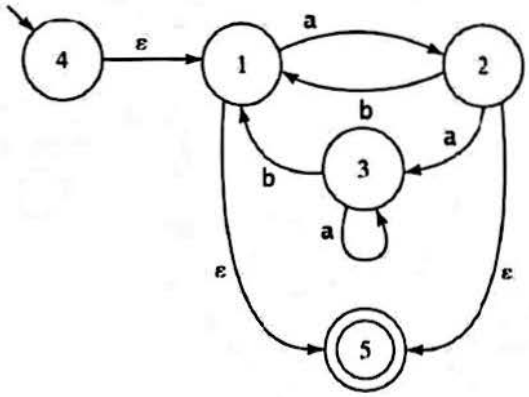
EXAMPLE 6.7 Building a Regular Expression from an FSM

Let M be:

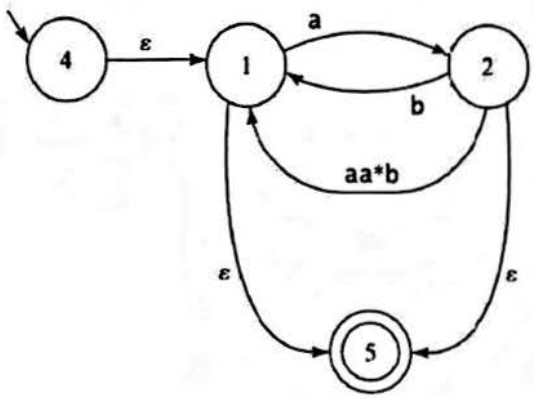


EXAMPLE 6.7 *(Continued)*

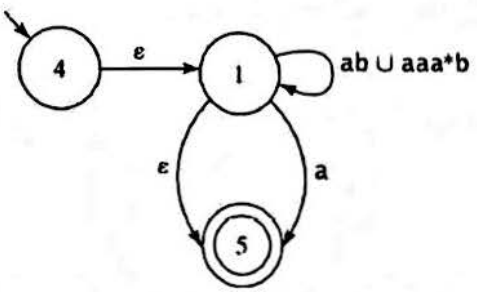
Create a new start state and a new accepting state and link them to M:



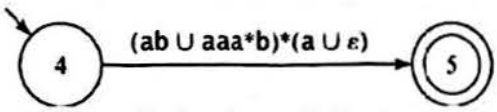
Remove state 3:



Remove state 2:

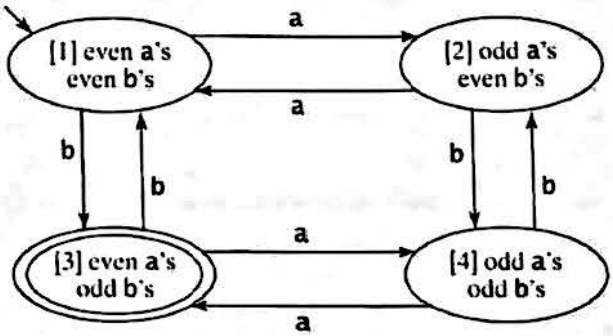


Remove state 1:



EXAMPLE 6.8 A Simple FSM With No Simple Regular Expression

Let M be the FSM that we built in Example 5.9 for the language $L = \{w \in \{a, b\}^* : w$ contains an even number of a's and an odd number of b's$\}$. M is:



Try to apply *fsmtoregexheuristic* to M. It will not be easy because it is not at all obvious how to implement step 6.3. For example, if we attempt to remove state [2], this changes not just the way that M can move from state [1] to state [4]. It also changes, for example, the way that M can move from state [1] to state [3] because it changes how M can move from state [1] back to itself.

To prove that for every FSM there exists a corresponding regular expression will require a construction in which we make clearer what must be done each time a state is removed and replaced by a regular expression. The algorithm that we are about to describe has that property, although it comes at the expense of simplicity in easy cases such as the one in Example 6.7.

THEOREM 6.2 For Every FSM There is an Equivalent Regular Expression

Theorem: Every regular language (i.e., every language that can be accepted by some FSM) can be defined with a regular expression.

Proof: The proof is by construction. Given an FSM $M = (K, \Sigma, \delta, s, A)$, we can construct a regular expression α such that $L(M) = L(\alpha)$.

As we did in *fsmtoregexheuristic*, we will begin by assuring that M has no unreachable states and that it has a start state that has no transitions into it and a single accepting state that has no transitions out from it. But now we will make a further important modification to M before we start removing states: From every state other than the accepting state there must be exactly one transition to every state (including itself) except the start state. And into every state other than the start state there must be exactly one transition from every state (including itself) except the accepting state. To make this true, we do two things:

- If there is more than one transition between states p and q, collapse them into a single transition. If the set of labels on the original set of such transitions is

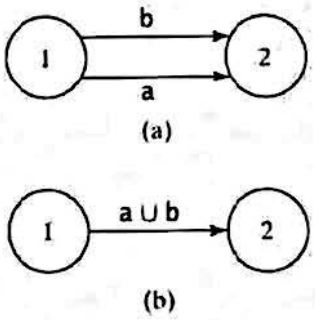
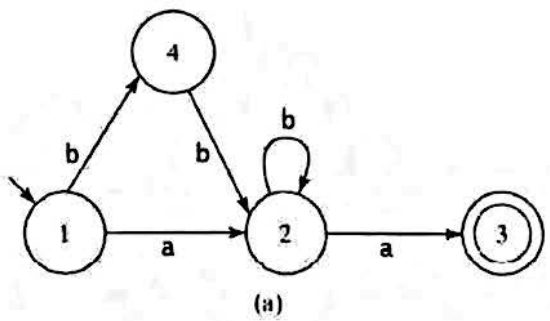


FIGURE 6.2 Collapsing multiple transitions into one.

$\{c_1, c_2, \ldots, c_n\}$, then delete those transitions and replace them by a single transition with the label $c_1 \cup c_2 \cup \ldots \cup c_n$. For example, consider the FSM fragment shown in Figure 6.2(a). We must collapse the two transitions between states 1 and 2. After doing so, we have the fragment shown in Figure 6.2(b).

- If any of the required transitions are missing, add them. We can add all of those transitions without changing $L(M)$ by labeling all of the new transitions with the regular expression $\emptyset$. So there is no string that will allow them to be taken. For example, let M be the FSM shown in Figure 6.3(a). Several new transitions are required. When we add them, we have the new FSM shown in Figure 6.3(b).



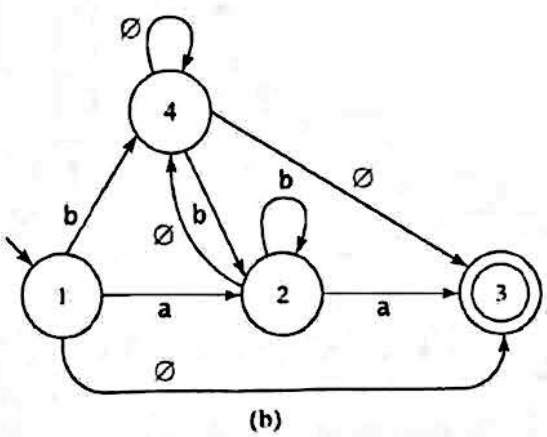


FIGURE 6.3 Adding all the required transitions.

Now suppose that we select a state *rip* and remove it and the transitions into and out of it. Then we must modify every remaining transition so that M's function stays the same. Since M already contains a transition between each pair of states (except the ones that are not allowed into and out of the start and accepting states), if all those transitions are modified correctly then M's behavior will be correct.

So, suppose that we remove some state that we will call *rip*. How should the remaining transitions be changed? Consider any pair of states p and q. Once we remove *rip*, how can M get from p to q?

- It can still take the transition that went directly from p to q, or
- It can take the transition from p to *rip*. Then, it can take the transition from *rip* back to itself zero or more times. Then it can take the transition from *rip* to q.

Let $R(p, q)$ be the regular expression that labels the transition in M from p to q. Then, in the new machine M' that will be created by removing *rip*, the new regular expression that should label the transition from p to q is:

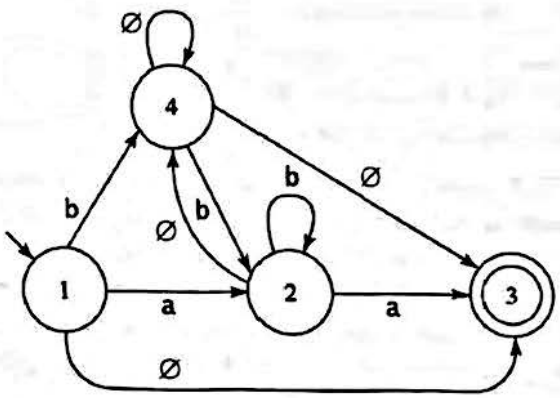
$R(p, q)$		/* Go directly from p to q,
	$\cup$	/* or
$R(p, rip)$		/* go from p to *rip*, then
$R(rip, rip)^*$		/*go from *rip* back to itself any number of times, then
$R(rip, q)$		/* go from *rip* to q.

We'll denote this new regular expression $R'(p, q)$. Writing it out without the comments, we have:

$$R' = R(p, q) \cup R(p, rip)R(rip, rip)^*R(rip, q).$$

EXAMPLE 6.9 Ripping States Out One at a Time

Again, let M be:



Let *rip* be state 2. Then:

$$R'(1, 3) = R(1, 3) \cup R(1, rip)R(rip, rip)^*R(rip, 3).$$
$$= R(1, 3) \cup R(1, 2)R(2, 2)^*R(2, 3).$$

EXAMPLE 6.9 *(Continued)*

$$= \varnothing \cup a \quad b^* \quad a,$$
$$= ab^*a.$$

Notice that ripping state 2 also changes another way the original machine had to get from state 1 to state 3: It could have gone from state 1 to state 4 to state 2 and then to state 3. But we don't have to worry about that in computing $R'(1,3)$. The required change to that path will occur when we compute $R'(4,3)$.

When all states except the start state s and the accepting state a have been removed, $R(s, a)$ will describe the set of strings that can drive M from its start state to its accepting state. So $R(s, a)$ will describe $L(M)$.

We can now define an algorithm to build, from any FSM $M = (K, \Sigma, \delta, s, A)$, a regular expression that describes $L(M)$. We'll use two subroutines, *standardize*, which will convert M to the required form, and *buildregex*, which will construct, from the modified machine M, the required regular expression.

standardize(M: FSM) =

1. Remove from M any states that are unreachable from the start state.
2. If the start state of M is part of a loop (i.e., it has any transitions coming into it), create a new start state s and connect s to M's start state via an ε-transition.
3. If there is more than one accepting state of M or if there is just one but there are any transitions out of it, create a new accepting state and connect each of M's accepting states to it via an ε-transition. Remove the old accepting states from the set of accepting states.
4. If there is more than one transition between states p and q, collapse them into a single transition.
5. If there is a pair of states p, q and there is no transition between them and p is not the accepting state and q is not the start state, then create a transition from p to q labeled $\varnothing$.

buildregex(M: FSM) =

1. If M has no accepting states, then halt and return the simple regular expression $\varnothing$.
2. If M has only one state, then halt and return the simple regular expression ε.
3. Until only the start state and the accepting state remain do:
   3.1. Select some state *rip* of M. Any state except the start state or the accepting state may be chosen.

3.2. For every transition from some state p to some state q, if both p and q are not *rip* then, using the current labels given by the expressions R, compute the new label R' for the transition from p to q using the formula:

$$R'(p, q) = R(p, q) \cup R(p, rip)R(rip, rip)^*R(rip, q).$$

3.3. Remove *rip* and all transitions into and out of it.

4. Return the regular expression that labels the one remaining transition from the start state to the accepting state.

We can show that the new FSM that is built by *standardize* is equivalent to the original machine (i.e., that they accept the same language) by showing that the language that is accepted is preserved at each step of the procedure. We can show that *buildregex*(M) builds a regular expression that correctly defines $L(M)$ by induction on the number of states that must be removed before it halts. Using those two procedures, we can now define:

fsmtoregex(M: FSM) =

1. M' = *standardize* (M).
2. Return *buildregex*(M').

6.2.3 The Equivalence of Regular Expressions and FSMs

The last two theorems enable us to prove the next one, due to Stephen Kleene 💻.

THEOREM 6.3 Kleene's Theorem

Theorem: The class of languages that can be defined with regular expressions is exactly the class of regular languages.

Proof: Theorem 6.1 says that every language that can be defined with a regular expression is regular. Theorem 6.2 says that every regular language can be defined by some regular expression.

6.2.4 Kleene's Theorem, Regular Expressions, and Finite State Machines

Kleene's Theorem tells us that there is no difference between the formal power of regular expressions and finite state machines. But, as some of the examples that we just considered suggest, there is a practical difference in their effectiveness as problem solving tools:

- As we said in the introduction to this chapter, the regular expression language is a pattern language. In particular, regular expressions must specify the order in which a sequence of symbols must occur. This is useful when we want to describe patterns such as phone numbers (it matters that the area code comes first) or email addresses (it matters that the user name comes before the domain).

- But there are some applications where order doesn't matter. The vending machine example that we considered at the beginning of Chapter 5 is an instance of this class of problem. The order in which the coins were entered doesn't matter. Parity checking is another. Only the total number of 1 bits matters, not where they occur in the string. Finite state machines can be very effective in solving problems such as this. But the regular expressions that correspond to those FSMs may be too complex to be useful.

The bottom line is that sometimes it is easy to write a finite state machine to describe a language. For other problems, it may be easier to write a regular expression.

Sometimes Writing Regular Expressions is Easy

Because, for some problems, regular expressions are easy to write, Kleene's theorem is useful. It gives us a second way to show that a language is regular. We need only show a regular expression that defines it.

EXAMPLE 6.10 No More Than One b

Let $L = \{w \in \{a, b\}^* :$ there is no more than one b$\}$. L is regular because it can be described with the following regular expression:

$$a^* (b \cup \varepsilon) a^*.$$

EXAMPLE 6.11 No Two Consecutive Letters are the Same

Let $L = \{w \in \{a, b\}^* :$ no two consecutive letters are the same$\}$. L is regular because it can be described with either of the following regular expressions:

$$(b \cup \varepsilon) (ab)^* (a \cup \varepsilon).$$
$$(a \cup \varepsilon) (ba)^* (b \cup \varepsilon).$$

EXAMPLE 6.12 Floating Point Numbers

Consider again FLOAT, the language of floating point numbers that we described in Example 5.7. Kleene's Theorem tells us that, since FLOAT is regular, there must be some regular expression that describes it. In fact, regular expressions can be used easily to describe languages like FLOAT. We'll use one shorthand. Let:

D stand for $(0 \cup 1 \cup 2 \cup 3 \cup 4 \cup 5 \cup 6 \cup 7 \cup 8 \cup 9)$.

Then FLOAT is the language described by the following regular expression:

$$(\varepsilon \cup + \cup -)D^+ (\varepsilon \cup . D^+) (\varepsilon \cup (E (\varepsilon \cup + \cup -)D^+).$$

It is useful to think of programs, queries, and other strings in practical languages as being composed of a sequence of tokens, where a token is the smallest string that has meaning. So variable and function names, numbers and other constants, operators, and reserved words are all tokens. The regular expression we just wrote for the language FLOAT describes one kind of token. The first thing a compiler does, after reading its input, is to divide it into tokens. That process is called lexical analysis. It is common to use regular expressions to define the behavior of a lexical analyzer. (G.4.1)

Sometimes Building a Deterministic FSM is Easy

Given an arbitrary regular expression, the general algorithms presented in the proof of Theorem 6.1 will typically construct a highly nondeterministic FSM. But there is a useful special case in which it is possible to construct a DFSM directly from a set of patterns. Suppose that we are given a set K of n keywords and a text string s. We want to find occurrences in s of the keywords in K. We can think of K as defining a language that can be described by a regular expression of the form:

$$(\Sigma^*(k_1 \cup k_2 \cup \ldots \cup k_n)\Sigma^*)^+.$$

In other words, we will accept any string in which at least one keyword occurs. For some applications this will be good enough. For others, we may care which keyword was matched. For yet others we'll want to find all substrings that match some keyword in K.

By letting the keywords correspond to sequences of amino acids, this idea can be used to build a fast search engine for protein databases. (K.3)

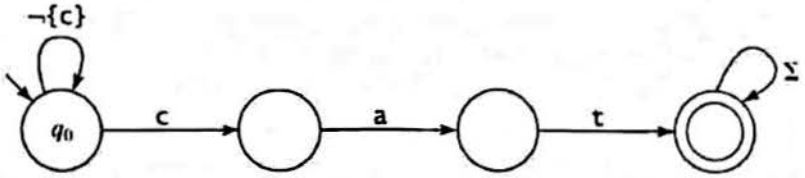
In any of these special cases, we can build a deterministic FSM M by first building a decision tree out of the set of keywords and then adding arcs as necessary to tell M what to do when it reaches a dead end branch of the tree. The following algorithm builds an FSM that accepts any string that contains at least one of the specified keywords:

buildkeywordFSM(K: set of keywords) =

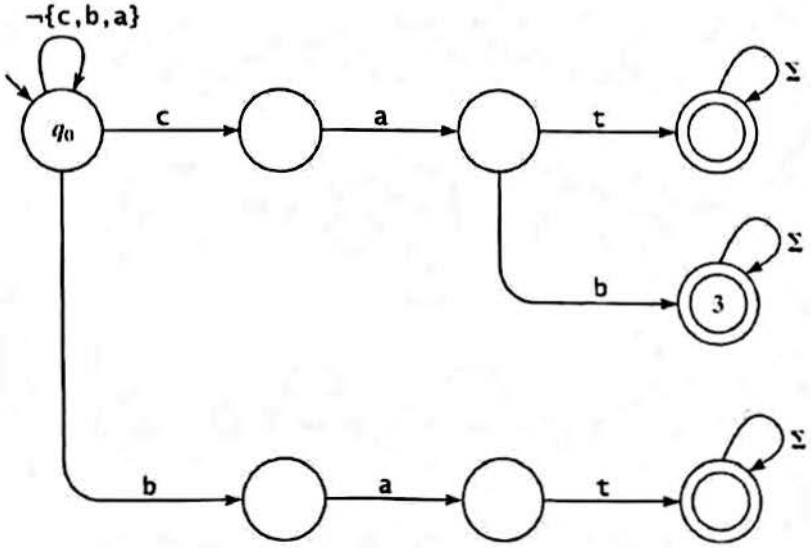
1. Create a start state q_0.
2. For each element k of K do:
   Create a branch corresponding to k.
3. Create a set of transitions that describe what to do when a branch dies, either because its complete pattern has been found or because the next character is not the correct one to continue the pattern.
4. Make the states at the ends of each branch accepting.

EXAMPLE 6.13 Recognizing a Set of Keywords

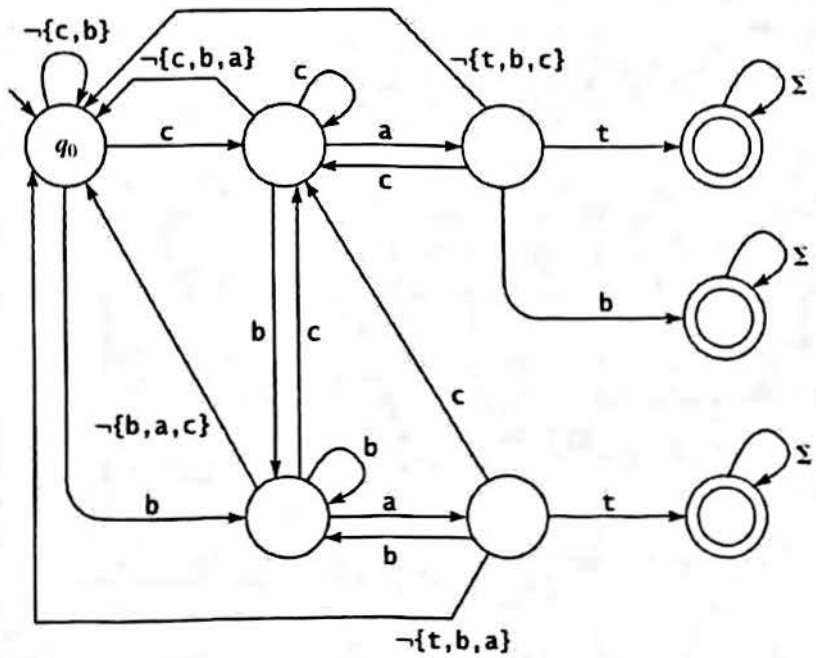
Consider the set of keywords {cat, bat, cab}. We can use *buildkeywordFSM* to build a DFSM to accept strings that contain at least one of these keywords. We begin by creating a start state and then a path to accept the first keyword, cat:



Next we add branches for the remaining keywords, bat and cab:



Finally, we add transitions that let the machine recover after a path dies:



6.3 Applications of Regular Expressions

Patterns are everywhere.

> Regular expressions can be matched against the subject fields of emails to find at least some of the ones that are likely to be spam. (O.1)

Because patterns are everywhere, applications of regular expressions are everywhere. Before we look at some specific examples, one important caveat is required: The term *regular expression* is used in the modern computing world 💻 in a much more general way than we have defined it here. Many programming languages and scripting systems provide support for regular expression matching. Each of them has its own syntax. They all have the basic operators union, concatenation, and Kleene star. They typically have others as well. Many, for example, have a substitution operator so that, after a pattern is successfully matched against a string, a new string can be produced. In many cases, these other operators provide enough additional power that languages that are not regular can be described. So, in discussing "regular expressions" or "regexes", it is important to be clear exactly what definition is being used. In the rest of this book, we will use the definition that we presented in Section 6.1, with two additions to be described below, unless we clearly state that, for some particular purpose, we are going to use a different definition.

> The programming language Perl, for example, supports regular expression matching. (Appendix O) In Exercise 6.19, we'll consider the formal power of the Perl regular expression language.

Real applications need more than two or three characters. But we do not want to have to write expressions like:

$$(a \cup b \cup c \cup d \cup e \cup f \cup g \cup h \cup i \cup j \cup k \cup l \cup m \cup n \cup o \cup p \cup q \cup r \cup s \cup t \cup u \cup v \cup w \cup x \cup y \cup z).$$

It would be much more convenient to be able to write (a-z). So, in cases where there is an agreed upon collating sequence, we will use the shorthand $(\alpha - \omega)$ to mean $(\alpha \cup \ldots \cup \omega)$, where all the characters in the collating sequence between α and ω are included in the union.

EXAMPLE 6.14 Decimal Numbers

The following regular expression matches decimal encodings of numbers:

```
-? ([0-9]+(\.[0-9]*)? | \.[0-9]+)
```

EXAMPLE 6.14 *(Continued)*

In most standard regular expression dialects, the notation $\alpha?$ is equivalent to $(\alpha \cup \varepsilon)$. In other words, α is optional. So, in this example, the minus sign is optional. So is the decimal point.

Because the symbol . has a special meaning in most regular expression dialects, we must quote it when we want to match it as a literal character. The quote character in most regular expression dialects is \.

> Meaningful "words" in protein sequences are called motifs. They can be described with regular expressions. (K.3.2)

EXAMPLE 6.15 Legal Passwords

Consider the problem of determining whether a string is a legal password. Suppose that we require that all passwords meet the following requirements:

- A password must begin with a letter.
- A password may contain only letters, numbers, and the underscore character.
- A password must contain at least four characters and no more than eight characters.

The following regular expression describes the language of legal passwords. The line breaks have no significance. We have used them just to make the expression easier to read.

((a-z) ∪ (A-Z))
((a-z) ∪ (A-Z) ∪ (0-9) ∪ _)
((a-z) ∪ (A-Z) ∪ (0-9) ∪ _)
((a-z) ∪ (A-Z) ∪ (0-9) ∪ _)
((a-z) ∪ (A-Z) ∪ (0-9) ∪ _ ∪ ε)
((a-z) ∪ (A-Z) ∪ (0-9) ∪ _ ∪ ε)
((a-z) ∪ (A-Z) ∪ (0-9) ∪ _ ∪ ε)
((a-z) ∪ (A-Z) ∪ (0-9) ∪ _ ∪ ε).

While straightforward, the regular expression that we just wrote is a nuisance to write and not very easy to read. The problem is that, so far, we have only three ways to specify how many times a pattern must occur:

- α means that the pattern α must occur exactly once.
- α^* means that the pattern α may occur any number (including zero) of times.
- α^+ means that the pattern α may occur any positive number of times.

What we needed in the previous example was a way to specify how many times a pattern α should occur. We can do this with the following notations:

- $\alpha\{n, m\}$ means that the pattern α must occur at least n times and no more than m times.
- $\alpha\{n\}$ means that the pattern α must occur exactly n times.

Using this notation, we can rewrite the regular expression of Example 6.15 as:

$$((\texttt{a-z}) \cup (\texttt{A-Z}))((\texttt{a-z}) \cup (\texttt{A-Z}) \cup (\texttt{0-9}) \cup \_)\{3, 7\}.$$

EXAMPLE 6.16 IP Addresses

The following regular expression searches for Internet (IP) addresses:

```
([0-9]{1, 3} (\. [0-9]{1, 3}){3}).
```

> In XML, regular expressions are one way to define parts of new document types. (Q.1.2)

6.4 Manipulating and Simplifying Regular Expressions

The regular expressions (a ∪ b)* (a ∪ b)* and (a ∪ b)* define the same language. The second one is simpler than the first and thus easier to work with. In this section we discuss techniques for manipulating and simplifying regular expressions. All of these techniques are based on the equivalence of the languages that the regular expressions define. So we will say that, for two regular expressions α and β, $\alpha = \beta$ if $L(\alpha) = L(\beta)$.

We first consider identities that follow from the fact that the meaning of every regular expression is a language, which means that it is a set:

- Union is commutative: For any regular expressions α and β, $\alpha \cup \beta = \beta \cup \alpha$.
- Union is associative: For any regular expressions α, β, and γ, $(\alpha \cup \beta) \cup \gamma = \alpha \cup (\beta \cup \gamma)$.
- $\emptyset$ is the identity for union: For any regular expression α, $\alpha \cup \emptyset = \emptyset \cup \alpha = \alpha$.
- Union is idempotent: For any regular expression α, $\alpha \cup \alpha = \alpha$.
- Given any two sets A and B, if $B \subseteq A$, then $A \cup B = A$. So, for example, a* ∪ aa = a*, since $L(\texttt{aa}) \subseteq L(\texttt{a}^*)$.

Next we consider identities involving concatenation:

- Concatenation is associative: For any regular expressions α, β, and γ, $(\alpha\beta)\gamma = \alpha(\beta\gamma)$.

- ε is the identity for concatenation: For any regular expression α, $\alpha\,\varepsilon = \varepsilon\,\alpha = \alpha$.
- $\varnothing$ is a zero for concatenation: For any regular expression α, $\alpha\,\varnothing = \varnothing\,\alpha = \varnothing$.

Concatenation distributes over union:

- For any regular expressions α, β, and γ, $(\alpha \cup \beta)\gamma = (\alpha\gamma) \cup (\beta\gamma)$. Every string in either of these languages is composed of a first part followed by a second part. The first part must be drawn from $L(\alpha)$ or $L(\beta)$. The second part must be drawn from $L(\gamma)$.
- For any regular expressions α, β, and γ, $\gamma(\alpha \cup \beta) = (\gamma\alpha) \cup (\gamma\beta)$. (By a similar argument.)

Finally, we introduce identities involving Kleene star:

- $\varnothing^* = \varepsilon$.
- $\varepsilon^* = \varepsilon$.
- For any regular expression α, $(\alpha^*)^* = \alpha^*$. $L(\alpha^*)$ contains all and only the strings that are composed of zero or more strings from $L(\alpha)$, concatenated together. All of them are also in $L((\alpha^*)^*)$ since $L((\alpha^*)^*)$ contains, among other things, every individual string in $L(\alpha^*)$. No other strings are in $L((\alpha^*)^*)$ since it can contain only strings that are formed from concatenating together elements of $L(\alpha^*)$, which are in turn concatenations of strings from $L(\alpha)$.
- For any regular expression α, $\alpha^*\alpha^* = \alpha^*$. Every string in either of these languages is composed of zero or more strings from α concatenated together.
- More generally, for any regular expressions α and β, if $L(\alpha^*) \subseteq L(\beta^*)$ then $\alpha^*\beta^* = \beta^*$. For example:

$$\mathtt{a}^*\,(\mathtt{a} \cup \mathtt{b})^* = (\mathtt{a} \cup \mathtt{b})^*, \text{ since } L(\mathtt{a}^*) \subseteq L((\mathtt{a} \cup \mathtt{b})^*).$$

  α is redundant because any string it can generate and place at the beginning of a string to be generated by the combined expression $\alpha^*\beta^*$ can also be generated by β^*.
- Similarly, if $L(\beta^*) \subseteq L(\alpha^*)$ then $\alpha^*\beta^* = \alpha^*$.
- For any regular expressions α and β, $(\alpha \cup \beta)^* = (\alpha^*\beta^*)^*$. To form a string in either language, a generator must walk through the Kleene star loop zero or more times. Using the first expression, each time through the loop it chooses either a string from $L(\alpha)$ or a string from $L(\beta)$. That process can be copied using the second expression by picking exactly one string from $L(\alpha)$ and then ε from $L(\beta)$ or one string from $L(\beta)$ and then ε from $L(\alpha)$. Using the second expression, a generator can pick a sequence of strings from $L(\alpha)$ and then a sequence of strings from $L(\beta)$ each time through the loop. But that process can be copied using the first expression by simply selecting each element of the sequence one at a time on successive times through the loop.
- For any regular expressions α and β, if $L(\beta) \subseteq L(\alpha^*)$ then $(\alpha \cup \beta)^* = \alpha^*$. For example, $(\mathtt{a} \cup \varepsilon)^* = \mathtt{a}^*$, since $\{\varepsilon\} \subseteq L(\mathtt{a}^*)$. β is redundant since any string it can generate can also be generated by α^*.

EXAMPLE 6.17 Simplifying a Regular Expression

$$
\begin{array}{lll}
((a^* \cup \varnothing)^* \cup aa)(b \cup bb)^* b^* ((a \cup b)^* b^* \cup ab)^* = & /^* L(\varnothing) \subseteq L(a^*). \\
((a^*)^* \cup aa)(b \cup bb)^* b^* ((a \cup b)^* b^* \cup ab)^* = & \\
(a^* \cup aa)(b \cup bb)^* b^* ((a \cup b)^* b^* \cup ab)^* = & /^* L(aa) \subseteq L(a^*). \\
a^* (b \cup bb)^* b^* ((a \cup b)^* b^* \cup ab)^* = & /^* L(bb) \subseteq L(b^*). \\
a^* b^* b^* ((a \cup b)^* b^* \cup ab)^* = & \\
a^* b^* ((a \cup b)^* b^* \cup ab)^* = & /^* L(b^*) \subseteq L((a \cup b)^*). \\
a^* b^* ((a \cup b)^* \cup ab)^* = & /^* L(ab) \subseteq L((a \cup b)^*). \\
a^* b^* ((a \cup b)^*)^* = & \\
a^* b^* (a \cup b)^* = & /^* L(b^*) \subseteq L((a \cup b)^*). \\
a^* (a \cup b)^* = & /^* L(a^*) \subseteq L((a \cup b)^*). \\
(a \cup b)^* &
\end{array}
$$

Exercises

1. Describe in English, as briefly as possible, the language defined by each of these regular expressions:
   a. $(b \cup ba)(b \cup a)^* (ab \cup b)$.
   b. $(((a^*b^*)^*ab) \cup ((a^*b^*)^*ba))(b \cup a)^*$.
2. Write a regular expressions to describe each of the following languages:
   a. $\{w \in \{a, b\}^* :$ every a in w is immediately preceded and followed by b$\}$.
   b. $\{w \in \{a, b\}^* : w$ does not end in ba$\}$.
   c. $\{w \in \{0, 1\}^* : \exists y \in \{0, 1\}^* (|xy|$ is even$)\}$.
   d. $\{w \in \{0, 1\}^* : w$ corresponds to the binary encoding, without leading 0s, of natural numbers that are evenly divisible by 4$\}$.
   e. $\{w \in \{0, 1\}^* : w$ corresponds to the binary encoding, without leading 0s, of natural numbers that are powers of 4$\}$.
   f. $\{w \in \{0\text{-}9\}^* : w$ corresponds to the decimal encoding, without leading 0s, of an odd natural number$\}$.
   g. $\{w \in \{0, 1\}^* : w$ has 001 as a substring$\}$.
   h. $\{w \in \{0, 1\}^* : w$ does not have 001 as a substring$\}$.
   i. $\{w \in \{a, b\}^* : w$ has bba as a substring$\}$.
   j. $\{w \in \{a, b\}^* : w$ has both aa and bb as substrings$\}$.
   k. $\{w \in \{a, b\}^* : w$ has both aa and aba as substrings$\}$.
   l. $\{w \in \{a, b\}^* : w$ contains at least two b's that are not followed by an a$\}$.
   m. $\{w \in \{0, 1\}^* : w$ has at most one pair of consecutive 0s and at most one pair of consecutive 1s$\}$.

n. $\{w \in \{0,1\}^* :$ none of the prefixes of w ends in $0\}$.

o. $\{w \in \{a,b\}^* : \#_a(w) \equiv_3 0\}$.

p. $\{w \in \{a,b\}^* : \#_a(w) \leq 3\}$.

q. $\{w \in \{a,b\}^* : w$ contains exactly two occurrences of the substring aa$\}$.

r. $\{w \in \{a,b\}^* : w$ contains no more than two occurrences of the substring aa$\}$.

s. $\{w \in \{a,b\}^* - L\}$, where $L = \{w \in \{a,b\}^* : w$ contains bba as a substring$\}$.

t. $\{w \in \{0,1\}^* :$ every odd length string in L begins with 11$\}$.

u. $\{w \in \{0\text{-}9\}^* : w$ represents the decimal encoding of an odd natural number without leading 0s.

v. $L_1 - L_2$, where $L_1 =$ a*b*c* and $L_2 =$ c*b*a*.

w. The set of legal United States zip codes 📪.

x. The set of strings that correspond to domestic telephone numbers in your country.

3. Simplify each of the following regular expressions:

a. (a ∪ b)* (a ∪ ε) b*.

b. (∅* ∪ b) b*.

c. (a ∪ b)*a* ∪ b.

d. ((a ∪ b)*)*.

e. ((a ∪ b)$^+$)*.

f. a ((a ∪ b)(b ∪ a))* ∪ a ((a ∪ b) a)* ∪ a ((b ∪ a) b)*.

4. For each of the following expressions E, answer the following three questions and prove your answer:

i. Is E a regular expression?

ii. If E is a regular expression, give a simpler regular expression.

iii. Does E describe a regular language?

a. ((a ∪ b) ∪ (ab))*.

b. (a$^+$ a^{n}b^n).

c. ((ab)*∅).

d. (((ab) ∪ c)* ∩ (b ∪ c*)).

e. (∅* ∪ (bb*)).

5. Let $L = \{a^n b^n : 0 \leq n \leq 4\}$.

a. Show a regular expression for L.

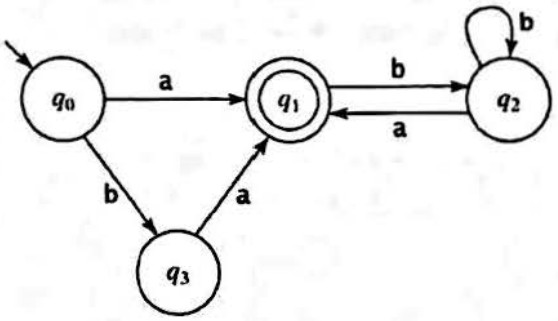
b. Show an FSM that accepts L.

6. Let $L = \{w \in \{1,2\}^* :$ for all prefixes p of w, if $|p| > 0$ and $|p|$ is even, then the last character of p is $1\}$.

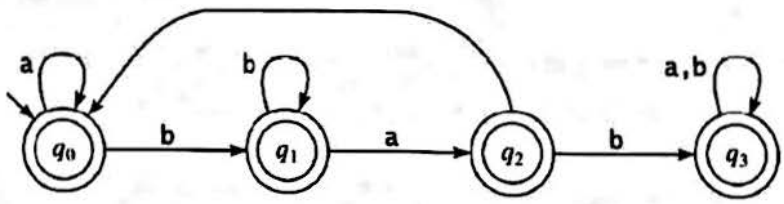
a. Write a regular expression for L.

b. Show an FSM that accepts L.

7. Use the algorithm presented in the proof of Kleene's Theorem to construct an FSM to accept the language generated by each of the following regular expressions:
   a. (b(b ∪ ε)b)*.
   b. bab ∪ a*.
8. Let L be the language accepted by the following finite state machine:



   Indicate, for each of the following regular expressions, whether it correctly describes L:
   a. (a ∪ ba)bb*a.
   b. (ε ∪ b)a(bb*a)*.
   c. ba ∪ ab*a.
   d. (a ∪ ba)(bb*a)*.
9. Consider the following FSM M:



   a. Show a regular expression for $L(M)$.
   b. Describe $L(M)$ in English.
10. Consider the FSM M of Example 5.3. Use *fsmtoregexheuristic* to construct a regular expression that describes $L(M)$.
11. Consider the FSM M of Example 6.9. Apply *fsmtoregex* to M and show the regular expression that results.
12. Consider the FSM M of Example 6.8. Apply *fsmtoregex* to M and show the regular expression that results. (Hint: This one is exceedingly tedious, but it can be done.)
13. Show a possibly nondeterministic FSM to accept the language defined by each of the following regular expressions:
   a. (((a ∪ ba) b ∪ aa)*.
   b. (b ∪ε)(ab)*(a ∪ ε).
   c. (babb* ∪ a)*.

d. (ba ∪ ((a ∪ bb) a*b)).

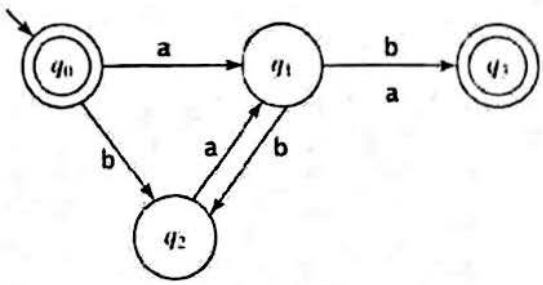
e. (a ∪ b)* aa (b ∪ aa) bb (a ∪ b)*.

14. Show a DFSM to accept the language defined by each of the following regular expressions:

a. (aba ∪ aabaa)*.

b. (ab)*(aab)*.

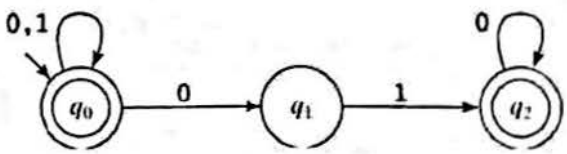
15. Consider the following DFSM M:



a. Write a regular expression that describes $L(M)$.

b. Show a DFSM that accepts $\neg L\ (M)$.

16. Given the following DFSM M, write a regular expression that describes $\neg L\ (M)$:



17. Add the keyword able to the set in Example 6.13 and show the FSM that will be built by *buildkeywordFSM* from the expanded keyword set.

18. Let Σ = {a, b}. Let L = {ε, a, b}. Let R be a relation defined on Σ* as follows: ∀xy (xRy iff $y = xb$). Let R' be the reflexive, transitive closure of R. Let L' = {x : ∃$y \in L$ ($yR'x$)}. Write a regular expression for L'.

19. In Appendix O we summarize the main features of the regular expression language in Perl. What feature of that regular expression language makes it possible to write regular expressions that describe languages that aren't regular?

20. For each of the following statements, state whether it is *True* or *False*. Prove your answer.

a. (ab)*a = a(ba)*.

b. (a ∪ b)* b (a ∪ b)* = a* b (a ∪ b)*.

c. (a ∪ b)* b (a ∪ b)* ∪ (a ∪ b)* a (a ∪ b)* = (a ∪ b)*.

d. (a ∪ b)* b (a ∪ b)* ∪ (a ∪ b)* a (a ∪ b)* = (a ∪ b)+.

e. (a ∪ b)* b a (a ∪ b)* ∪ a*b* = (a ∪ b)*.

f. a* b (a ∪ b)* = (a ∪ b)* b (a ∪ b)*.

g. If α and β are any two regular expressions, then $(\alpha \cup \beta)^* = \alpha\,(\beta\alpha \cup \alpha)$.

h. If α and β are any two regular expressions, then $(\alpha\beta)^*\alpha = \alpha\,(\beta\alpha)^*$.