FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

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 $\varnothing$, then the concatenation will also be equal to $\varnothing$.
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 $\varnothing$, then $L(\alpha^+)$ is also equal to $\varnothing$. 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 $L(\varnothing^*) = \{w : w$ is formed by concatenating together zero or more strings from $\varnothing\}$. But how many ways are there to concatenate together zero or more strings from $\varnothing$? 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 $L(\varnothing^*) = \{\varepsilon\}$. Thus, whenever we would like to write ε, we could instead write $\varnothing^*$. 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 $\varnothing^*$ 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

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

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 $L((\mathsf{a} \cup \mathsf{b})^*\mathsf{b})$.

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

EXAMPLE 6.2 Another Simple Regular Expression

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

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 $L(\mathtt{a}^*) = \{w : w \text{ is composed of zero or more a's}\}$. 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 $M_2 = (K_2, \Sigma, \delta_2, s_2, A_2)$.

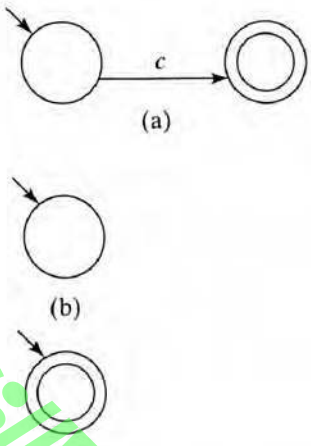


FIGURE 6.1 FSMs for primitive regular expressions.

- 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-transition. 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:

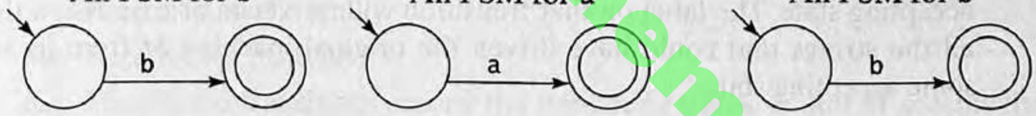
Construct an FSM as described above.

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)

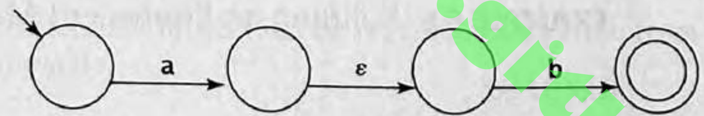
EXAMPLE 6.5 Building an FSM from a Regular Expression

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

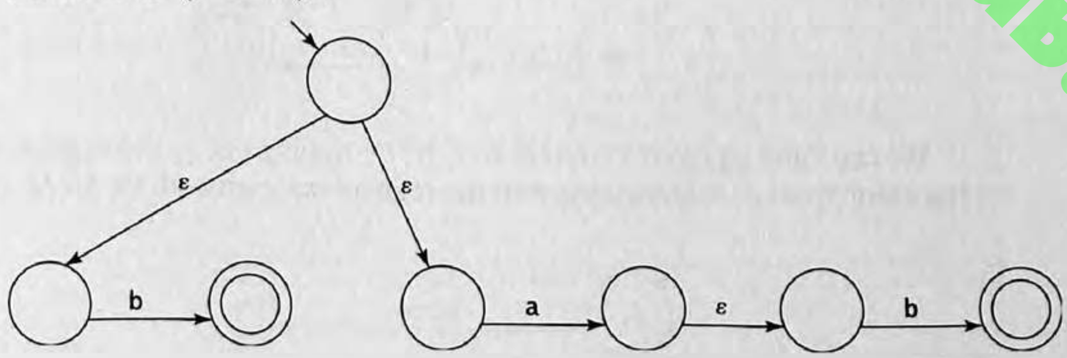
An FSM for b   An FSM for a   An FSM for b



An FSM for ab:

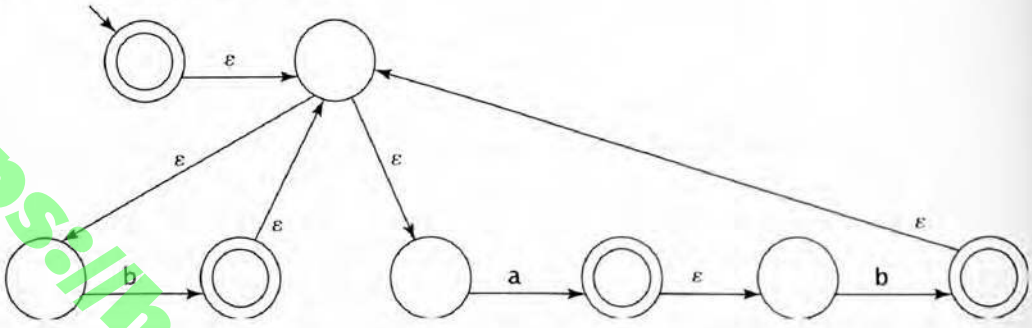


An FSM for (b ∪ ab):



EXAMPLE 6.5 *(Continued)*

An FSM for (b ∪ ab)*

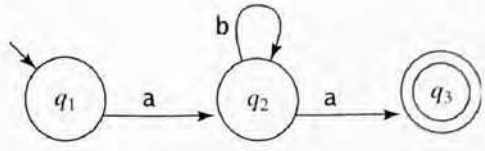


6.2.2 Building a Regular Expression from an FSM

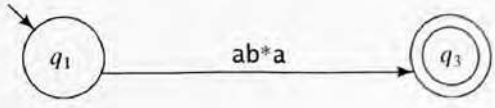
Next we must show that it is possible to go the other direction, namely to build, from an FSM, a corresponding regular expression. The idea behind the algorithm that we are about to present is the following: Instead of limiting the labels on the transitions of an FSM to a single character or ε, we will 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.

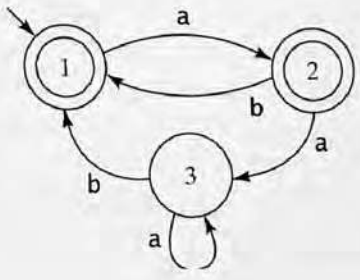
The following algorithm creates a regular expression that defines $L(M)$, provided that step 6 can be executed correctly:

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.

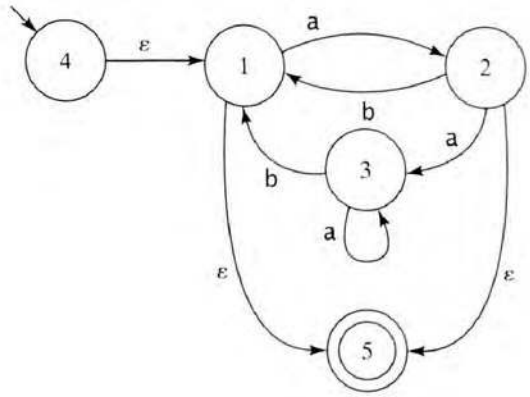
EXAMPLE 6.7 Building a Regular Expression from an FSM

Let M be:

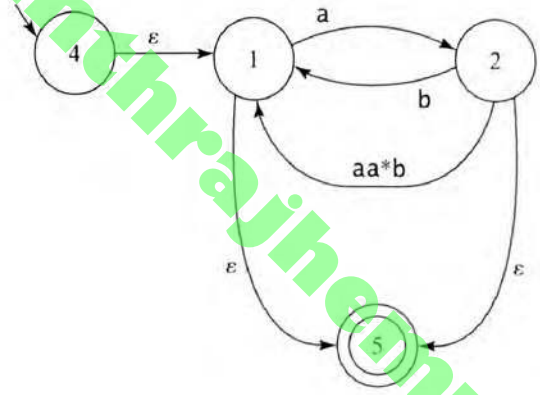


EXAMPLE 6.7 *(Continued)*

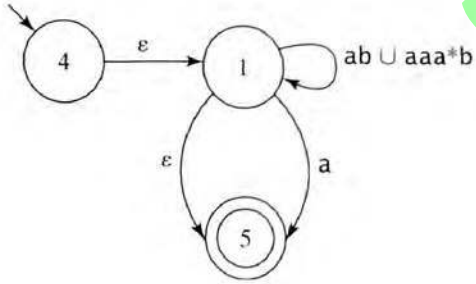
Create a new start state and a new accepting state and link them to M:



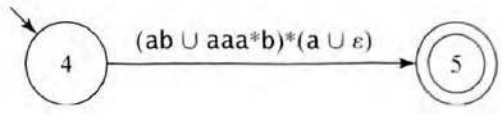
Remove state 3:



Remove state 2:

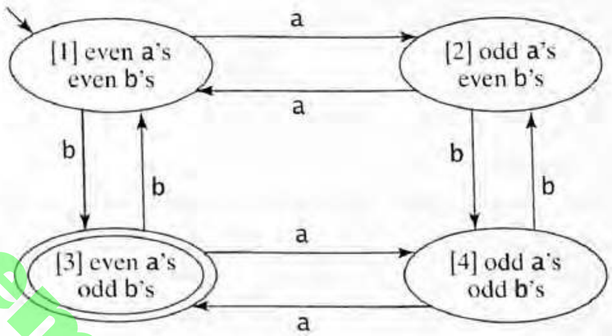


Remove state 1:



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

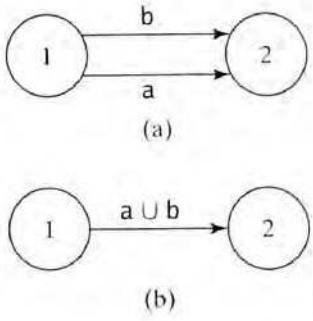


FIGURE 6.2 Collapsing multiple transitions into one.

$\{c_1, c_2, \ldots, c_n\}$, 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).

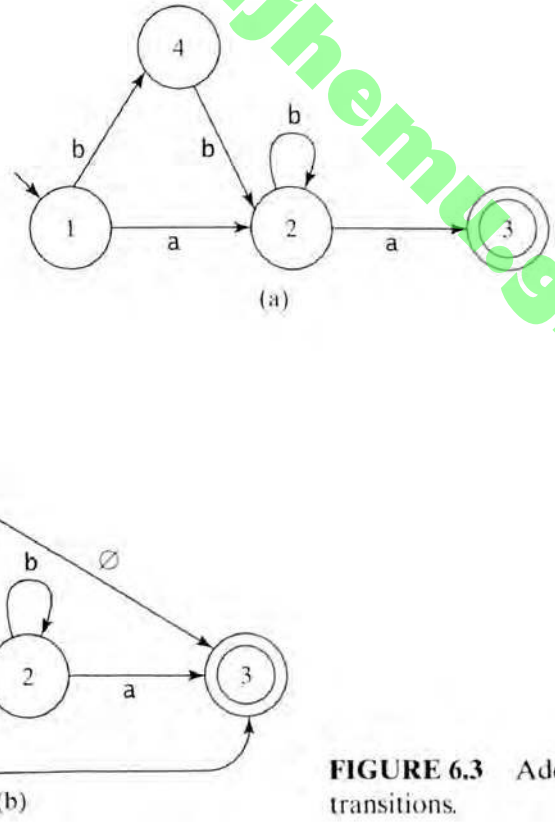


FIGURE 6.3 Adding all the required transitions.

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:

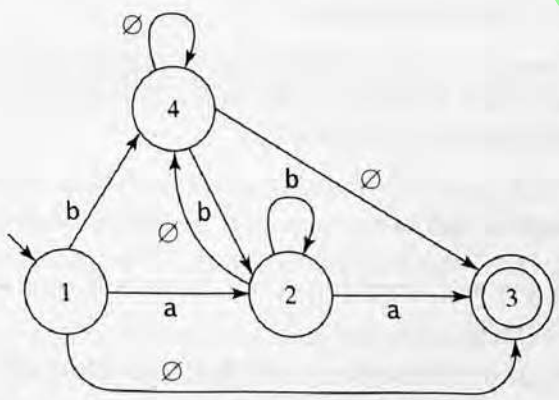
$R(p, q)$	/* Go directly from p to q,
$\cup$	/* or
$R(p, rip)$	/* go from p to *rip*, then
$R(rip, rip)^*$	/* go from *rip* back to itself any number of times, then
$R(rip, q)$	/* go from *rip* to q.

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:

$$\begin{aligned} R'(1, 3) &= R(1, 3) \cup R(1, rip)R(rip, rip)^*R(rip, 3). \\ &= R(1, 3) \cup R(1, 2)R(2, 2)^*R(2, 3). \end{aligned}$$

EXAMPLE 6.9 *(Continued)*

$$= \varnothing \cup \text{a} \quad \text{b}^* \quad \text{a}.$$
$$= \text{ab}^*\text{a}.$$

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 $L = \{w \in \{a, b\}^* : \text{there is no more than one b}\}$. 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 $L = \{w \in \{a, b\}^* : \text{no two consecutive letters are the same}\}$. 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:

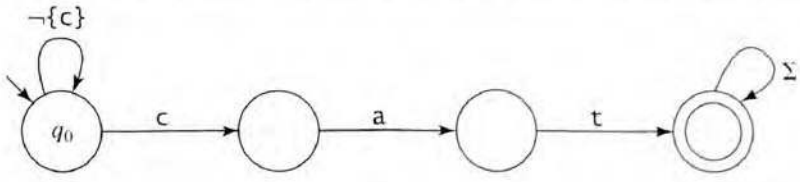
buildkeywordFSM(K: set of keywords) =

1. Create a start state q_0.
2. For each element k of K do:

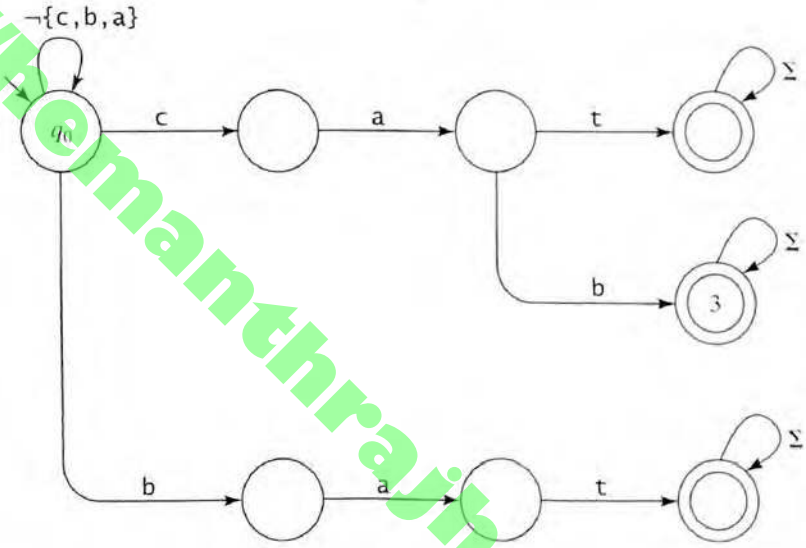
   Create a branch corresponding to k.
3. Create a set of transitions that describe what to do when a branch dies, either because its complete pattern has been found or because the next character is not the correct one to continue the pattern.
4. Make the states at the ends of each branch accepting.

EXAMPLE 6.13 Recognizing a Set of Keywords

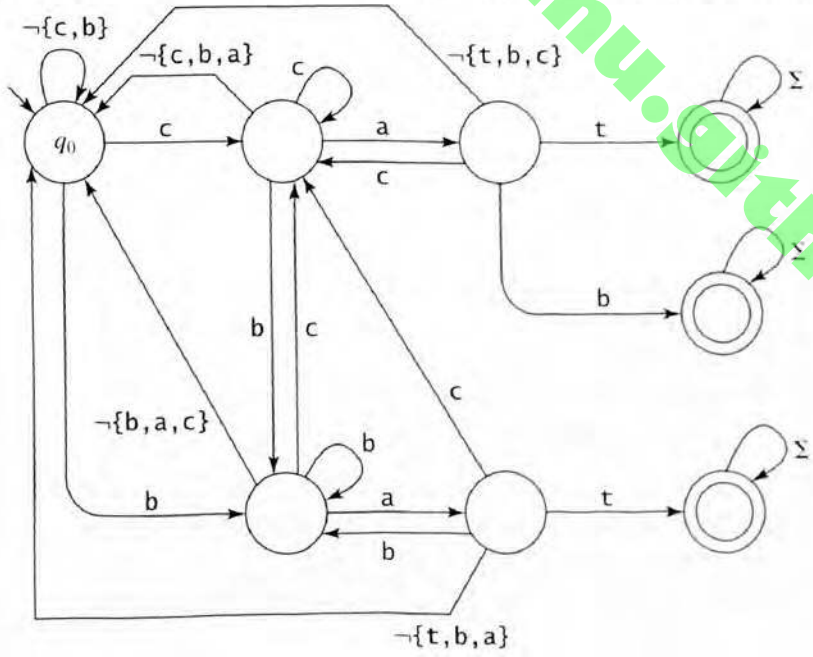
Consider the set of keywords {cat, bat, cab}. We can use *buildkeywordFSM* to build a DFSM to accept strings that contain at least one of these keywords. We begin by creating a start state and then a path to accept the first keyword, cat:



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 ∪ b ∪ c ∪ d ∪ e ∪ f ∪ g ∪ h ∪ i ∪ j ∪ k ∪ l ∪ m ∪ n ∪ o ∪ p ∪ q ∪ r ∪ s ∪ t ∪ u ∪ v ∪ w ∪ x ∪ y ∪ 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 $(\texttt{a} \cup \texttt{b})^* (\texttt{a} \cup \texttt{b})^*$ and $(\texttt{a} \cup \texttt{b})^*$ 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, $(\alpha \cup \beta) \cup \gamma = \alpha \cup (\beta \cup \gamma)$.
- $\varnothing$ is the identity for union: For any regular expression α, $\alpha \cup \varnothing = \varnothing \cup \alpha = \alpha$.
- 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, $\texttt{a}^* \cup \texttt{aa} = \texttt{a}^*$, 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:

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

  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, $(\mathsf{a} \cup \varepsilon)^* = \mathsf{a}^*$, since $\{\varepsilon\} \subseteq L(\mathsf{a}^*)$. β is redundant since any string it can generate can also be generated by α^*.

EXAMPLE 6.17 Simplifying a Regular Expression

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

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.

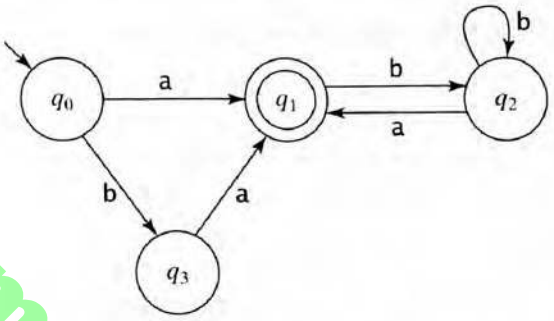
b. Show an FSM that accepts L.

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 $1\}$.

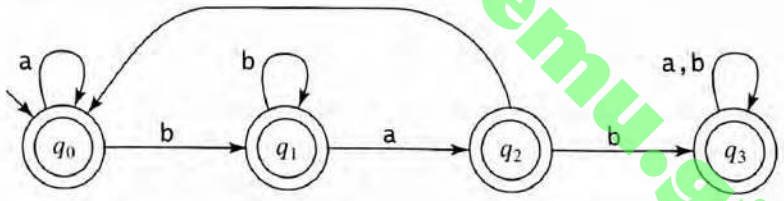
a. Write a regular expression for L.

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)).

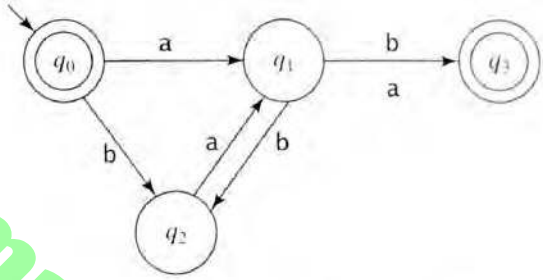
e. (a ∪ b)* aa (b ∪ aa) bb (a ∪ b)*.

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

a. (aba ∪ aabaa)*.

b. (ab)*(aab)*.

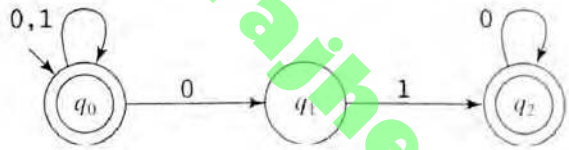
15. Consider the following DFSM M:



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 $(\alpha\beta)^*\alpha = \alpha\,(\beta\alpha)^*$.

CHAPTER 7

Regular Grammars

So far, we have considered two equivalent ways to describe exactly the class of regular languages:

- Finite state machines.
- Regular expressions.

We now introduce a third:

- Regular grammars (sometimes also called right linear grammars).

7.1 Definition of a Regular Grammar

A ***regular grammar*** G is a quadruple (V, Σ, R, S), where:

- V is the rule alphabet, which contains nonterminals (symbols that are used in the grammar but that do not appear in strings in the language) and terminals (symbols that can appear in strings generated by G),
- Σ (the set of terminals) is a subset of V,
- R (the set of rules) is a finite set of rules of the form $X \rightarrow Y$, and
- S (the start symbol) is a nonterminal.

In a regular grammar, all rules in R must:

- have a left-hand side that is a single nonterminal, and
- have a right-hand side that is ε or a single terminal or a single terminal followed by a single nonterminal.

So $S \rightarrow \mathsf{a}$, $S \rightarrow \varepsilon$, and $T \rightarrow \mathsf{a}S$ are legal rules in a regular grammar. $S \rightarrow \mathsf{a}S\mathsf{a}$ and $\mathsf{a}S\mathsf{a} \rightarrow T$ are not legal rules in a regular grammar.

We will formalize the notion of a grammar generating a language in Chapter 11, when we introduce a more powerful grammatical framework, the context-free grammar. For now, an informal notion will do. The language generated by a grammar $G = (V, \Sigma, R, S)$, denoted $L(G)$, is the set of all strings w in Σ^* such that it is possible to start with S, apply some finite set of rules in R, and derive w.

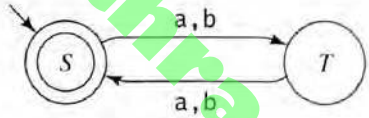
To make writing grammars easy, we will adopt the convention that, unless otherwise specified, the start symbol of any grammar G will be the symbol on the left-hand side of the first rule in R_G.

EXAMPLE 7.1 Even Length Strings

Let $L = \{w \in \{\mathsf{a}, \mathsf{b}\}^*: |w| \text{ is even}\}$. The following regular expression defines L:

$$((\mathsf{aa}) \cup (\mathsf{ab}) \cup (\mathsf{ba}) \cup (\mathsf{bb}))^*.$$

The following DFSM M accepts L:



The following regular grammar G also defines L:

$$S \rightarrow \varepsilon$$
$$S \rightarrow \mathsf{a}T$$
$$S \rightarrow \mathsf{b}T$$
$$T \rightarrow \mathsf{a}S$$
$$T \rightarrow \mathsf{b}S$$

In G, the job of the nonterminal S is to generate an even length string. It does this either by generating the empty string or by generating a single character and then creating T. The job of T is to generate an odd length string. It does this by generating a single character and then creating S. S generates ε, the shortest possible even length string. So, if T can be shown to generate all and only the odd length strings, we can show that S generates all and only the remaining even length strings. T generates every string whose length is one greater than the length of some string S generates. So, if S generates all and only the even length strings, then T generates all and only the other odd length strings.

Notice the clear correspondence between M and G, which we have highlighted by naming M's states S and T. Even length strings drive M to state S. Even length strings are generated by G starting with S. Odd length strings drive M to state T. Odd length strings are generated by G starting with T.

7.2 Regular Grammars and Regular Languages

THEOREM 7.1 Regular Grammars Define Exactly the Regular Languages

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

Proof: We first show that any language that can be defined with a regular grammar can be accepted by some FSM and so is regular. Then we must show that every regular language (i.e., every language that can be accepted by some FSM) can be defined with a regular grammar. Both proofs are by construction.

Regular grammar* → *FSM: The following algorithm constructs an FSM M from a regular grammar $G = (V, \Sigma, R, S)$ and assures that $L(M) = L(G)$:

grammartofsm(G: regular grammar) =

1. Create in M a separate state for each nonterminal in V.
2. Make the state corresponding to S the start state.
3. If there are any rules in R of the form $X \rightarrow w$, for some $w \in \Sigma$, then create an additional state labeled #.
4. For each rule of the form $X \rightarrow wY$, add a transition from X to Y labeled w.
5. For each rule of the form $X \rightarrow w$, add a transition from X to # labeled w.
6. For each rule of the form $X \rightarrow \varepsilon$, mark state X as accepting.
7. Mark state # as accepting.
8. If M is incomplete (i.e., there are some (state, input) pairs for which no transition is defined), M requires a dead state. Add a new state D. For every (q, i) pair for which no transition has already been defined, create a transition from q to D labeled i. For every i in Σ, create a transition from D to D labeled i.

***FSM* → *Regular grammar*:** The construction is effectively the reverse of the one we just did. We leave this step as an exercise.

EXAMPLE 7.2 Strings that End with aaaa

Let $L = \{w \in \{\mathrm{a}, \mathrm{b}\}^*: w$ ends with the pattern aaaa$\}$. Alternatively, $L = (\mathrm{a} \cup \mathrm{b})^*$ aaaa. The following regular grammar defines L:

$S \rightarrow \mathrm{a}S$	/* An arbitrary number of a's and b's can be generated
$S \rightarrow \mathrm{b}S$	before the pattern starts.
$S \rightarrow \mathrm{a}B$	/* Generate the first a of the pattern.

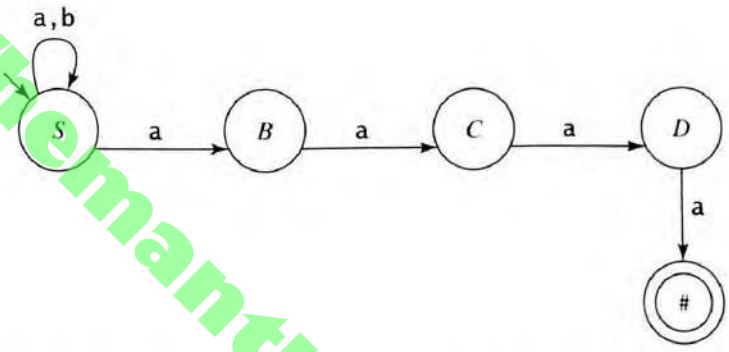
EXAMPLE 7.2 *(Continued)*

$B \rightarrow$ aC /* Generate the second a of the pattern.

$C \rightarrow$ aD /* Generate the third a of the pattern.

$D \rightarrow$ a /* Generate the last a of the pattern and quit.

Applying *grammartofsm* to this grammar, we get, omitting the dead state:



Notice that the machine that *grammartofsm* builds is not necessarily deterministic.

EXAMPLE 7.3 The Missing Letter Language

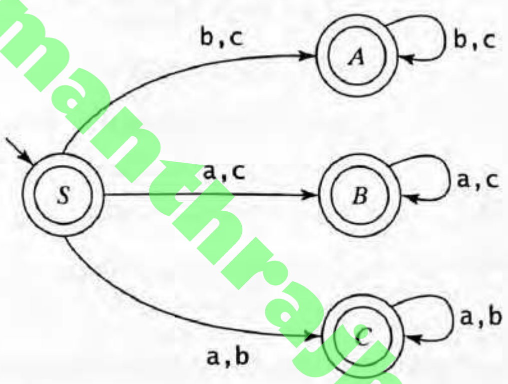
Let $\Sigma = \{$a, b, c$\}$. Let L be $L_{Missing} = \{w$: there is a symbol $a_i \in \Sigma$ not appearing in $w\}$, which we defined in Example 5.12. The following grammar G generates $L_{Missing}$:

$S \rightarrow \varepsilon$
$S \rightarrow$ aB
$S \rightarrow$ aC
$S \rightarrow$ bA
$S \rightarrow$ bC
$S \rightarrow$ cA
$S \rightarrow$ cB
$A \rightarrow$ bA
$A \rightarrow$ cA
$A \rightarrow \varepsilon$
$B \rightarrow$ aB
$B \rightarrow$ cB

$B \rightarrow \varepsilon$
$C \rightarrow \mathsf{a}C$
$C \rightarrow \mathsf{b}C$
$C \rightarrow \varepsilon$

The job of S is to generate some string in $L_{Missing}$. It does that by choosing a first character of the string and then choosing which other character will be missing. The job of A is to generate all strings that do not contain any a's. The job of B is to generate all strings that do not contain any b's. And the job of C is to generate all strings that do not contain any c's.

If we apply *grammartofsm* to G, we get M =



M is identical to the NDFSM we had previously built for $L_{Missing}$ except that it waits to guess whether to go to A, B or C until it has seen its first input character.

Our proof of the first half of Theorem 7.1 clearly describes the correspondence between the nonterminals in a regular grammar and the states in a corresponding FSM. This correspondence suggests a natural way to think about the design of a regular grammar. The nonterminals in such a grammar need to "remember" the relevant state of a left-to-right analysis of a string.

EXAMPLE 7.4 Satisfying Multiple Criteria

Let $L = \{w \in \{\mathsf{a}, \mathsf{b}\}^*: w$ contains an odd number of a's and w ends in a$\}$. We can write a regular grammar G that defines L. G will contain four nonterminals, each with a unique function (corresponding to the states of a simple FSM that accepts L). So, in any derived string, if the remaining nonterminal is:

- S, then the number of a's so far is even. We don't have worry about whether the string ends in a since, to derive a string in L, it will be necessary to generate at least one more a anyway.

EXAMPLE 7.4 (*Continued*)

- T, then the number of a's so far is odd and the derived string ends in a.
- X, then the number of a's so far is odd and the derived string does not end in a.

Since only T captures the situation in which the number of a's so far is odd and the derived string ends in a, T is the only nonterminal that can generate ε. G contains the following rules:

$S \rightarrow$ bS	/* Initial b's don't matter.
$S \rightarrow$ aT	/* After this, the number of a's is odd and the generated string ends in a.
$T \rightarrow \varepsilon$	/* Since the number of a's is odd, and the string ends in a, it's okay to quit.
$T \rightarrow$ aS	/* After this, the number of a's will be even again.
$T \rightarrow$ bX	/* After this, the number of a's is still odd but the generated string no longer ends in a.
$X \rightarrow$ aS	/* After this, the number of a's will be even.
$X \rightarrow$ bX	/* After this, the number of a's is still odd and the generated string still does not end in a.

To see how this grammar works, we can watch it generate the string baaba:

$S \Rightarrow$ bS	/* Still an even number of a's.
$\Rightarrow$ ba$T$	/* Now an odd number of a's and ends in a. The process could quit now since the derived string, ba, is in L.
$\Rightarrow$ baa$S$	/* Back to having an even number of a's, so it doesn't matter what the last character is.
$\Rightarrow$ baab$S$	/* Still even a's.
$\Rightarrow$ baaba$T$	/* Now an odd number of a's and ends in a. The process can quit, by applying the rule $T \rightarrow \varepsilon$.
$\Rightarrow$ baaba	

So now we know that regular grammars define exactly the regular languages. But regular grammars are not often used in practice. The reason, though, is not that they couldn't be. It is simply that there is something better. Given some regular language L, the structure of a reasonable FSM for L very closely mirrors the structure of a reasonable regular grammar for it. And FSMs are easier to work with. In addition, there exist regular expressions. In Parts III and IV, as we move outward to larger classes of languages, there will no longer exist a technique like regular expressions.

At that point, particularly as we are considering the context-free languages, we will see that grammars are a very important and useful way to define languages.

Exercises

1. Show a regular grammar for each of the following languages:
   a. $\{w \in \{a, b\}^*: w$ contains an even number of a's and an odd number of b's$\}$.
   b. $\{w \in \{a, b\}^*: w$ does not end in aa$\}$.
   c. $\{w \in \{a, b\}^*: w$ contains the substring abb$\}$.
   d. $\{w \in \{a, b\}^*:$ if w contains the substring aa then $|w|$ is odd$\}$.
   e. $\{w \in \{a, b\}^*: w$ does not contain the substring aabb$\}$.
2. Consider the following regular grammar G:
   $S \rightarrow aT$
   $T \rightarrow bT$
   $T \rightarrow a$
   $T \rightarrow aW$
   $W \rightarrow \varepsilon$
   $W \rightarrow aT$
   a. Write a regular expression that generates $L(G)$.
   b. Use *grammartofsm* to generate an FSM M that accepts $L(G)$.
3. Consider again the FSM M shown in Exercise 5.1. Show a regular grammar that generates $L(M)$.
4. Show by construction that, for every FSM M there exists a regular grammar G such that $L(G) = L(M)$.
5. Let $L = \{w \in \{a, b\}^*:$ every a in w is immediately followed by at least one b$\}$.
   a. Write a regular expression that describes L.
   b. Write a regular grammar that generates L.
   c. Construct an FSM that accepts L.

CHAPTER 8

Regular and Nonregular Languages

The language a^*b^* is regular. The language $A^nB^n = \{a^nb^n : n \geq 0\}$ is not regular (intuitively because it is not possible, given some finite number of states, to count an arbitrary number of a's and then compare that count to the number of b's). The language $\{w \in \{a, b\}^* :$ every a is immediately followed by a b$\}$ is regular. The similar sounding language $\{w \in \{a, b\}^* :$ every a has a matching b somewhere and no b matches more than one a$\}$ is not regular (again because it is now necessary to count the a's and make sure that the number of b's is at least as great as the number of a's.)

Given a new language L, how can we know whether or not it is regular? In this chapter, we present a collection of techniques that can be used to answer that question.

8.1 How Many Regular Languages Are There?

First, we observe that there are *many* more nonregular languages than there are regular ones:

THEOREM 8.1 The Regular Languages are Countably Infinite

Theorem: There is a countably infinite number of regular languages.

Proof: We can lexicographically enumerate all the syntactically legal DFSMs with input alphabet Σ. Every regular language is accepted by at least one of them. So there cannot be more regular languages than there are DFSMs. Thus there are at most a countably infinite number of regular languages. There is not a one-to-one relationship between regular languages and DFSMs since there is an infinite number of machines that accept any given language. But the number of regular languages is infinite because it includes the following infinite set of languages:

$$\{a\}, \{aa\}, \{aaa\}, \{aaaa\}, \{aaaaa\}, \{aaaaaa\}, \ldots$$

But, by Theorem 2.3, there is an uncountably infinite number of languages over any nonempty alphabet So there are many more nonregular languages than there are regular ones.

8.2 Showing That a Language Is Regular

But many languages *are* regular. How can we know which ones? We start with the simplest cases.

THEOREM 8.2 The Finite Languages

Theorem: Every finite language is regular.

Proof: If L is the empty set, then it is defined by the regular expression $\emptyset$ and so is regular. If it is any finite language composed of the strings $s_1, s_2, \ldots s_n$ for some positive integer n, then it is defined by the regular expression:

$$s_1 \cup s_2 \cup \cdots \cup s_n$$

So it too is regular.

EXAMPLE 8.1 The Intersection of Two Infinite Languages

Let $L = L_1 \cap L_2$, where $L_1 = \{a^n b^n : n \geq 0\}$ and $L_2 = \{b^n a^n : n \geq 0\}$. As we will soon be able to prove, neither L_1 nor L_2 is regular. But L is. $L = \{\varepsilon\}$, which is finite.

EXAMPLE 8.2 A Finite Language We May Not Be Able to Write Down

Let $L = \{w \in \{0 - 9\}^* : w$ is the social security number of a living US resident$\}$. L is regular because it is finite. It doesn't matter that no individual or organization happens, at any given instant, to know what strings are in L.

Note, however, that although the language in Example 8.2 is formally regular, the techniques that we have described for recognizing regular languages would not be very useful in building a program to check for a valid social security number. Regular expressions are most useful when the elements of L match one or more patterns. FSMs are most useful when the elements of L share some simple structural properties. Other techniques, like hash tables, are better suited to handling finite languages whose elements are chosen by our world, rather than by rule.

EXAMPLE 8.3 Santa Clause, God, and the History of the Americas

Let:

- $L_1 = \{w \in \{0 - 9\}^* : w$ is the social security number of the current US president$\}$.
- $L_2 = \{1$ if Santa Claus exists and 0 otherwise$\}$.
- $L_3 = \{1$ if God exists and 0 otherwise$\}$.
- $L_4 = \{1$ if there were people in North America more than 10,000 years ago and 0 otherwise$\}$.
- $L_5 = \{1$ if there were people in North America more than 15,000 years ago and 0 otherwise$\}$.
- $L_6 = \{w \in \{0 - 9\}^+ : w$ is the decimal representation, without leading 0's, of a prime Fermat number$\}$.

L_1 is clearly finite, and thus regular. There exists a simple FSM to accept it, even though none of us happens to know what that FSM is. L_2 and L_3 are perhaps a little less clear, but that is because the meanings of "Santa Claus" and "God" are less clear. Pick a definition for either of them. Then something that satisfies that definition either does or does not exist. So either the simple FSM that accepts $\{0\}$ and nothing else or the simple FSM that accepts $\{1\}$ and nothing else accepts L_2. And one of them (possibly the same one, possibly the other one) accepts L_3. L_4 is clear. It is the set $\{1\}$. L_5 is also finite, and thus regular. Either there were people in North America by 15,000 years ago or there were not, although the currently available fossil evidence 🖳 is unclear as to which. So we (collectively) just don't know yet which machine to build. L_6 is similar, although this time what is lacking is mathematics, as opposed to fossils. Recall from Section 4.1 that the Fermat numbers are defined by

$$F_n = 2^{2^n} + 1, n \geq 0.$$

The first five elements of F_n are $\{3, 5, 17, 257, 65{,}537\}$. All of them are prime. It appears likely 🖳 that no other Fermat numbers are prime. If that is true, then L_6 is finite and thus regular. If it turns out that the set of Fermat numbers is infinite, then it is almost surely not regular.

Not every regular language is computationally tractable. Consider the Towers of Hanoi language. (P. 2)

But, of course, most interesting regular languages are infinite. So far, we've developed four techniques for showing that a (finite or infinite) language L is regular:

- Exhibit a regular expression for L.
- Exhibit an FSM for L.

- Show that the number of equivalence classes of $\approx_L$ is finite.
- Exhibit a regular grammar for L.

8.3 Some Important Closure Properties of Regular Languages

We now consider one final technique, which allows us, when analyzing complex languages, to exploit the other techniques as subroutines. The regular languages are closed under many common and useful operations. So, if we wish to show that some language L is regular and we can show that L can be constructed from other regular languages using those operations, then L must also be regular.

THEOREM 8.3 Closure under Union, Concatenation and Kleene Star

Theorem: The regular languages are closed under union, concatenation, and Kleene star.

Proof: By the same constructions that were used in the proof of Kleene's theorem.

THEOREM 8.4 Closure under Complement, Intersection, Difference, Reverse and Letter Substitution

Theorem: The regular languages are closed under complement, intersection, difference, reverse, and letter substitution.

Proof:

- The regular languages are closed under complement. If L_1 is regular, then there exists a DFSM $M_1 = (K, \Sigma, \delta, s, A)$ that accepts it. The DFSM $M_2 = (K, \Sigma, \delta, s, K - A)$, namely M_1 with accepting and nonaccepting states swapped, accepts $\neg(L(M_1))$ because it rejects all strings that M_1 accepts and rejects all strings that M_1 accepts.

  Given an arbitrary (possibly nondeterministic) FSM $M_1 = (K_1, \Sigma, \Delta_1, s_1, A_1)$, we can construct a DFSM $M_2 = (K_2, \Sigma, \delta_2, s_2, A_2)$ such that $L(M_2) = \neg(L(M_1))$. We do so as follows: From M_1, construct an equivalent deterministic FSM $M' = (K_{M'}, \Sigma, \delta_{M'}, s_{M'}, A_{M'})$, using the algorithm *ndfsmtodfsm*, presented in the proof of Theorem 5.3. (If M_1 is already deterministic, $M' = M_1$.) M' must be stated completely, so if it is described with an implied dead state, add the dead state and all required transitions to it. Begin building M_2 by setting it equal to M'. Then swap the accepting and the nonaccepting states. So $M_2 = (K_{M'}, \Sigma, \delta_{M'}, s_{M'}, K_{M'} - A_{M'})$.

- The regular languages are closed under intersection. We note that:

$$L(M_1) \cap L(M_2) = \neg(\neg L(M_1) \cup \neg L(M_2)).$$

  We have already shown that the regular languages are closed under both complement and union. Thus they are also closed under intersection.

It is also possible to prove this claim by construction of an FSM that accepts $L(M_1) \cap L(M_2)$. We leave that proof as an exercise.

- The regular languages are closed under set difference (subtraction). We note that:

$$L(M_1) - L(M_2) = L(M_1) \cap \neg L(M_2).$$

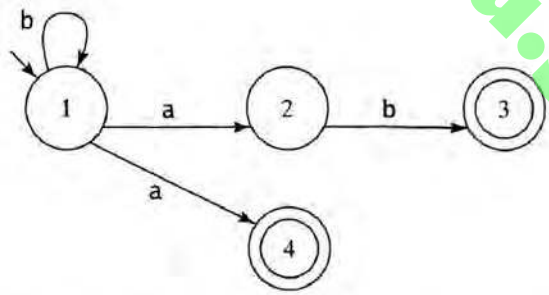
We have already shown that the regular languages are closed under both complement and intersection. Thus they are also closed under set difference.

This claim too can also be proved by construction, which we leave as an exercise.

- The regular languages are closed under reverse. Recall that $L^R = \{w \in \Sigma^* : w = x^R \text{ for some } x \in L\}$. We leave the proof of this as an exercise.
- The regular languages are closed under letter substitution, defined as follows: Consider any two alphabets, Σ_1 and Σ_2. Let *sub* be any function from Σ_1 to Σ_2^*. Then *letsub* is a letter substitution function from L_1 to L_2 iff $letsub(L_1) = \{w \in \Sigma_2^* : \exists y \in L_1 (w = y \text{ except that every character } c \text{ of } y \text{ has been replaced by } sub(c))\}$. For example, suppose that $\Sigma_1 = \{\mathtt{a}, \mathtt{b}\}$, $\Sigma_2 = \{\mathtt{0}, \mathtt{1}\}$, $sub(\mathtt{a}) = \mathtt{0}$, and $sub(\mathtt{b}) = \mathtt{11}$. Then $letsub(\{\mathtt{a}^n\mathtt{b}^n : n \geq 0\}) = \{\mathtt{0}^n\mathtt{1}^{2n} : n \geq 0\}$. We leave the proof that the regular languages are closed under letter substitution as an exercise.

EXAMPLE 8.4 Closure Under Complement

Consider the following NDFSM M =



If we use the algorithm that we just described to convert M to a new machine M' that accepts $\neg L(M)$, the last step is to swap the accepting and the nonaccepting states. A quick look at M makes it clear why it is necessary first to make M deterministic and then to complete it by adding the dead state. M accepts the input a in state 4. If we simply swapped accepting and nonaccepting states, without

making the other changes, M' would also accept a. It would do so in state 2. The problem is that M is nondeterministic, and has one path along which a is accepted and one along which it is rejected.

To see why it is necessary to add the dead state, consider the input string aba. M rejects it since the path from state 3 dies when M attempts to read the final a and the path from state 4 dies when it attempts to read the b. But, if we don't add the dead state, M' will also reject it since, in it too, both paths will die.

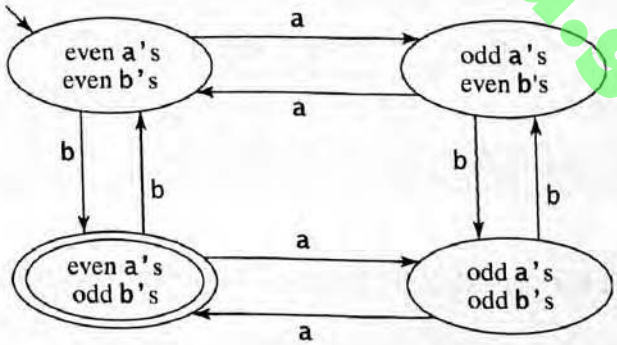
The closure theorems that we have now proved make it easy to take a divide-and-conquer approach to showing that a language is regular. They also let us reuse proofs and constructions that we've already done.

EXAMPLE 8.5 The Divide-and-Conquer Approach

Let $L = \{w \in \{a, b\}^* : w$ contains an even number of a's and an odd number of b's and all a's come in runs of three$\}$. L is regular because it is the intersection of two regular languages. $L = L_1 \cap L_2$, where:

- $L_1 = \{w \in \{a, b\}^* : w$ contains an even number of a's and an odd number of b's $\}$, and
- $L_2 = \{w \in \{a, b\}^* :$ all a's come in runs of three$\}$.

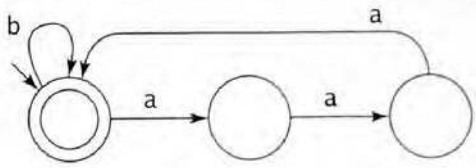
We already know that L_1 is regular, since we showed an FSM that accepts it in Example 5.9:



Of course, we could start with this machine and modify it so that it accepts L. But an easier way is exploit a divide-and-conquer approach. We'll just use the machine we have and then build a second simple machine, this one to accept L_2.

EXAMPLE 8.5 *(Continued)*

Then we can prove that L is regular by exploiting the fact that the regular languages are closed under intersection. The following machine accepts L_2:



The closure theorems are powerful, but they say only what they say. We have stated each of the closure theorems in as strong a form as possible. Any similar claims that are not implied by the theorems as we have stated them are almost certainly false, which can usually be shown easily by finding a simple counterexample.

EXAMPLE 8.6 What the Closure Theorem for Union Does Not Say

The closure theorem for union says that:

$$\textit{if} \quad L_1 \text{ and } L_2 \text{ are regular} \quad \textit{then} \quad L = L_1 \cup L_2 \text{ is regular.}$$

The theorem says nothing, for example, about what happens if L is regular. Does that mean that L_1 and L_2 are also? The answer is maybe. We know that a^+ is regular. We will consider two cases for L_1 and L_2. First, let them be:

$$a^+ = \{a^p : p > 0 \text{ and } p \text{ is prime}\} \cup \{a^p : p > 0 \text{ and } p \text{ is not prime}\}.$$

$$a^+ = \qquad L_1 \qquad \cup \qquad L_2.$$

As we will see in the next section, neither L_1 nor L_2 is regular. But now consider:

$$a^+ = \{a^p : p > 0 \text{ and } p \text{ is even}\} \cup \{a^p : p > 0 \text{ and } p \text{ is odd}\}.$$

$$a^+ = \qquad L_1 \qquad \cup \qquad L_2.$$

In this case, both L_1 and L_2 are regular.

EXAMPLE 8.7 What the Closure Theorem for Concatenation Does Not Say

The closure theorem for concatenation says that:

$$\textit{if} \quad L_1 \text{ and } L_2 \text{ are regular} \quad \textit{then} \quad L = L_1L_2 \text{ is regular.}$$

But the theorem says nothing, for example, about what happens if L_2 is not regular. Does that mean that L isn't regular either? Again, the answer is maybe. We first consider the following example:

$$\{aba^nb^n : n \geq 0\} = \{ab\}\{a^nb^n : n \geq 0\}.$$

$$L \qquad = \quad L_1 \qquad\qquad L_2.$$

As we'll see in the next section, L_2 is not regular. And, in this case, neither is L. But now consider:

$$\{aaa^*\} = \{a^*\}\{a^p : p \text{ is prime}\}.$$
$$L \quad = \quad L_1 \quad L_2.$$

While again L_2 is not regular, now L is.

8.4 Showing That a Language is Not Regular

We can show that a language is regular by exhibiting a regular expression or an FSM or a finite list of the equivalence classes of $\approx_L$ or a regular grammar, or by using the closure properties that we have proved hold for the regular languages. But how shall we show that a language is not regular? In other words, how can we show that none of those descriptions exists for it? It is not sufficient to argue that we tried to find one of them and failed. Perhaps we didn't look in the right place. We need a technique that does not rely on our cleverness (or lack of it).

What we can do is to make use of the following observation about the regular languages: Every regular language L can be accepted by an FSM M with a finite number of states. If L is infinite, then there must be at least one loop in M. All sufficiently long strings in L must be characterized by one or more repeating patterns, corresponding to the substrings that drive M through its loops. It is also true that, if L is infinite, then any regular expression that describes L must contain at least one Kleene star, but we will focus here on FSMs.

To help us visualize the rest of this discussion, consider the FSM M_{LOOP}, shown in Figure 8.1 (a). M_{LOOP} has 5 states. It can accept an infinite number of strings. But the longest one that it can accept without going through any loops has length 4. Now consider the slightly different FSM M_ε, shown in Figure 8.1 (b). M_ε also has 5 states and one loop. But it accepts only one string, aab. The only string that can drive M_ε through its loop is ε. No matter how many times M_ε goes through the loop, it cannot accept any longer strings.

To simplify the following discussion, we will consider only DFSMs, which have no ε-transitions. Each transition step that a DFSM takes corresponds to exactly one character in its input. Since any language that can be accepted by an NDFSM can also be accepted by a DFSM, this restriction will not affect our conclusions.

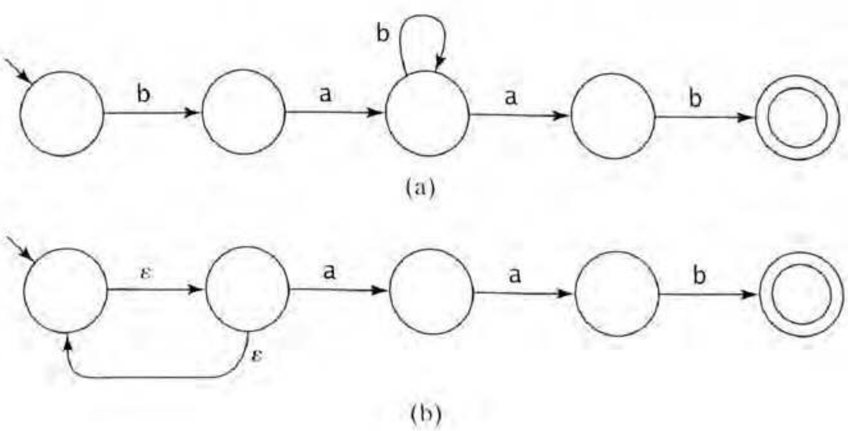


FIGURE 8.1 What is the longest string that a 5-state FSM can accept?

THEOREM 8.5 Long Strings Force Repeated States

Theorem: Let $M = (K, \Sigma, \delta, s, A)$ be any DFSM. If M accepts any string of length $|K|$ or greater, then that string will force M to visit some state more than once (thus traversing at least one loop).

Proof: M must start in one of its states. Each time it reads an input character, it visits some state. So, in processing a string of length n, M creates a total of $n + 1$ state visits (the initial one plus one for each character it reads). If $n + 1 > |K|$, then, by the pigeonhole principle, some state must get more than one visit. So, if $n \geq |K|$, then M must visit at least one state more than once.

Let $M = (K, \Sigma, \delta, s, A)$ be any DFSM. Suppose that there exists some "long" string w (i.e., $|w| \geq |K|$) such that $w \in L(M)$. Then M must go through at least one loop when it reads w. So there is some substring y of w that drove M through at least one loop. Suppose we excise y from w. The resulting string must also be in $L(M)$ since M can accept it just as it accepts w but skipping one pass through one loop. Further, suppose that we splice in one or more extra copies of y, immediately adjacent to the original one. All the resulting strings must also be in $L(M)$ since M can accept them by going through its loop one or more additional times. Using an analogy with a pump, we'll say that we can ***pump*** y out once or in an arbitrary number of times and the resulting string must still be in L.

To make this concrete, let's look again at M_{LOOP}, which accepts, for example, the string babbab. babbab is "long" since its length is 6 and $|K| = 5$. The second b drove M_{LOOP} through its loop. Call the string (in this case b) that drove M_{LOOP} through its loop y. We can pump it out, producing babab, which is also accepted by M_{LOOP}. Or we can pump in as many copies of b as we like, generating such strings as babbbab, babbbbbab, and so forth. M_{LOOP} also accepts all of them. Returning to the original string babbab, the third b also drove M_{LOOP} through its loop. We could also pump it (in or out) and get a similar result.

This property of FSMs, and the languages that they can accept, is the basis for a powerful tool for showing that a language is not regular. If a language contains even one long (to be defined precisely below) string that cannot be pumped in the fashion that we have just described, then it is not accepted by any FSM and so is not regular. We formalize this idea, as the Pumping Theorem, in the next section.

8.4.1 The Pumping Theorem for Regular Languages

THEOREM 8.6 The Pumping Theorem for Regular Languages

Theorem: If L is a regular language, then:

$$\exists k \geq 1\ (\forall \text{ strings } w \in L, \text{ where } |w| \geq k\ (\exists x, y, z\ (w = xyz, \\ |xy| \leq k, \\ y \neq \varepsilon, \text{ and} \\ \forall q \geq 0\ (xy^qz \in L))))).$$

Proof: The proof is the argument that we gave above: If L is regular then it is accepted by some DFSM $M = (K, \Sigma, \delta, s, A)$. Let k be $|K|$. Let w be any string in L of length k or greater. By Theorem 8.5, to accept w, M must traverse some loop at least once. We can carve w up and assign the name y to the first substring to drive M through a loop. Then x is the part of w that precedes y and z is the part of w that follows y. We show that each of the last three conditions must then hold:

- $|xy| \leq k$: M must not only traverse a loop eventually when reading w, it must do so for the first time by at least the time it has read k characters. It can read $k - 1$ characters without revisiting any states. But the k^{th} character must, if no earlier character already has, take M to a state it has visited before. Whatever character does that is the last in one pass through some loop.
- $y \neq \varepsilon$: Since M is deterministic, there are no loops that can be traversed by ε.
- $\forall q \geq 0 \, (xy^q z \in L)$: y can be pumped out once (which is what happens if $q = 0$) or in any number of times (which happens if q is greater than 1) and the resulting string must be in L since it will be accepted by M. It is possible that we could chop y out more than once and still generate a string in L, but without knowing how much longer w is than k, we don't know any more than that it can be pumped out once.

The Pumping Theorem tells us something that is true of every regular language. Generally, if we already know that a language is regular, we won't particularly care about what the Pumping Theorem tells us about it. But suppose that we are interested in some language L and we want to know whether or not it is regular. If we could show that the claims made in the Pumping Theorem are not true of L, then we would know that L is not regular. It is in arguments such as this that we will find the Pumping Theorem very useful. In particular, we will use it to construct ***proofs by contradiction***. We will say, "If L were regular, then it would possess certain properties. But it does not possess those properties. Therefore, it is not regular."

EXAMPLE 8.8 A^nB^n is not Regular

Let L be $A^nB^n = \{a^n b^n : n \geq 0\}$. We can use the Pumping Theorem to show that L is not regular. If it were, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = a^k b^k$. Since $|w| = 2k$, w is long enough and it is in L, so it must satisfy the conditions of the Pumping Theorem. So there must exist x, y, and z, such that $w = xyz$, $|xy| \leq k$, $y \neq \varepsilon$, and $\forall q \geq 0 \, (xy^q z \in L)$. But we show that no such x, y, and z exist. Since we must guarantee that $|xy| \leq k$, y must occur within the first k characters and so $y = a^p$ for some p. Since we must guarantee that $y \neq \varepsilon$, p must be greater than 0. Let $q = 2$. (In other words, we pump in one extra copy of y.) The resulting string is $a^{k+p} b^k$. The last condition of the Pumping Theorem states that this string must be in L, but it is not since it has more a's than b's. Thus there exists at least one long string in L that fails to satisfy the conditions of the Pumping Theorem. So $L = A^nB^n$ is not regular.

The Pumping Theorem is a powerful tool for showing that a language is not regular. But, as with any tool, using it effectively requires some skill. To see how the theorem can be used, let's state it again in its most general terms:

For any language L, if L is regular, then every "long" string in L is pumpable.

So, to show that L is not regular, it suffices to find a single long string w that is in L but is not pumpable. To show that a string is not pumpable, we must show that there is no way to carve it up into x, y, and z in such a way that all three of the conditions of the theorem are met. It is not sufficient to pick a particular y and show that it doesn't work. (We focus on y since, once it has been chosen, everything to the left of it is x and everything to the right of it is z). We must show that there is *no* value for y that works. To do that, we consider all the logically possible classes of values for y (sometimes there is only one such class, but sometimes several must be considered). Then we show that each of them fails to satisfy at least one of the three conditions of the theorem. Generally we do that by assuming that y does satisfy the first two conditions, namely that it occurs within the first k characters and is not ε. Then we consider the third requirement, namely that, for all values of q, xy^qz is in L. To show that it is not possible to satisfy that requirement, it is sufficient to find a single value of q such that the resulting string is not in L. Typically, this can be done by setting q to 0 (thus pumping out once) or to 2 (pumping in once), although sometimes some other value of q must be considered.

In a nutshell then, to use the Pumping Theorem to show that a language L is not regular, we must:

1. Choose a string w, where $w \in L$ and $|w| \geq k$. Note that we do not know what k is; we know only that it exists. So we must state w in terms of k.
2. Divide the possibilities for y into a set of equivalence classes so that all strings in a class can be considered together.
3. For each such class of possible y values, where $|xy| \leq k$ and $y \neq \varepsilon$:
   Choose a value for q such that xy^qz is not in L.

In Example 8.8, y had to fall in the initial a region of w, so that was the only case that needed to be considered. But, had we made a less judicious choice for w, our proof would not have been so simple. Let's look at another proof, with a different w:

EXAMPLE 8.9 A Less Judicious Choice for *w*

Again let L be $A^nB^n = \{a^nb^n : n \geq 0\}$. If A^nB^n were regular, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. Let $w = a^{\lceil k/2 \rceil}b^{\lceil k/2 \rceil}$. (We must use $\lceil k/2 \rceil$, i.e., the smallest integer greater than $k/2$, rather than truncating the division, since k might be odd.) Since $|w| \geq k$ and w is in L, w must satisfy the conditions of the Pumping Theorem. So, there must exist x, y, and z, such that $w = xyz$, $|xy| \leq k$, $y \neq \varepsilon$, and $\forall q \geq 0\ (xy^qz \in L)$. We show that no such x, y, and z exist. This time, if they did, y

could be almost anywhere in w (since all the Pumping Theorem requires is that it occur in the first k characters and there are only at most $k + 1$ characters). So we must consider three cases and show that, in all three, there is no y that satisfies all conditions of the Pumping Theorem. A useful way to describe the cases is to imagine w divided into two regions:

aaaaa.....aaaaaa | bbbbb.....bbbbbb

1 | 2

Now we see that y can fall:

- Exclusively in region 1: In this case, the proof is identical to the proof we did for Example 8.8.
- Exclusively in region 2: then $y = \text{b}^p$ for some p. Since $y \neq \varepsilon$, p must be greater than 0. Let $q = 2$. The resulting string is a^kb^{k+p}. But this string is not in L, since it has more b's than a's.
- Straddling the boundary between regions 1 and 2: Then $y = \text{a}^p\text{b}^r$ for some non-zero p and r. Let $q = 2$. The resulting string will have interleaved a's and b's, and so is not in L.

There exists at least one long string in L that fails to satisfy the conditions of the Pumping Theorem. So $L = \text{A}^n\text{B}^n$ is not regular.

To make maximum use of the Pumping Theorem's requirement that y fall in the first k characters, it is often a good idea to choose a string w that is substantially longer than the k characters required by the theorem. In particular, if w can be chosen so that there is a uniform first region of length at least k, it may be possible to consider just a single case for where y can fall.

> The Pumping Theorem inspires poets ☐, as we'll see in Chapter 10.

A^nB^n is a simple language that illustrates the kind of property that characterizes languages that aren't regular. It isn't of much practical importance, but it is typical of a family of languages, many of which are of more practical significance. In the next example, we consider Bal, the language of balanced parentheses. The structure of Bal is very similar to that of A^nB^n. Bal is important because most languages for describing arithmetic expressions, Boolean queries, and markup systems require balanced delimiters.

EXAMPLE 8.10 The Balanced Parenthesis Language is Not Regular

Let L be Bal $= \{w \in \{), (\}^* :$ the parentheses are balanced$\}$. If L were regular, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. Bal contains complex strings like (())(()()). But it is

EXAMPLE 8.10 *(Continued)*

almost always easier to use the Pumping Theorem if we pick as simple a string as possible. So, let $w = (^k)^k$. Since $|w| = 2k$ and w is in L, w must satisfy the conditions of the Pumping Theorem. So there must exist x, y, and z, such that $w = xyz$, $|xy| \leq k$, $y \neq \varepsilon$, and $\forall q \geq 0\ (xy^qz \in L)$. But we show that no x, y, and z exist. Since $|xy| \leq k$, y must occur within the first k characters and so $y = (^p$ for some p). Since $y \neq \varepsilon$, p must be greater than 0. Let $q = 2$. (In other words, we pump in one extra copy of y.) The resulting string is $(^{k+p})^k$. The last condition of the Pumping Theorem states that this string must be in L, but it is not since it has more ('s than)'s. There exists at least one long string in L that fails to satisfy the conditions of the Pumping Theorem. So L = Bal is not regular.

EXAMPLE 8.11 The Even Palindrome Language is Not Regular

Let L be PalEven $= \{ww^R : w \in \{\mathtt{a}, \mathtt{b}\}^*\}$. PalEven is the language of even-length palindromes of a's and b's. We can use the Pumping Theorem to show that PalEven is not regular. If it were, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. (Note here that the variable w used in the definition of L is different from the variable w mentioned in the Pumping Theorem.) We will choose w so that we only have to consider one case for where y could fall. Let $w = \mathtt{a}^k\mathtt{b}^k\mathtt{b}^k\mathtt{a}^k$. Since $|w| = 4k$ and w is in L, w must satisfy the conditions of the Pumping Theorem. So there must exist x, y, and z, such that $w = xyz$, $|xy| \leq k$, $y \neq \varepsilon$, and $\forall q \geq 0\ (xy^qz \in L)$. Since $|xy| \leq k$, y must occur within the first k characters and so $y = \mathtt{a}^p$ for some p. Since $y \neq \varepsilon$, p must be greater than 0. Let $q = 2$. The resulting string is $\mathtt{a}^{k+p}\mathtt{b}^k\mathtt{b}^k\mathtt{a}^k$. If p is odd, then this string is not in PalEven because all strings in PalEven have even length. If p is even then it is at least 2, so the first half of the string has more a's than the second half does, so it is not in PalEven. So L = PalEven is not regular.

The Pumping Theorem says that, for any language L, if L is regular, then all long strings in L must be pumpable. Our strategy in using it to show that a language L is not regular is to find *one* string that fails to meet that requirement. Often, there are many long strings that *are* pumpable. If we try to work with them, we will fail to derive the contradiction that we seek. In that case, we will know nothing about whether or not L is regular. To find a w that is not pumpable, think about what property of L is not checkable by an FSM and choose a w that exhibits that property. Consider again our last example. The thing that an FSM cannot do is to remember an arbitrarily long first half and check it against the second half. So we chose a w that would have forced it to do that. Suppose instead that we had let $w = \mathtt{a}^k\mathtt{a}^k$. It is in L and long enough. But y could be aa and we could pump it out or in and all the resulting strings would be in L.

So far, all of our Pumping Theorem proofs have set q to 2. But that is not always the thing to do. Sometimes it will be necessary to set it to 0. (In other words, we will pump y out).

EXAMPLE 8.12 The Language with More a's Than b's is Not Regular

Let $L = \{a^n b^m : n > m\}$. We can use the Pumping Theorem to show that L is not regular. If it were, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = a^{k+1}b^k$. Since $|w| = 2k + 1$ and w is in L, w must satisfy the conditions of the Pumping Theorem. So there must exist x, y, and z, such that $w = xyz$, $|xy| \leq k$, $y \neq \varepsilon$, and $\forall q \geq 0\ (xy^qz \in L)$. Since $|xy| \leq k$, y must occur within the first k characters and so $y = a^p$ for some p. Since $y \neq \varepsilon$, p must be greater than 0. There are already more a's than b's, as required by the definition of L. If we pump in, there will be even more a's and the resulting string will still be in L. But we can set q to 0 (and so pump out). The resulting string is then $a^{k+1-p}b^k$. Since $p > 0$, $k + 1 - p \leq k$, so the resulting string no longer has more a's than b's and so is not in L. There exists at least one long string in L that fails to satisfy the conditions of the Pumping Theorem. So L is not regular.

Notice that the proof that we just did depended on our having chosen a w that is just barely in L. It had exactly one more a than b. So y could be any string of up to k a's. If we pumped in extra copies of y, we would have gotten strings that were still in L. But if we pumped out even a single a, we got a string that was not in L, and so we were able to complete the proof.. Suppose, though, that we had chosen $w = a^{2k}b^k$. Again, pumping in results in strings in L. And now, if y were simply a, we could pump out and get a string that was still in L. So that proof attempt fails. In general, it is a good idea to choose a w that barely meets the requirements for L. That makes it more likely that pumping will create a string that is not in L.

Sometimes values of q other than 0 or 2 may also be required.

EXAMPLE 8.13 The Prime Number of a's Language is Not Regular

Let L be $\text{Prime}_a = \{a^n : n \text{ is prime}\}$. We can use the Pumping Theorem to show that L is not regular. If it were, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = a^j$, where j is the smallest prime number greater than $k + 1$. Since $|w| > k$, w must satisfy the conditions of the Pumping Theorem. So there must exist x, y, and z, such that $w = xyz$, $|xy| \leq k$ and $y \neq \varepsilon$. $y = a^p$ for some p. The Pumping Theorem further requires that $\forall q \geq 0\ (xy^qz \in L)$. So, $\forall q \geq 0$ ($a^{|x|+|z|+q \cdot |y|}$ must be in L). That means that $|x| + |z| + q \cdot |y|$ must be prime.

EXAMPLE 8.13 *(Continued)*

But suppose that $q = |x| + |z|$. Then:

$$\begin{aligned} |x| + |z| + q \cdot |y| &= |x| + |z| + (|x| + |z|) \cdot y \\ &= (|x| + |z|) \cdot (1 + |y|), \end{aligned}$$

which is composite (non-prime) if both factors are greater than 1. $(|x| + |z|) > 1$ because $|w| > k + 1$, and $|y| \leq k$. $(1 + |y|) > 1$ because $|y| > 0$. So, for at least that one value of q, the resulting string is not in L. So L is not regular.

When we do a Pumping Theorem proof that a language L is not regular, we have two choices to make: a value for w and a value for q. As we have just seen, there are some useful heuristics that can guide our choices:

- To choose w:
  - Choose a w that is in the part of L that makes it not regular.
  - Choose a w that is only barely in L.
  - Choose a w with as homogeneous as possible an initial region of length at least k.
- To choose q:
  - Try letting q be either 0 or 2.
  - If that doesn't work, analyze L to see if there is some other specific value that will work.

8.4.2 Using Closure Properties

Sometimes the easiest way to prove that a language L is not regular is to use the closure theorems for regular languages, either alone or in conjunction with the Pumping Theorem. The fact that the regular languages are closed under intersection is particularly useful.

EXAMPLE 8.14 Using Intersection to Force Order Constraints

Let $L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$. If L were regular, then $L' = L \cap a^*b^*$ would also be regular. But $L' = \{a^n b^n : n \geq 0\}$, which we have already shown is not regular. So L isn't either.

EXAMPLE 8.15 Using Closure Under Complement

Let $L = \{a^i b^j : i, j \geq 0 \text{ and } i \neq j\}$. It seems unlikely that L is regular since any machine to accept it would have to count the a's. It is possible to use the Pumping

Theorem to prove that L is not regular but it is not easy to see how. Suppose, for example, that we let $w = a^{k+1}b^k$. But then y could be aa and it would pump since $a^{k-1}b^k$ is in L, and so is $a^{k+1+2(q-1)}b^k$, for all nonnegative values of q.

Instead, let $w = a^k b^{k+k!}$. Then $y = a^p$ for some nonzero p. Let $q = (k!/p) + 1$ (in other words, pump in $(k!/p)$ times). Note that $(k!/p)$ must be an integer because $p < k$. The number of a's in the resulting string is $k + (k!/p)p = k + k!$. So the resulting string is $a^{k+k!}b^{k+k!}$, which has equal numbers of a's and b's and so is not in L.

The closure theorems provide an easier way. We observe that if L were regular, then $\neg L$ would also be regular, since the regular languages are closed under complement. $\neg L = \{a^n b^n : n \geq 0\} \cup \{$ strings of a's and b's that do not have all a's in front of all b's$\}$. If $\neg L$ is regular, then $\neg L \cap a^*b^*$ must also be regular. But $\neg L \cap a^*b^* = \{a^n b^n : n \geq 0\}$, which we have already shown is not regular. So neither is $\neg L$ or L.

Sometimes, using the closure theorems is more than a convenience. There are languages that are not regular but that do meet all the conditions of the Pumping Theorem. The Pumping Theorem alone is insufficient to prove that those languages are not regular, but it may be possible to complete a proof by exploiting the closure properties of the regular languages.

EXAMPLE 8.16 Sometimes We Must Use the Closure Theorems

Let $L = \{a^i b^j c^k : i, j, k \geq 0$ and (if $i = 1$ then $j = k)\}$. Every string of length at least 1 that is in L is pumpable. It is easier to see this if we rewrite the final condition as $(i \neq 1)$ or $(j = k)$. Then we observe:

- If $i = 0$ then: If $j \neq 0$, let y be b; otherwise, let y be c. Pump in or out. Then i will still be 0 and thus not equal to 1, so the resulting string is in L.
- If $i = 1$ then: Let y be a. Pump in or out. Then i will no longer equal 1, so the resulting string is in L.
- If $i = 2$ then: Let y be aa. Pump in or out. Then i cannot equal 1, so the resulting string is in L.
- If $i > 2$ then: Let y be a. Pump out once or in any number of times. Then i cannot equal 1, so the resulting string is in L.

But L is not regular. One way to prove this is to use the fact that the regular languages are closed under intersection. So, if L were regular, then $L' = L \cap ab^*c^* = \{ab^j c^k : j, k \geq 0$ and $j = k\}$ would also be regular. But it is not, which we can show using the Pumping Theorem. Let $w = ab^k c^k$. Then y must occur in the first k characters of w. If y includes the initial a, pump in once. The resulting string is not in L' because it contains more than one a. If y does not include the initial a, then it must be b^p, where $0 < p < k$. Pump in once. The resulting string is not in L' because it contains more b's than c's. Since L' is not regular, neither is L.

EXAMPLE 8.16 *(Continued)*

Another way to show that L is not regular is to use the fact that the regular languages are closed under reverse. $L^R = \{\mathrm{c}^k\ \mathrm{b}^j\ \mathrm{a}^i : i, j, k \geq 0$ and (if $i = 1$ then $j = k)\}$. If L were regular then L^R would also be regular. But it is not, which we can show using the Pumping Theorem. Let $w = \mathrm{c}^k\ \mathrm{b}^k\ \mathrm{a}$. y must occur in the first k characters of w, so $y = \mathrm{c}^p$, where $0 < p \leq k$. Set q to 0. The resulting string contains a single a, so the number of b's and c's must be equal for it to be in L^R. But there are fewer c's than b's. So the resulting string is not in L^R. L^R is not regular. Since L^R is not regular, neither is L.

8.5 Exploiting Problem-Specific Knowledge

Given some new language L, the theory that we have been describing provides the skeleton for an analysis of L. If L is simple, that may be enough. But if L is based on a real problem, any analysis of it will also depend on knowledge of the task domain. We got a hint of this in Example 8.13, where we had to use some knowledge about numbers and algebra. Other problems also require mathematical facts.

EXAMPLE 8.17 The Octal Representation of a Number Divisible by 7

Let $L = \{w \in \{0, 1, 2, 3, 4, 5, 6, 7\}^* : w$ is the octal representation of a nonnegative integer that is divisible by $7\}$. The first several strings in L are: 0, 7, 16, 25, 34, 43, 52, and 61. Is L regular? Yes, because there is a simple, 7-state DFSM M that accepts L. The structure of M takes advantage of the fact that w is in L iff the sum of its digits, viewed as numbers, is divisible by 7. So the states of M correspond to the modulo 7 sum of the digits so far. We omit the details.

Sometimes L corresponds to a problem from a domain other than mathematics, in which case facts from that domain will be important.

EXAMPLE 8.18 A Music Language

Let Σ = {𝅝, 𝅗𝅥, ♩, ♪, 𝅘𝅥𝅯, 𝅘𝅥𝅰}. Let $L = \{w : w$ represents a song written in 4/4 time$\}$. L is regular. It can be accepted by an FSM that checks for 4 beats between measure bars, where 𝅝 counts as 4, 𝅗𝅥 counts as 2, ♩ counts as 1, ♪ counts as $^1/_2$, 𝅘𝅥𝅯 counts as $^1/_4$, and 𝅘𝅥𝅰 counts as 1/8.

> Other techniques described in this book can also be applied to the language of music. (N.1)

EXAMPLE 8.19 English

Is English a regular language? If we assume that there is a longest sentence, then English is regular because it is finite. If we assume that there is not a longest sentence and that the recursive constructs in English can be arbitrarily nested, then it is easy to show that English is not regular. We consider a very small subset of English, sentences such as:

- `The rat ran.`
- `The rat that the cat saw ran.`
- `The rat that the cat that the dog chased saw ran.`

There is a limit on how deeply nested sentences such as this can be if people are going to be able to understand them easily. But the grammar of English imposes no hard upper bound. So we must allow any number of embedded sentences. Let A = {`cat`, `rat`, `dog`, `bird`, `bug`, `pony`} and let V = {`ran`, `saw`, `chased`, `flew`, `sang`, `frolicked`}. If English were regular, then L = English $\cap$ {`The` A (`that the` A)$^*V^*V$ } would also be regular. But every English sentence of this form has the same number of nouns as verbs. So we have that:

$$L = \{\texttt{The } A(\texttt{that the } A)^n\, V^n V, n \geq 0\}.$$

We can show that L is not regular by pumping. The outline of the proof is the same as the one we used in Example 8.9 to show that A^nB^n is not regular. Let w = `The cat (that the rat)`^{k} `saw`^{k} `ran`. y must occur within the first k characters of w. If y is anything other than (`the` A `that`)^p, or (A `that the`)^p, or (`that the` A)^p, for some nonzero p, pump in once and the resulting string will not be of the correct form. If y is equal to one of those strings, pump in once and the number of nouns will no longer equal the number of verbs. In either case the resulting string is not in L. So English is not regular.

> Is there a longest English sentence? Are there other ways of showing that English isn't regular? Would it be useful to describe English as a regular language even if we could? (L.3.1)

8.6 Functions on Regular Languages

In Section 8.3, we considered some important functions that can be applied to the regular languages and we showed that the class of regular languages is closed under them. In this section, we will look at some additional functions and ask whether the regular languages are closed under them. In some cases, we will see that the answer is yes. We will prove that the answer is yes by showing a construction that builds one FSM from another. In other cases, we will see that the answer is no, which we now have the tools to prove.

EXAMPLE 8.20 The Function *firstchars*

Consider again the function *firstchars*, which we defined in Example 4.11. Recall that $firstchars(L) = \{w : \exists y \in L\ (y = cx, c \in \Sigma_L, x \in \Sigma_L^*, \text{ and } w \in c^*)\}$. In other words, to compute $firstchars(L)$, we find all the characters that can be initial characters of some string in L. For each such character c, $c^* \subseteq firstchars(L)$.

The regular languages are closed under *firstchars*. The proof is by construction. If L is a regular language, then there exists some DFSM $M = (K, \Sigma, \delta, s, A)$ that accepts L. We construct, from M, a new DFSM $M' = (K', \Sigma, \delta', s', A')$ that accepts $firstchars(L)$. The algorithm to construct M' is:

1. Mark all the states in M from which there exists some path to some accepting state.

   /* Find all the characters that are initial characters in some string in L.

2. $clist = \emptyset$.
3. For each character c in Σ do:

   If there is a transition from s, with label c, to some state q, and q was marked in step 1 then:

   $clist = clist \cup \{c\}$.

   /* Build M'.

4. If $clist = \emptyset$ then construct M' with a single state s', which is not accepting.
5. Else do:

   Create a start state s' and make it the first state in A'.

   For each character c in $clist$ do:

   Create a new state q_c and add it to A'.
   Add a transition from s' to q_c labeled c.
   Add a transition from q_c to q_c labeled c.

M' accepts exactly the strings in $firstchars(L)$, so $firstchars(L)$ is regular.

We can also prove that $firstchars(L)$ must be regular by showing how to construct a regular expression that describes it. We begin by computing $clist = \{c_1, c_2, \ldots, c_n\}$ as described above. Then a regular expression that describes $firstchars(L)$ is:

$$c_1^* \cup c_2^* \cup \cdots \cup c_n^*.$$

The algorithm that we just presented constructs one program (a DFSM), using another program (another DFSM) as a starting point. The algorithm is straightforward. We have omitted a detailed proof of its correctness, but that proof is also straightforward. Suppose that, instead of representing an input language L as a DFSM, we had represented it as an arbitrary program (written in C++ or Java or whatever) that accepted it. It would not have been as straightforward to have designed a corresponding algorithm to convert that program into one that accepted $firstchars(L)$. We have just seen another advantage of the FSM formalism.

EXAMPLE 8.21 The Function *chop*

Consider again the function *chop*, defined in Example 4.10. $Chop(L) = \{w : \exists x \in L (x = x_1 c x_2, x_1 \in \Sigma_L^*, x_2 \in \Sigma_L^*, c \in \Sigma_L, |x_1| = |x_2|, \text{ and } w = x_1 x_2)\}$. In other words, $chop(L)$ is all the odd length strings in L with their middle character chopped out.

The regular languages are not closed under *chop*. To show this, it suffices to show one counterexample, i.e., one regular language L such that $chop(L)$ is not regular. Let $L =$ a*db*. L is regular since it can be described with a regular expression.

What is *chop*(a*db*)? Let w be some string in a*db*. Now we observe:

- If $|w|$ is even, then there is no middle character to chop so w contributes no string to *chop* (a*db*).
- If $|w|$ is odd and w has an equal number of a's and b's, then its middle character is d. Chopping out the d produces, and contributes to *chop*(a*db*), a string in $\{a^n b^n : n \geq 0\}$.
- If $|w|$ is odd and w does not have an equal number of a's and b's, then its middle character is not d. Chopping out the middle character produces a string that still contains one d. Also note that, since $|w|$ is odd and the number of a's differs from the number of b's, it must differ by at least two. So, when w's middle character is chopped out, the resulting string will still have different numbers of a's and b's.

So *chop*(a*db*) contains all strings in $\{a^n b^n : n \geq 0\}$ plus some strings in $\{w \in$ a*db* : $|w|$ is even and $\#_a(w) \neq \#_b(w)\}$. We can now show that *chop*(a*db*) is not regular. If it were, then the language $L' = chop(a^*db^*) \cap a^*b^*$, would also be regular since the regular languages are closed under intersection. But $L' = \{a^n b^n : n \geq 0\}$, which we have already shown is not regular. So neither is *chop*(a*db*). Since there exists at least one regular language L with the property that $chop(L)$ is not regular, the regular languages are not closed under *chop*.

EXAMPLE 8.22 The Function *maxstring*

Define $maxstring(L) = \{w: w \in L \text{ and } \forall z \in \Sigma^*(z \neq \varepsilon \rightarrow wz \notin L)\}$. In other words, $maxstring(L)$ contains exactly those strings in L that cannot be extended on the right and still be in L. Let's look at *maxstring* applied to some languages:

L	$maxstring(L)$
$\emptyset$	$\emptyset$
a*b*	$\emptyset$
ab*a	ab*a
a*b*a	a*b$^+$a

EXAMPLE 8.23 The Function *mix*

Define $mix(L) = \{w : \exists x, y, z (x \in L, x = yz, |y| = |z|, w = yz^R)\}$. In other words, $mix(L)$ contains exactly those strings that can be formed by taking some even length string in L and reversing its second half. Let's look at *mix* applied to some languages:

L	$mix(L)$
$\varnothing$	$\varnothing$
$(a \cup b)^*$	$((a \cup b)(a \cup b))^*$
$(ab)^*$	$\{(ab)^{2n+1} : n \geq 0\} \cup \{(ab)^n(ba)^n : n \geq 0\}$
$(ab)^*a(ab)^*$	$\varnothing$

The regular languages are closed under *maxstring*. They are not closed under *mix*. We leave the proof of these claims as an exercise.

Exercises

1. For each of the following languages L, state whether L is regular or not and prove your answer:

a. $\{a^i b^j : i, j \geq 0 \text{ and } i + j = 5\}$.

b. $\{a^i b^j : i, j \geq 0 \text{ and } i - j = 5\}$.

c. $\{a^i b^j : i, j \geq 0 \text{ and } |i - j| \equiv_5 0\}$.

d. $\{w \in \{0, 1, \#\}^* : w = x \# y, \text{ where } x, y \in \{0, 1\}^* \text{ and } |x| \cdot |y| \equiv_5 0\}$.

e. $\{a^i b^j : 0 \leq i < j < 2000\}$.

f. $\{w \in \{Y, N\}^* : w \text{ contains at least two Y's and at most two N's}\}$.

g. $\{w = xy : x, y \in \{a, b\}^* \text{ and } |x| = |y| \text{ and } \#_a(x) \geq \#_a(y)\}$.

h. $\{w = xyzy^R x : x, y, z \in \{a, b\}^*\}$.

i. $\{w = xyzy : x, y, z \in \{0, 1\}^+\}$.

j. $\{w \in \{0, 1\}^* : \#_0(w) \neq \#_1(w)\}$.

k. $\{w \in \{a, b\}^* : w = w^R\}$.

l. $\{w \in \{a, b\}^* : \exists x \in \{a, b\}^+ \ (w = x\, x^R x)\}$.

m. $\{w \in \{a, b\}^* : \text{the number of occurrences of the substring ab equals the number of occurrences of the substring ba}\}$.

n. $\{w \in \{a, b\}^* : w \text{ contains exactly two more b's than a's}\}$.

o. $\{w \in \{a, b\}^* : w = xyz, |x| = |y| = |z|, \text{ and } z = x \text{ with every a replaced by b and every b replaced by a}\}$. Example: abbbabbaa $\in L$, with $x =$ abb, $y =$ bab, and $z =$ baa.

p. $\{w : w \in \{a - z\}^* \text{ and the letters of } w \text{ appear in reverse alphabetical order}\}$. For example, spoonfeed $\in L$.

q. $\{w: w \in \{\mathsf{a} - \mathsf{z}\}^*$ every letter in w appears at least twice$\}$. For example, $\mathsf{unprosperousness} \in L$.

r. $\{w : w$ is the decimal encoding of a natural number in which the digits appear in a non-decreasing order without leading zeros$\}$.

s. $\{w$ of the form: $<integer_1> + <integer_2> = <integer_3>$, where each of the substrings $<integer_1>$, $<integer_2>$, and $<integer_3>$ is an element of $\{0 - 9\}^*$ and $integer_3$ is the sum of $integer_1$ and $integer_2\}$. For example, $\mathsf{124+5=129} \in L$.

t. L_0^* where $L_0 = \{\mathsf{ba}^i\,\mathsf{b}^j\,\mathsf{a}^k, j \geq 0, 0 \leq i \leq k\}$.

u. $\{w : w$ is the encoding of a date that occurs in a year that is a prime number$\}$. A date will be encoded as a string of the form *mm/dd/yyyy*, where each m, d, and y is drawn from $\{0–9\}$.

v. $\{w \in \{1\}^* : w$ is, for some $n \geq 1$, the unary encoding of $10^n\}$. (So $L = \{1111111111, 1^{100}, 1^{1000}, \ldots\}$.)

2. For each of the following languages L, state whether L is regular or not and prove your answer:

a. $\{w \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}^* :$ in each prefix x of w, $\#_{\mathsf{a}}(x) = \#_{\mathsf{b}}(x) = \#_{\mathsf{c}}(x))\}$.

b. $\{w \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}^* : \exists$ some prefix x of w $(\#_{\mathsf{a}}(x) = \#_{\mathsf{b}}(x) = \#_{\mathsf{c}}(x))\}$.

c. $\{w \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}^* : \exists$ some prefix x of w $(x \neq \varepsilon$ and $\#_{\mathsf{a}}(x) = \#_{\mathsf{b}}(x) = \#_{\mathsf{c}}(x))\}$.

3. Define the following two languages:

$L_{\mathsf{a}} = \{w \in \{\mathsf{a}, \mathsf{b}\}^* :$ in each prefix x of w, $\#_{\mathsf{a}}(x) \geq \#_{\mathsf{b}}(x)\}$.

$L_{\mathsf{b}} = \{w \in \{\mathsf{a}, \mathsf{b}\}^* :$ in each prefix x of w, $\#_{\mathsf{b}}(x) \geq \#_{\mathsf{a}}(x)\}$.

a. Let $L_1 = L_{\mathsf{a}} \cap L_{\mathsf{b}}$. Is L_1 regular? Prove your answer.

b. Let $L_2 = L_{\mathsf{a}} \cup L_{\mathsf{b}}$. Is L_2 regular? Prove your answer.

4. For each of the following languages L, state whether L is regular or not and prove your answer:

a. $\{uww^{\mathrm{R}}v : u, v, w \in \{\mathsf{a}, \mathsf{b}\}^+\}$.

b. $\{xyzy^{\mathrm{R}}x : x, y, z \in \{\mathsf{a}, \mathsf{b}\}^+\}$.

5. Use the Pumping Theorem to complete the proof, given in L.3.1, that English isn't regular.

6. Prove *by construction* that the regular languages are closed under:

a. intersection.

b. set difference.

7. Prove that the regular languages are closed under each of the following operations:

a. $pref(L) = \{w : \exists x \in \Sigma^*(wx \in L)\}$.

b. $suff(L) = \{w : \exists x \in \Sigma^*(xw \in L)\}$.

c. $reverse(L) = \{x \in \Sigma^* : x = w^{\mathrm{R}}$ for some $w \in L\}$.

d. letter substitution (as defined in Section 8.3).

8. Using the defintions of *maxstring* and *mix* given in Section 8.6, give a precise definition of each of the following languages:

a. *maxstring*($A^n B^n$).

b. *maxstring*($a^i b^j c^k, 1 \le k \le j \le i$).

c. *maxstring*(L_1L_2), where $L_1 = \{w \in \{a, b\}^* : w$ contains exactly one a$\}$ and $L_2 = \{a\}$.

d. *mix*((aba)*).

e. *mix*(a*b*).

9. Prove that the regular languages are not closed under *mix*.

10. Recall that $maxstring(L) = \{w : w \in L$ and $\forall z \in \Sigma^*(z \neq \varepsilon \rightarrow wz \notin L)\}$.

a. Prove that the regular languages are closed under *maxstring*.

b. If *maxstring*(L) is regular, must L also be regular? Prove your answer.

11. Define the function $midchar(L) = \{c : \exists w \in L(w = ycz, c \in \Sigma_L, y \in \Sigma_L^*, z \in \Sigma_L^*, |y| = |z|)\}$. Answer each of the following questions and prove your answer:

a. Are the regular languages closed under *midchar*?

b. Are the nonregular languages closed under *midchar*?

12. Define the function $twice(L) = \{w : \exists x \in L$ (x can be written as $c_1c_2 \ldots c_n$, for some $n \ge 1$, where each $c_i \in \Sigma_L$, and $w = c_1c_1c_2c_2 \ldots c_nc_n)\}$.

a. Let $L =$ (1 ∪ 0)*1. Write a regular expression for *twice*(L).

b. Are the regular languages closed under *twice*? Prove your answer.

13. Define the function $shuffle(L) = \{w : \exists x \in L$ (w is some permutation of x)$\}$. For example, if $L =$ {ab, abc}, then $shuffle(L) =$ {ab, abc, ba, acb, bac, bca, cab, cba}. Are the regular languages closed under *shuffle*? Prove your answer.

14. Define the function $copyandreverse(L) = \{w : \exists x \in L(w = xx^R)\}$. Are the regular languages closed under *copyandreverse*? Prove your answer.

15. Let L_1 and L_2 be regular languages. Let L be the language consisting of strings that are contained in exactly one of L_1 and L_2. Prove that L is regular.

16. Define two integers i and j to be ***twin primes*** □ iff both i and j are prime and $|j - i| = 2$.

a. Let $L = \{w \in \{1\}^* : w$ is the unary notation for a natural number n such that there exists a pair p and q of twin primes, both $> n\}$. Is L regular?

b. Let $L = \{x, y : x$ is the decimal encoding of a positive integer i, y is the decimal encoding of a positive integer j, and i and j are twin primes$\}$. Is L regular?

17. Consider any function $f(L_1) = L_2$, where L_1 and L_2 are both languages over the alphabet $\Sigma = \{0, 1\}$. A function f is ***nice*** iff whenever L_2 is regular, L_1 is regular. For each of the following functions, f, state whether or not it is nice and prove your answer.

a. $f(L) = L^R$.

b. $f(L) = \{w : w$ is formed by taking a string in L and replacing all 1's with 0's and leaving the 0's unchanged$\}$.

c. $f(L) = L \cup 0^*$.

d. $f(L) = \{w : w$ is formed by taking a string in L and replacing all 1's with 0's and all 0's with 1's (simultaneously)$\}$.

e. $f(L) = \{w : \exists x \in L\ (w = x00)\}$.

f. $f(L) = \{w : w$ is formed by taking a string in L and removing the last character$\}$.

18. We'll say that a language L over an alphabet Σ is ***splitable*** iff the following property holds: Let w be any string in L that can be written as $c_1c_2 \ldots c_{2n}$, for some $n \geq 1$, where each $c_i \in \Sigma$. Then $x = c_1c_3 \ldots c_{2n-1}$ is also in L.

a. Give an example of a splitable regular language.

b. Is every regular language splitable?

c. Does there exist a nonregular language that is splitable?

19. Define the class IR to be the class of languages that are both infinite and regular. Tell whether the class IR closed under:

a. union.

b. intersection.

c. Kleene star.

20. Consider the language $L = \{x0^n y\ 1^n z : n \geq 0, x \in P, y \in Q, z \in R$, where P, Q, and R are nonempty sets over the alphabet $\{0, 1\}\}$. Can you find regular sets P, Q, and R such that L is not regular? Can you find regular sets P, Q, and R such that L is regular?

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

a. There are uncountably many non-regular languages over $\Sigma = \{a, b\}$.

b. The union of an infinite number of regular languages must be regular.

c. The union of an infinite number of regular languages is never regular.

d. If L_1 and L_2 are not regular languages, then $L_1 \cup L_2$ is not regular.

e. If L_1 and L_2 are regular languages, then $L_1 \otimes L_2 = \{w : w \in (L_1 - L_2)$ or $w \in (L_2 - L_1)\}$ is regular.

f. If L_1 and L_2 are regular languages and $L_1 \subseteq L \subseteq L_2$, then L must be regular.

g. The intersection of a regular language and a nonregular language must be regular.

h. The intersection of a regular language and a nonregular language must not be regular.

i. The intersection of two nonregular languages must not be regular.

j. The intersection of a finite number of nonregular languages must not be regular.

k. The intersection of an infinite number of regular languages must be regular.

l. It is possible that the concatenation of two nonregular languages is regular.

m. It is possible that the union of a regular language and a nonregular language is regular.

n. Every nonregular language can be described as the intersection of an infinite number of regular languages.

o. If L is a language that is not regular, then L^* is not regular.

p. If L^* is regular, then L is regular.

q. The nonregular languages are closed under intersection.

r. Every subset of a regular language is regular.

s. Let $L_4 = L_1L_2L_3$. If L_1 and L_2 are regular and L_3 is not regular, it is possible that L_4 is regular.

t. If L is regular, then so is $\{xy : x \in L \text{ and } y \notin L\}$.

u. Every infinite regular language properly contains another infinite regular language.