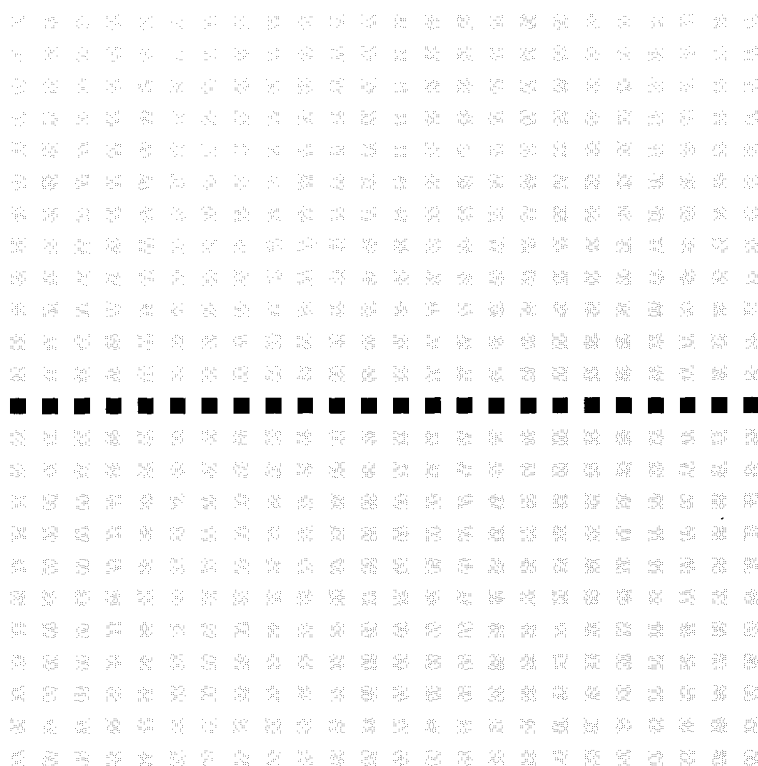
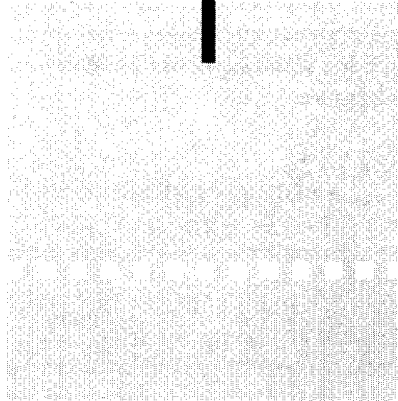


PART ONE



AUTOMATA AND LANGUAGES

1



REGULAR LANGUAGES

The theory of computation begins with a question: What is a computer? It is perhaps a silly question, as everyone knows that this thing I type on is a computer. But these real computers are quite complicated—too much so to allow us to set up a manageable mathematical theory of them directly. Instead we use an idealized computer called a *computational model*. As with any model in science, a computational model may be accurate in some ways but perhaps not in others. Thus we will use several different computational models, depending on the features we want to focus on. We begin with the simplest model, called the *finite state machine* or *finite automaton*.

1.1 有限自动机

FINITE AUTOMATA

Finite automata are good models for computers with an extremely limited amount of memory. What can a computer do with such a small memory? Many useful things! In fact, we interact with such computers all the time, as they lie at the heart of various electromechanical devices.

The controller for an automatic door is one example of such a device. Often found at supermarket entrances and exits, automatic doors swing open when sensing that a person is approaching. An automatic door has a pad in front to

detect the presence of a person about to walk through the doorway. Another pad is located to the rear of the doorway so that the controller can hold the door open long enough for the person to pass all the way through and also so that the door does not strike someone standing behind it as it opens. This configuration is shown in the following figure.

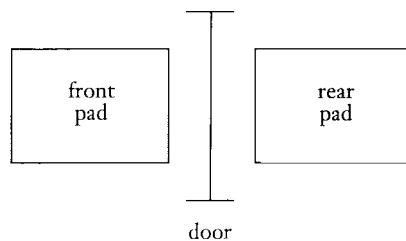


FIGURE 1.1
Top view of an automatic door

The controller is in either of two states: “OPEN” or “CLOSED,” representing the corresponding condition of the door. As shown in the following figures, there are four possible input conditions: “FRONT” (meaning that a person is standing on the pad in front of the doorway), “REAR” (meaning that a person is standing on the pad to the rear of the doorway), “BOTH” (meaning that people are standing on both pads), and “NEITHER” (meaning that no one is standing on either pad).

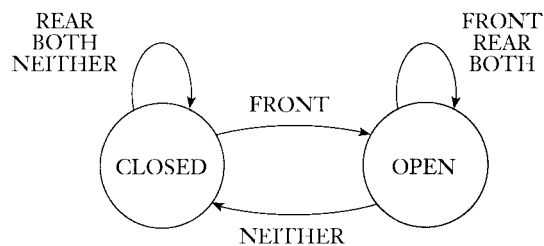


FIGURE 1.2
State diagram for automatic door controller

		input signal			
state		NEITHER	FRONT	REAR	BOTH
	CLOSED	CLOSED	OPEN	CLOSED	CLOSED
	OPEN	CLOSED	OPEN	OPEN	OPEN

FIGURE 1.3
State transition table for automatic door controller

The controller moves from state to state, depending on the input it receives. When in the CLOSED state and receiving input NEITHER or REAR, it remains in the CLOSED state. In addition, if the input BOTH is received, it stays CLOSED because opening the door risks knocking someone over on the rear pad. But if the input FRONT arrives, it moves to the OPEN state. In the OPEN state, if input FRONT, REAR, or BOTH is received, it remains in OPEN. If input NEITHER arrives, it returns to CLOSED.

For example, a controller might start in state CLOSED and receive the series of input signals FRONT, REAR, NEITHER, FRONT, BOTH, NEITHER, REAR, and NEITHER. It then would go through the series of states CLOSED (starting), OPEN, OPEN, CLOSED, OPEN, OPEN, CLOSED, CLOSED, and CLOSED.

Thinking of an automatic door controller as a finite automaton is useful because that suggests standard ways of representation as in Figures 1.2 and 1.3. This controller is a computer that has just a single bit of memory, capable of recording which of the two states the controller is in. Other common devices have controllers with somewhat larger memories. In an elevator controller a state may represent the floor the elevator is on and the inputs might be the signals received from the buttons. This computer might need several bits to keep track of this information. Controllers for various household appliances such as dishwashers and electronic thermostats, as well as parts of digital watches and calculators, are additional examples of computers with limited memories. The design of such devices requires keeping the methodology and terminology of finite automata in mind.

Finite automata and their probabilistic counterpart *Markov chains* are useful tools when we are attempting to recognize patterns in data. These devices are used in speech processing and in optical character recognition. Markov chains have even been used to model and predict price changes in financial markets.

We will now take a closer look at finite automata from a mathematical perspective. We will develop a precise definition of a finite automaton, terminology for describing and manipulating finite automata, and theoretical results that describe their power and limitations. Besides giving you a clearer understanding of what finite automata are and what they can and cannot do, this theoretical development will allow you to practice and become more comfortable with mathematical definitions, theorems, and proofs in a relatively simple setting.

In beginning to describe the mathematical theory of finite automata, we do so in the abstract, without reference to any particular application. The following figure depicts a finite automaton called M_1 .

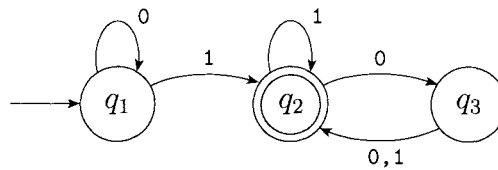


FIGURE 1.4

A finite automaton called M_1 that has three states

Figure 1.4 is called the *state diagram* of M_1 . It has three *states*, labeled q_1 , q_2 , and q_3 . The *start state*, q_1 , is indicated by the arrow pointing at it from nowhere. The *accept state*, q_2 , is the one with a double circle. The arrows going from one state to another are called *transitions*.

When this automaton receives an input string such as 1101, it processes that string and produces an output. The output is either *accept* or *reject*. We will consider only this yes/no type of output for now to keep things simple. The processing begins in M_1 's start state. The automaton receives the symbols from the input string one by one from left to right. After reading each symbol, M_1 moves from one state to another along the transition that has that symbol as its label. When it reads the last symbol, M_1 produces its output. The output is *accept* if M_1 is now in an accept state and *reject* if it is not.

For example, when we feed the input string 1101 to the machine M_1 in Figure 1.4, the processing proceeds as follows.

1. Start in state q_1 .
2. Read 1, follow transition from q_1 to q_2 .
3. Read 1, follow transition from q_2 to q_2 .
4. Read 0, follow transition from q_2 to q_3 .
5. Read 1, follow transition from q_3 to q_2 .
6. *Accept* because M_1 is in an accept state q_2 at the end of the input.

Experimenting with this machine on a variety of input strings reveals that it accepts the strings 1, 01, 11, and 0101010101. In fact, M_1 accepts any string that ends with a 1, as it goes to its accept state q_2 whenever it reads the symbol 1. In addition, it accepts strings 100, 0100, 110000, and 0101000000, and any string that ends with an even number of 0s following the last 1. It rejects other strings, such as 0, 10, 101000. Can you describe the language consisting of all strings that M_1 accepts? We will do so shortly.

FORMAL DEFINITION OF A FINITE AUTOMATON

In the preceding section we used state diagrams to introduce finite automata. Now we define finite automata formally. Although state diagrams are easier to grasp intuitively, we need the formal definition, too, for two specific reasons.

First, a formal definition is precise. It resolves any uncertainties about what is allowed in a finite automaton. If you were uncertain about whether finite automata were allowed to have 0 accept states or whether they must have exactly one transition exiting every state for each possible input symbol, you could consult the formal definition and verify that the answer is yes in both cases. Second, a formal definition provides notation. Good notation helps you think and express your thoughts clearly.

The language of a formal definition is somewhat arcane, having some similarity to the language of a legal document. Both need to be precise, and every detail must be spelled out.

A finite automaton has several parts. It has a set of states and rules for going from one state to another, depending on the input symbol. It has an input alphabet that indicates the allowed input symbols. It has a start state and a set of accept states. The formal definition says that a finite automaton is a list of those five objects: set of states, input alphabet, rules for moving, start state, and accept states. In mathematical language a list of five elements is often called a 5-tuple. Hence we define a finite automaton to be a 5-tuple consisting of these five parts.

We use something called a *transition function*, frequently denoted δ , to define the rules for moving. If the finite automaton has an arrow from a state x to a state y labeled with the input symbol 1, that means that, if the automaton is in state x when it reads a 1, it then moves to state y . We can indicate the same thing with the transition function by saying that $\delta(x, 1) = y$. This notation is a kind of mathematical shorthand. Putting it all together we arrive at the formal definition of finite automata.

DEFINITION 1.5

A *finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the *states*,
2. Σ is a finite set called the *alphabet*,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the *transition function*,¹
4. $q_0 \in Q$ is the *start state*, and
5. $F \subseteq Q$ is the *set of accept states*.²

¹Refer back to page 7 if you are uncertain about the meaning of $\delta: Q \times \Sigma \longrightarrow Q$.

²Accept states sometimes are called *final states*.

The formal definition precisely describes what we mean by a finite automaton. For example, returning to the earlier question of whether 0 accept states is allowable, you can see that setting F to be the empty set \emptyset yields 0 accept states, which is allowable. Furthermore, the transition function δ specifies exactly one next state for each possible combination of a state and an input symbol. That answers our other question affirmatively, showing that exactly one transition arrow exits every state for each possible input symbol.

We can use the notation of the formal definition to describe individual finite automata by specifying each of the five parts listed in Definition 1.5. For example, let's return to the finite automaton M_1 we discussed earlier, redrawn here for convenience.

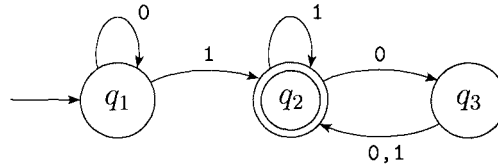


FIGURE 1.6
The finite automaton M_1

We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0, 1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state, and
5. $F = \{q_2\}$.

If A is the set of all strings that machine M accepts, we say that A is the **language of machine M** and write $L(M) = A$. We say that **M recognizes A** or that **M accepts A** . Because the term *accept* has different meanings when we refer to machines accepting strings and machines accepting languages, we prefer the term *recognize* for languages in order to avoid confusion.

A machine may accept several strings, but it always recognizes only one language. If the machine accepts no strings, it still recognizes one language—namely, the empty language \emptyset .

In our example, let

$$A = \{w \mid w \text{ contains at least one 1 and} \\ \text{an even number of 0s follow the last 1}\}.$$

Then $L(M_1) = A$, or equivalently, M_1 recognizes A .

EXAMPLES OF FINITE AUTOMATA

EXAMPLE 1.7

Here is the state diagram of finite automaton M_2 .

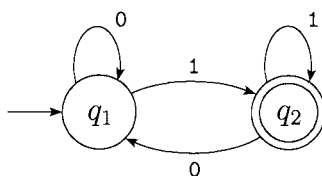


FIGURE 1.8

State diagram of the two-state finite automaton M_2

In the formal description $M_2 = (\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\})$. The transition function δ is

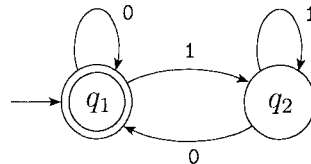
	0	1
q_1	q_1	q_2
q_2	q_1	q_2

Remember that the state diagram of M_2 and the formal description of M_2 contain the same information, only in different form. You can always go from one to the other if necessary.

A good way to begin understanding any machine is to try it on some sample input strings. When you do these “experiments” to see how the machine is working, its method of functioning often becomes apparent. On the sample string 1101 the machine M_2 starts in its start state q_1 and proceeds first to state q_2 after reading the first 1, and then to states q_2 , q_1 , and q_2 after reading 1, 0, and 1. The string is accepted because q_2 is an accept state. But string 110 leaves M_2 in state q_1 , so it is rejected. After trying a few more examples, you would see that M_2 accepts all strings that end in a 1. Thus $L(M_2) = \{w \mid w \text{ ends in a 1}\}$.

EXAMPLE 1.9

Consider the finite automaton M_3 .

**FIGURE 1.10**

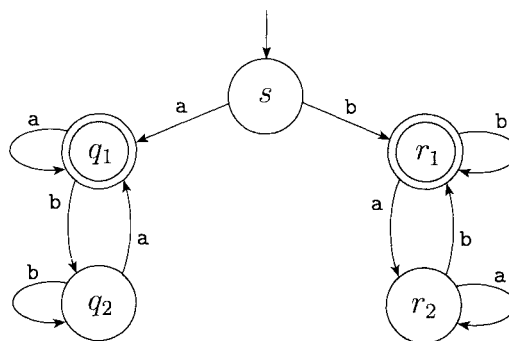
State diagram of the two-state finite automaton M_3

Machine M_3 is similar to M_2 except for the location of the accept state. As usual, the machine accepts all strings that leave it in an accept state when it has finished reading. Note that, because the start state is also an accept state, M_3 accepts the empty string ε . As soon as a machine begins reading the empty string it is at the end, so if the start state is an accept state, ε is accepted. In addition to the empty string, this machine accepts any string ending with a 0. Here,

$$L(M_3) = \{w \mid w \text{ is the empty string } \varepsilon \text{ or ends in a } 0\}.$$

EXAMPLE 1.11

The following figure shows a five-state machine M_4

**FIGURE 1.12**

Finite automaton M_4

Machine M_4 has two accept states, q_1 and r_1 , and operates over the alphabet $\Sigma = \{a, b\}$. Some experimentation shows that it accepts strings a , b , aa , bb , and bab , but not strings ab , ba , or $bbba$. This machine begins in state s , and after it reads the first symbol in the input, it goes either left into the q states or right into the r states. In both cases it can never return to the start state (in contrast to the previous examples), as it has no way to get from any other state back to s . If the first symbol in the input string is a , then it goes left and accepts when the string ends with an a . Similarly, if the first symbol is a b , the machine goes right and accepts when the string ends in b . So M_4 accepts all strings that start and end with a or that start and end with b . In other words, M_4 accepts strings that start and end with the same symbol.

EXAMPLE 1.13

Figure 1.14 shows machine M_5 , which has a four-symbol input alphabet, $\Sigma = \{\langle \text{RESET} \rangle, 0, 1, 2\}$. We treat $\langle \text{RESET} \rangle$ as a single symbol.

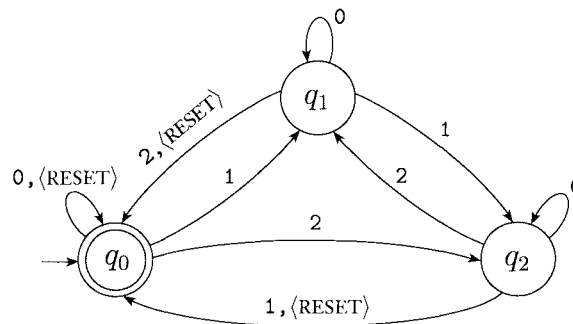


FIGURE 1.14

Finite automaton M_5

Machine M_5 keeps a running count of the sum of the numerical input symbols it reads, modulo 3. Every time it receives the $\langle \text{RESET} \rangle$ symbol it resets the count to 0. It accepts if the sum is 0, modulo 3, or in other words, if the sum is a multiple of 3.

Describing a finite automaton by state diagram is not possible in some cases. That may occur when the diagram would be too big to draw or if, as in this example, the description depends on some unspecified parameter. In these cases we resort to a formal description to specify the machine.

EXAMPLE 1.15

Consider a generalization of Example 1.13, using the same four-symbol alphabet Σ . For each $i \geq 1$ let A_i be the language of all strings where the sum of the numbers is a multiple of i , except that the sum is reset to 0 whenever the symbol $\langle \text{RESET} \rangle$ appears. For each A_i we give a finite automaton B_i , recognizing A_i . We describe the machine B_i formally as follows: $B_i = (Q_i, \Sigma, \delta_i, q_0, \{q_0\})$, where Q_i is the set of i states $\{q_0, q_1, q_2, \dots, q_{i-1}\}$, and we design the transition function δ_i so that for each j , if B_i is in q_j , the running sum is j , modulo i . For each q_j let

$$\begin{aligned}\delta_i(q_j, 0) &= q_j, \\ \delta_i(q_j, 1) &= q_k, \text{ where } k = j + 1 \text{ modulo } i, \\ \delta_i(q_j, 2) &= q_k, \text{ where } k = j + 2 \text{ modulo } i, \text{ and} \\ \delta_i(q_j, \langle \text{RESET} \rangle) &= q_0.\end{aligned}$$

FORMAL DEFINITION OF COMPUTATION

So far we have described finite automata informally, using state diagrams, and with a formal definition, as a 5-tuple. The informal description is easier to grasp at first, but the formal definition is useful for making the notion precise, resolving any ambiguities that may have occurred in the informal description. Next we do the same for a finite automaton's computation. We already have an informal idea of the way it computes, and we now formalize it mathematically.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \dots w_n$ be a string where each w_i is a member of the alphabet Σ . Then M **accepts** w if a sequence of states r_0, r_1, \dots, r_n in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \dots, n-1$, and
3. $r_n \in F$.

Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accept state. We say that M **recognizes language** A if $A = \{w \mid M \text{ accepts } w\}$.

DEFINITION 1.16

A language is called a **regular language** if some finite automaton recognizes it.

EXAMPLE 1.17

Take machine M_5 from Example 1.13. Let w be the string

$$10\langle\text{RESET}\rangle 22\langle\text{RESET}\rangle 012$$

Then M_5 accepts w according to the formal definition of computation because the sequence of states it enters when computing on w is

$$q_0, q_1, q_1, q_0, q_2, q_1, q_0, q_0, q_1, q_0,$$

which satisfies the three conditions. The language of M_5 is

$$L(M_5) = \{w \mid \text{the sum of the symbols in } w \text{ is } 0 \text{ modulo } 3, \\ \text{except that } \langle\text{RESET}\rangle \text{ resets the count to } 0\}.$$

As M_5 recognizes this language, it is a regular language.

DESIGNING FINITE AUTOMATA

Whether it be of automaton or artwork, design is a creative process. As such it cannot be reduced to a simple recipe or formula. However, you might find a particular approach helpful when designing various types of automata. That is, put *yourself* in the place of the machine you are trying to design and then see how you would go about performing the machine's task. Pretending that you are the machine is a psychological trick that helps engage your whole mind in the design process.

Let's design a finite automaton using the "reader as automaton" method just described. Suppose that you are given some language and want to design a finite automaton that recognizes it. Pretending to be the automaton, you receive an input string and must determine whether it is a member of the language the automaton is supposed to recognize. You get to see the symbols in the string one by one. After each symbol you must decide whether the string seen so far is in the language. The reason is that you, like the machine, don't know when the end of the string is coming, so you must always be ready with the answer.

First, in order to make these decisions, you have to figure out what you need to remember about the string as you are reading it. Why not simply remember all you have seen? Bear in mind that you are pretending to be a finite automaton and that this type of machine has only a finite number of states, which means a finite memory. Imagine that the input is extremely long—say, from here to the moon—so that you could not possibly remember the entire thing. You have a finite memory—say, a single sheet of paper—which has a limited storage capacity. Fortunately, for many languages you don't need to remember the entire input. You need to remember only certain crucial information. Exactly which information is crucial depends on the particular language considered.

For example, suppose that the alphabet is $\{0,1\}$ and that the language consists of all strings with an odd number of 1s. You want to construct a finite automaton E_1 to recognize this language. Pretending to be the automaton, you start getting

an input string of 0s and 1s symbol by symbol. Do you need to remember the entire string seen so far in order to determine whether the number of 1s is odd? Of course not. Simply remember whether the number of 1s seen so far is even or odd and keep track of this information as you read new symbols. If you read a 1, flip the answer, but if you read a 0, leave the answer as is.

But how does this help you design E_1 ? Once you have determined the necessary information to remember about the string as it is being read, you represent this information as a finite list of possibilities. In this instance, the possibilities would be

1. even so far, and
2. odd so far.

Then you assign a state to each of the possibilities. These are the states of E_1 , as shown here.

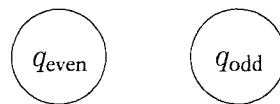


FIGURE 1.18

The two states q_{even} and q_{odd}

Next, you assign the transitions by seeing how to go from one possibility to another upon reading a symbol. So, if state q_{even} represents the even possibility and state q_{odd} represents the odd possibility, you would set the transitions to flip state on a 1 and stay put on a 0, as shown here.

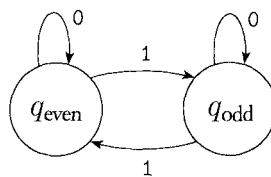


FIGURE 1.19

Transitions telling how the possibilities rearrange

Next, you set the start state to be the state corresponding to the possibility associated with having seen 0 symbols so far (the empty string ϵ). In this case the start state corresponds to state q_{even} because 0 is an even number. Last, set the accept states to be those corresponding to possibilities where you want to accept the input string. Set q_{odd} to be an accept state because you want to accept when

you have seen an odd number of 1s. These additions are shown in the following figure.

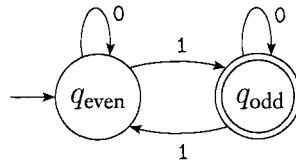


FIGURE 1.20
Adding the start and accept states

EXAMPLE 1.21

This example shows how to design a finite automaton E_2 to recognize the regular language of all strings that contain the string 001 as a substring. For example, 0010, 1001, 001, and 11111110011111 are all in the language, but 11 and 0000 are not. How would you recognize this language if you were pretending to be E_2 ? As symbols come in, you would initially skip over all 1s. If you come to a 0, then you note that you may have just seen the first of the three symbols in the pattern 001 you are seeking. If at this point you see a 1, there were too few 0s, so you go back to skipping over 1s. But if you see a 0 at that point, you should remember that you have just seen two symbols of the pattern. Now you simply need to continue scanning until you see a 1. If you find it, remember that you succeeded in finding the pattern and continue reading the input string until you get to the end.

So there are four possibilities: You

1. haven't just seen any symbols of the pattern,
2. have just seen a 0,
3. have just seen 00, or
4. have seen the entire pattern 001.

Assign the states q , q_0 , q_{00} , and q_{001} to these possibilities. You can assign the transitions by observing that from q reading a 1 you stay in q , but reading a 0 you move to q_0 . In q_0 reading a 1 you return to q , but reading a 0 you move to q_{00} . In q_{00} , reading a 1 you move to q_{001} , but reading a 0 leaves you in q_{00} . Finally, in q_{001} reading a 0 or a 1 leaves you in q_{001} . The start state is q , and the only accept state is q_{001} , as shown in Figure 1.22.

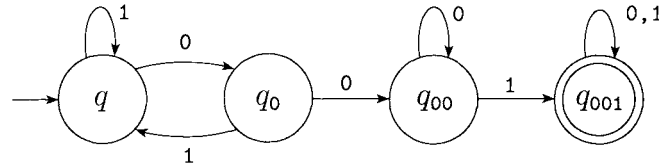


FIGURE 1.22
Accepts strings containing 001

THE REGULAR OPERATIONS

In the preceding two sections we introduced and defined finite automata and regular languages. We now begin to investigate their properties. Doing so will help develop a toolbox of techniques to use when you design automata to recognize particular languages. The toolbox also will include ways of proving that certain other languages are nonregular (i.e., beyond the capability of finite automata).

In arithmetic, the basic objects are numbers and the tools are operations for manipulating them, such as $+$ and \times . In the theory of computation the objects are languages and the tools include operations specifically designed for manipulating them. We define three operations on languages, called the **regular operations**, and use them to study properties of the regular languages.

DEFINITION 1.23

Let A and B be languages. We define the regular operations **union**, **concatenation**, and **star** as follows.

- **Union:** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
- **Concatenation:** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$.
- **Star:** $A^* = \{x_1x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$.

You are already familiar with the union operation. It simply takes all the strings in both A and B and lumps them together into one language.

The concatenation operation is a little trickier. It attaches a string from A in front of a string from B in all possible ways to get the strings in the new language.

The star operation is a bit different from the other two because it applies to a single language rather than to two different languages. That is, the star operation is a **unary operation** instead of a **binary operation**. It works by attaching any number of strings in A together to get a string in the new language. Because

“any number” includes 0 as a possibility, the empty string ϵ is always a member of A^* , no matter what A is.

EXAMPLE 1.24

Let the alphabet Σ be the standard 26 letters $\{a, b, \dots, z\}$. If $A = \{\text{good}, \text{bad}\}$ and $B = \{\text{boy}, \text{girl}\}$, then

$$A \cup B = \{\text{good}, \text{bad}, \text{boy}, \text{girl}\},$$

$$A \circ B = \{\text{goodboy}, \text{goodgirl}, \text{badboy}, \text{badgirl}\}, \text{ and}$$

$$A^* = \{\epsilon, \text{good}, \text{bad}, \text{goodgood}, \text{goodbad}, \text{badgood}, \text{badbad}, \text{goodgoodgood}, \text{goodgoodbad}, \text{goodbadgood}, \text{goodbadbad}, \dots\}.$$

Let $\mathcal{N} = \{1, 2, 3, \dots\}$ be the set of natural numbers. When we say that \mathcal{N} is *closed under multiplication* we mean that, for any x and y in \mathcal{N} , the product $x \times y$ also is in \mathcal{N} . In contrast \mathcal{N} is not closed under division, as 1 and 2 are in \mathcal{N} but $1/2$ is not. Generally speaking, a collection of objects is **closed** under some operation if applying that operation to members of the collection returns an object still in the collection. We show that the collection of regular languages is closed under all three of the regular operations. In Section 1.3 we show that these are useful tools for manipulating regular languages and understanding the power of finite automata. We begin with the union operation.

THEOREM 1.25

The class of regular languages is closed under the union operation.

In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

PROOF IDEA We have regular languages A_1 and A_2 and want to show that $A_1 \cup A_2$ also is regular. Because A_1 and A_2 are regular, we know that some finite automaton M_1 recognizes A_1 and some finite automaton M_2 recognizes A_2 . To prove that $A_1 \cup A_2$ is regular we demonstrate a finite automaton, call it M , that recognizes $A_1 \cup A_2$.

This is a proof by construction. We construct M from M_1 and M_2 . Machine M must accept its input exactly when either M_1 or M_2 would accept it in order to recognize the union language. It works by *simulating* both M_1 and M_2 and accepting if either of the simulations accept.

How can we make machine M simulate M_1 and M_2 ? Perhaps it first simulates M_1 on the input and then simulates M_2 on the input. But we must be careful here! Once the symbols of the input have been read and used to simulate M_1 , we can't “rewind the input tape” to try the simulation on M_2 . We need another approach.

Pretend that you are M . As the input symbols arrive one by one, you simulate both M_1 and M_2 simultaneously. That way only one pass through the input is necessary. But can you keep track of both simulations with finite memory? All you need to remember is the state that each machine would be in if it had read up to this point in the input. Therefore you need to remember a pair of states. How many possible pairs are there? If M_1 has k_1 states and M_2 has k_2 states, the number of pairs of states, one from M_1 and the other from M_2 , is the product $k_1 \times k_2$. This product will be the number of states in M , one for each pair. The transitions of M go from pair to pair, updating the current state for both M_1 and M_2 . The accept states of M are those pairs wherein either M_1 or M_2 is in an accept state.

PROOF

Let M_1 recognize A_1 , where $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, and
 M_2 recognize A_2 , where $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.

Construct M to recognize $A_1 \cup A_2$, where $M = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$.
 This set is the **Cartesian product** of sets Q_1 and Q_2 and is written $Q_1 \times Q_2$. It is the set of all pairs of states, the first from Q_1 and the second from Q_2 .
2. Σ , the alphabet, is the same as in M_1 and M_2 . In this theorem and in all subsequent similar theorems, we assume for simplicity that both M_1 and M_2 have the same input alphabet Σ . The theorem remains true if they have different alphabets, Σ_1 and Σ_2 . We would then modify the proof to let $\Sigma = \Sigma_1 \cup \Sigma_2$.
3. δ , the transition function, is defined as follows. For each $(r_1, r_2) \in Q$ and each $a \in \Sigma$, let

$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)).$$

Hence δ gets a state of M (which actually is a pair of states from M_1 and M_2), together with an input symbol, and returns M 's next state.

4. q_0 is the pair (q_1, q_2) .
5. F is the set of pairs in which either member is an accept state of M_1 or M_2 . We can write it as

$$F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\}.$$

This expression is the same as $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$. (Note that it is *not* the same as $F = F_1 \times F_2$. What would that give us instead?³)

³ This expression would define M 's accept states to be those for which *both* members of the pair are accept states. In this case M would accept a string only if both M_1 and M_2 accept it, so the resulting language would be the *intersection* and not the union. In fact, this result proves that the class of regular languages is closed under intersection.

This concludes the construction of the finite automaton M that recognizes the union of A_1 and A_2 . This construction is fairly simple, and thus its correctness is evident from the strategy described in the proof idea. More complicated constructions require additional discussion to prove correctness. A formal correctness proof for a construction of this type usually proceeds by induction. For an example of a construction proved correct, see the proof of Theorem 1.54. Most of the constructions that you will encounter in this course are fairly simple and so do not require a formal correctness proof.

We have just shown that the union of two regular languages is regular, thereby proving that the class of regular languages is closed under the union operation. We now turn to the concatenation operation and attempt to show that the class of regular languages is closed under that operation, too.

THEOREM 1.26

The class of regular languages is closed under the concatenation operation.

In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.

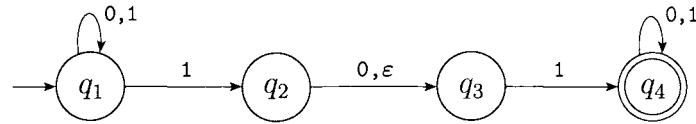
To prove this theorem let's try something along the lines of the proof of the union case. As before, we can start with finite automata M_1 and M_2 recognizing the regular languages A_1 and A_2 . But now, instead of constructing automaton M to accept its input if either M_1 or M_2 accept, it must accept if its input can be broken into two pieces, where M_1 accepts the first piece and M_2 accepts the second piece. The problem is that M doesn't know where to break its input (i.e., where the first part ends and the second begins). To solve this problem we introduce a new technique called nondeterminism.

1.2

NONDETERMINISM

Nondeterminism is a useful concept that has had great impact on the theory of computation. So far in our discussion, every step of a computation follows in a unique way from the preceding step. When the machine is in a given state and reads the next input symbol, we know what the next state will be—it is determined. We call this *deterministic* computation. In a *nondeterministic* machine, several choices may exist for the next state at any point.

Nondeterminism is a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton. As Figure 1.27 shows, nondeterministic finite automata may have additional features.

**FIGURE 1.27**

The nondeterministic finite automaton N_1

The difference between a deterministic finite automaton, abbreviated DFA, and a nondeterministic finite automaton, abbreviated NFA, is immediately apparent. First, every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. The nondeterministic automaton shown in Figure 1.27 violates that rule. State q_1 has one exiting arrow for 0, but it has two for 1; q_2 has one arrow for 0, but it has none for 1. In an NFA a state may have zero, one, or many exiting arrows for each alphabet symbol.

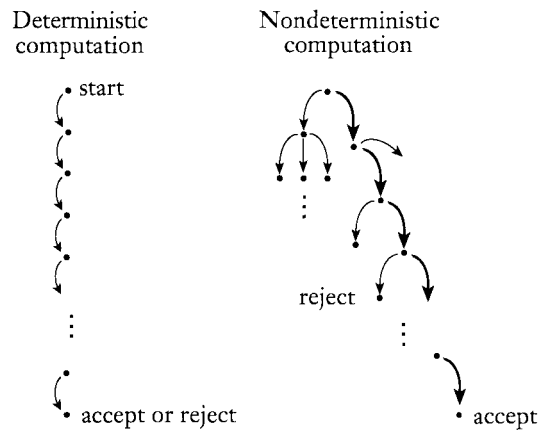
Second, in a DFA, labels on the transition arrows are symbols from the alphabet. This NFA has an arrow with the label ϵ . In general, an NFA may have arrows labeled with members of the alphabet or ϵ . Zero, one, or many arrows may exit from each state with the label ϵ .

How does an NFA compute? Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed. For example, say that we are in state q_1 in NFA N_1 and that the next input symbol is a 1. After reading that symbol, the machine splits into multiple copies of itself and follows *all* the possibilities in parallel. Each copy of the machine takes one of the possible ways to proceed and continues as before. If there are subsequent choices, the machine splits again. If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it. Finally, if *any one* of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

If a state with an ϵ symbol on an exiting arrow is encountered, something similar happens. Without reading any input, the machine splits into multiple copies, one following each of the exiting ϵ -labeled arrows and one staying at the current state. Then the machine proceeds nondeterministically as before.

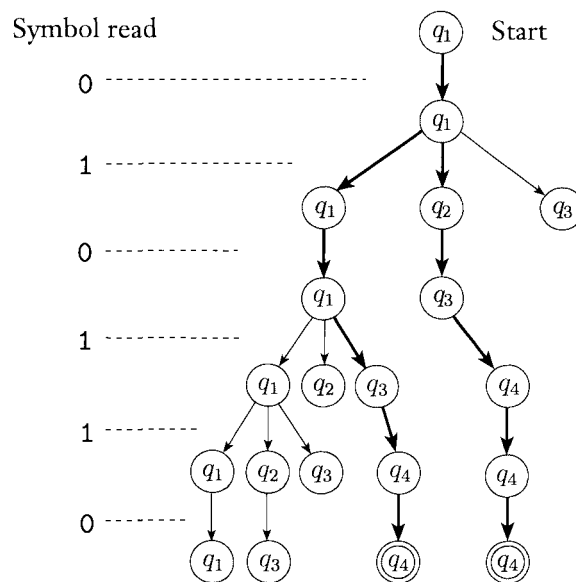
Nondeterminism may be viewed as a kind of parallel computation wherein multiple independent “processes” or “threads” can be running concurrently. When the NFA splits to follow several choices, that corresponds to a process “forking” into several children, each proceeding separately. If at least one of these processes accepts, then the entire computation accepts.

Another way to think of a nondeterministic computation is as a tree of possibilities. The root of the tree corresponds to the start of the computation. Every branching point in the tree corresponds to a point in the computation at which the machine has multiple choices. The machine accepts if at least one of the computation branches ends in an accept state, as shown in Figure 1.28.

**FIGURE 1.28**

Deterministic and nondeterministic computations with an accepting branch

Let's consider some sample runs of the NFA N_1 shown in Figure 1.27. The computation of N_1 on input 010110 is depicted in the following figure.

**FIGURE 1.29**

The computation of N_1 on input 010110

On input 010110 start in the start state q_1 and read the first symbol 0. From q_1 there is only one place to go on a 0—namely, back to q_1 , so remain there. Next read the second symbol 1. In q_1 on a 1 there are two choices: either stay in q_1 or move to q_2 . Nondeterministically, the machine splits in two to follow each choice. Keep track of the possibilities by placing a finger on each state where a machine could be. So you now have fingers on states q_1 and q_2 . An ε arrow exits state q_2 so the machine splits again; keep one finger on q_2 , and move the other to q_3 . You now have fingers on q_1 , q_2 , and q_3 .

When the third symbol 0 is read, take each finger in turn. Keep the finger on q_1 in place, move the finger on q_2 to q_3 , and remove the finger that has been on q_3 . That last finger had no 0 arrow to follow and corresponds to a process that simply “dies.” At this point you have fingers on states q_1 and q_3 .

When the fourth symbol 1 is read, split the finger on q_1 into fingers on states q_1 and q_2 , then further split the finger on q_2 to follow the ε arrow to q_3 , and move the finger that was on q_3 to q_4 . You now have a finger on each of the four states.

When the fifth symbol 1 is read, the fingers on q_1 and q_3 result in fingers on states q_1 , q_2 , q_3 , and q_4 , as you saw with the fourth symbol. The finger on state q_2 is removed. The finger that was on q_4 stays on q_4 . Now you have two fingers on q_4 , so remove one, because you only need to remember that q_4 is a possible state at this point, not that it is possible for multiple reasons.

When the sixth and final symbol 0 is read, keep the finger on q_1 in place, move the one on q_2 to q_3 , remove the one that was on q_3 , and leave the one on q_4 in place. You are now at the end of the string, and you accept if some finger is on an accept state. You have fingers on states q_1 , q_3 , and q_4 , and as q_4 is an accept state, N_1 accepts this string.

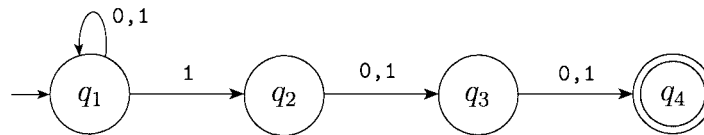
What does N_1 do on input 010? Start with a finger on q_1 . After reading the 0 you still have a finger only on q_1 , but after the 1 there are fingers on q_1 , q_2 , and q_3 (don’t forget the ε arrow). After the third symbol 0, remove the finger on q_3 , move the finger on q_2 to q_3 , and leave the finger on q_1 where it is. At this point you are at the end of the input, and as no finger is on an accept state, N_1 rejects this input.

By continuing to experiment in this way, you will see that N_1 accepts all strings that contain either 101 or 11 as a substring.

Nondeterministic finite automata are useful in several respects. As we will show, every NFA can be converted into an equivalent DFA, and constructing NFAs is sometimes easier than directly constructing DFAs. An NFA may be much smaller than its deterministic counterpart, or its functioning may be easier to understand. Nondeterminism in finite automata is also a good introduction to nondeterminism in more powerful computational models because finite automata are especially easy to understand. Now we turn to several examples of NFAs.

EXAMPLE 1.30

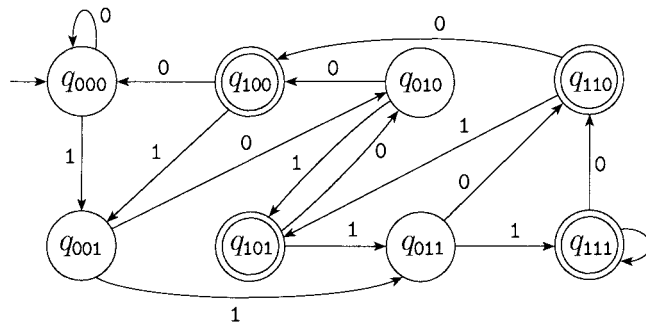
Let A be the language consisting of all strings over $\{0,1\}$ containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not). The following four-state NFA N_2 recognizes A .

**FIGURE 1.31**

The NFA N_2 recognizing A

One good way to view the computation of this NFA is to say that it stays in the start state q_1 until it “guesses” that it is three places from the end. At that point, if the input symbol is a 1, it branches to state q_2 and uses q_3 and q_4 to “check” on whether its guess was correct.

As mentioned, every NFA can be converted into an equivalent DFA, but sometimes that DFA may have many more states. The smallest DFA for A contains eight states. Furthermore, understanding the functioning of the NFA is much easier, as you may see by examining the following figure for the DFA.

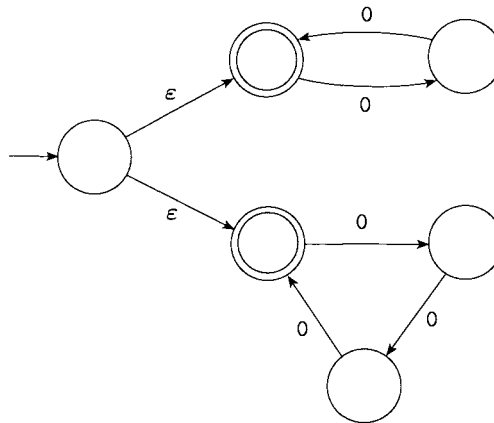
**FIGURE 1.32**

A DFA recognizing A

Suppose that we added ϵ to the labels on the arrows going from q_2 to q_3 and from q_3 to q_4 in machine N_2 in Figure 1.31. So both arrows would then have the label $0, 1, \epsilon$ instead of just $0, 1$. What language would N_2 recognize with this modification? Try modifying the DFA in Figure 1.32 to recognize that language.

EXAMPLE 1.33

Consider the following NFA N_3 that has an input alphabet $\{0\}$ consisting of a single symbol. An alphabet containing only one symbol is called a *unary alphabet*.

**FIGURE 1.34**

The NFA N_3

This machine demonstrates the convenience of having ε arrows. It accepts all strings of the form 0^k where k is a multiple of 2 or 3. (Remember that the superscript denotes repetition, not numerical exponentiation.) For example, N_3 accepts the strings ε , 00, 000, 0000, and 000000, but not 0 or 00000.

Think of the machine operating by initially guessing whether to test for a multiple of 2 or a multiple of 3 by branching into either the top loop or the bottom loop and then checking whether its guess was correct. Of course, we could replace this machine by one that doesn't have ε arrows or even any nondeterminism at all, but the machine shown is the easiest one to understand for this language.

EXAMPLE 1.35

We give another example of an NFA in the following figure. Practice with it to satisfy yourself that it accepts the strings ε , a, baba, and baa, but that it doesn't accept the strings b, bb, and babba. Later we use this machine to illustrate the procedure for converting NFAs to DFAs.

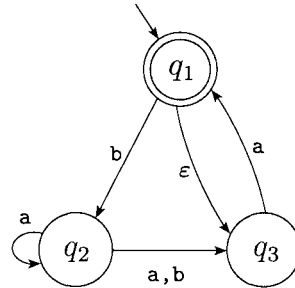


FIGURE 1.36
The NFA N_4

FORMAL DEFINITION OF A NONDETERMINISTIC FINITE AUTOMATON

The formal definition of a nondeterministic finite automaton is similar to that of a deterministic finite automaton. Both have states, an input alphabet, a transition function, a start state, and a collection of accept states. However, they differ in one essential way: in the type of transition function. In a DFA the transition function takes a state and an input symbol and produces the next state. In an NFA the transition function takes a state and an input symbol *or the empty string* and produces *the set of possible next states*. In order to write the formal definition, we need to set up some additional notation. For any set Q we write $\mathcal{P}(Q)$ to be the collection of all subsets of Q . Here $\mathcal{P}(Q)$ is called the **power set** of Q . For any alphabet Σ we write Σ_ϵ to be $\Sigma \cup \{\epsilon\}$. Now we can write the formal description of the type of the transition function in an NFA as $\delta: Q \times \Sigma_\epsilon \longrightarrow \mathcal{P}(Q)$.

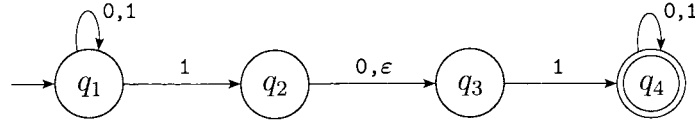
DEFINITION 1.37

A **nondeterministic finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta: Q \times \Sigma_\epsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

EXAMPLE 1.38

Recall the NFA N_1 :



The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0,1\}$,
3. δ is given as

	0	1	ϵ
q_1	$\{q_1\}$	$\{q_1, q_2\}$	\emptyset
q_2	$\{q_3\}$	\emptyset	$\{q_3\}$
q_3	\emptyset	$\{q_4\}$	\emptyset
q_4	$\{q_4\}$	$\{q_4\}$	\emptyset

4. q_1 is the start state, and
5. $F = \{q_4\}$.

The formal definition of computation for an NFA is similar to that for a DFA. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and w a string over the alphabet Σ . Then we say that N **accepts** w if we can write w as $w = y_1 y_2 \cdots y_m$, where each y_i is a member of Σ_ϵ and a sequence of states r_0, r_1, \dots, r_m exists in Q with three conditions:

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, \dots, m-1$, and
3. $r_m \in F$.

Condition 1 says that the machine starts out in the start state. Condition 2 says that state r_{i+1} is one of the allowable next states when N is in state r_i and reading y_{i+1} . Observe that $\delta(r_i, y_{i+1})$ is the *set* of allowable next states and so we say that r_{i+1} is a member of that set. Finally, condition 3 says that the machine accepts its input if the last state is an accept state.

EQUIVALENCE OF NFAS AND DFAS

Deterministic and nondeterministic finite automata recognize the same class of languages. Such equivalence is both surprising and useful. It is surprising because NFAs appear to have more power than DFAs, so we might expect that NFAs recognize more languages. It is useful because describing an NFA for a given language sometimes is much easier than describing a DFA for that language.

Say that two machines are **equivalent** if they recognize the same language.

THEOREM 1.39

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

PROOF IDEA If a language is recognized by an NFA, then we must show the existence of a DFA that also recognizes it. The idea is to convert the NFA into an equivalent DFA that simulates the NFA.

Recall the “reader as automaton” strategy for designing finite automata. How would you simulate the NFA if you were pretending to be a DFA? What do you need to keep track of as the input string is processed? In the examples of NFAs you kept track of the various branches of the computation by placing a finger on each state that could be active at given points in the input. You updated the simulation by moving, adding, and removing fingers according to the way the NFA operates. All you needed to keep track of was the set of states having fingers on them.

If k is the number of states of the NFA, it has 2^k subsets of states. Each subset corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^k states. Now we need to figure out which will be the start state and accept states of the DFA, and what will be its transition function. We can discuss this more easily after setting up some formal notation.

PROOF Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA recognizing some language A . We construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ recognizing A . Before doing the full construction, let's first consider the easier case wherein N has no ϵ arrows. Later we take the ϵ arrows into account.

1. $Q' = \mathcal{P}(Q)$.

Every state of M is a set of states of N . Recall that $\mathcal{P}(Q)$ is the set of subsets of Q .

2. For $R \in Q'$ and $a \in \Sigma$ let $\delta'(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ for some } r \in R\}$. If R is a state of M , it is also a set of states of N . When M reads a symbol a in state R , it shows where a takes each state in R . Because each state may go to a set of states, we take the union of all these sets. Another way to write this expression is

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).^4$$

3. $q_0' = \{q_0\}$.

M starts in the state corresponding to the collection containing just the start state of N .

4. $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$.

The machine M accepts if one of the possible states that N could be in at this point is an accept state.

⁴The notation $\bigcup_{r \in R} \delta(r, a)$ means: the union of the sets $\delta(r, a)$ for each possible r in R .

Now we need to consider the ε arrows. To do so we set up an extra bit of notation. For any state R of M we define $E(R)$ to be the collection of states that can be reached from R by going only along ε arrows, including the members of R themselves. Formally, for $R \subseteq Q$ let

$$E(R) = \{q \mid q \text{ can be reached from } R \text{ by traveling along 0 or more } \varepsilon \text{ arrows}\}.$$

Then we modify the transition function of M to place additional fingers on all states that can be reached by going along ε arrows after every step. Replacing $\delta(r, a)$ by $E(\delta(r, a))$ achieves this effect. Thus

$$\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ for some } r \in R\}.$$

Additionally we need to modify the start state of M to move the fingers initially to all possible states that can be reached from the start state of N along the ε arrows. Changing q_0' to be $E(\{q_0\})$ achieves this effect. We have now completed the construction of the DFA M that simulates the NFA N .

The construction of M obviously works correctly. At every step in the computation of M on an input, it clearly enters a state that corresponds to the subset of states that N could be in at that point. Thus our proof is complete.

If the construction used in the preceding proof were more complex we would need to prove that it works as claimed. Usually such proofs proceed by induction on the number of steps of the computation. Most of the constructions that we use in this book are straightforward and so do not require such a correctness proof. An example of a more complex construction that we do prove correct appears in the proof of Theorem 1.54.

Theorem 1.39 states that every NFA can be converted into an equivalent DFA. Thus nondeterministic finite automata give an alternative way of characterizing the regular languages. We state this fact as a corollary of Theorem 1.39.

COROLLARY 1.40

A language is regular if and only if some nondeterministic finite automaton recognizes it.

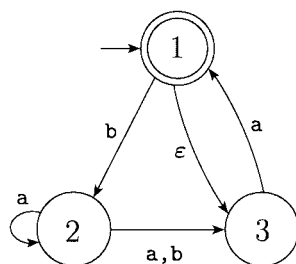
One direction of the “if and only if” condition states that a language is regular if some NFA recognizes it. Theorem 1.39 shows that any NFA can be converted into an equivalent DFA. Consequently, if an NFA recognizes some language, so does some DFA, and hence the language is regular. The other direction of the “if and only if” condition states that a language is regular only if some NFA recognizes it. That is, if a language is regular, some NFA must be recognizing it. Obviously, this condition is true because a regular language has a DFA recognizing it and any DFA is also an NFA.

EXAMPLE 1.41

Let's illustrate the procedure we gave in the proof of Theorem 1.39 for converting an NFA to a DFA by using the machine N_4 that appears in Example 1.35. For clarity, we have relabeled the states of N_4 to be $\{1, 2, 3\}$. Thus in the formal description of $N_4 = (Q, \{a, b\}, \delta, 1, \{1\})$, the set of states Q is $\{1, 2, 3\}$ as shown in the following figure.

To construct a DFA D that is equivalent to N_4 , we first determine D 's states. N_4 has three states, $\{1, 2, 3\}$, so we construct D with eight states, one for each subset of N_4 's states. We label each of D 's states with the corresponding subset. Thus D 's state set is

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

**FIGURE 1.42**

The NFA N_4

Next, we determine the start and accept states of D . The start state is $E(\{1\})$, the set of states that are reachable from 1 by traveling along ϵ arrows, plus 1 itself. An ϵ arrow goes from 1 to 3, so $E(\{1\}) = \{1, 3\}$. The new accept states are those containing N_4 's accept state; thus $\{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$.

Finally, we determine D 's transition function. Each of D 's states goes to one place on input a and one place on input b . We illustrate the process of determining the placement of D 's transition arrows with a few examples.

In D , state $\{2\}$ goes to $\{2, 3\}$ on input a , because in N_4 , state 2 goes to both 2 and 3 on input a and we can't go farther from 2 or 3 along ϵ arrows. State $\{2\}$ goes to state $\{3\}$ on input b , because in N_4 , state 2 goes only to state 3 on input b and we can't go farther from 3 along ϵ arrows.

State $\{1\}$ goes to \emptyset on a , because no a arrows exit it. It goes to $\{2\}$ on b . Note that the procedure in Theorem 1.39 specifies that we follow the ϵ arrows *after* each input symbol is read. An alternative procedure based on following the ϵ arrows before reading each input symbol works equally well, but that method is not illustrated in this example.

State $\{3\}$ goes to $\{1, 3\}$ on a , because in N_4 , state 3 goes to 1 on a and 1 in turn goes to 3 with an ϵ arrow. State $\{3\}$ on b goes to \emptyset .

State $\{1,2\}$ on a goes to $\{2,3\}$ because 1 points at no states with a arrows and 2 points at both 2 and 3 with a arrows and neither points anywhere with ϵ arrows. State $\{1,2\}$ on b goes to $\{2,3\}$. Continuing in this way we obtain the following diagram for D .

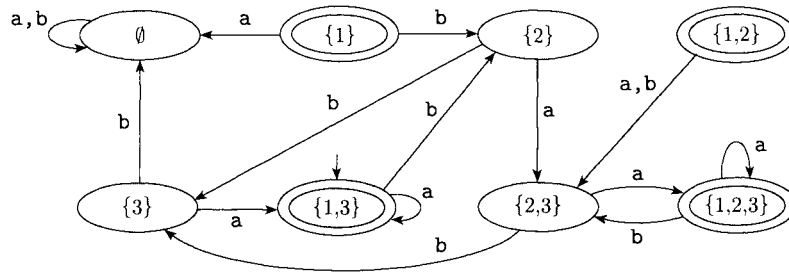


FIGURE 1.43
A DFA D that is equivalent to the NFA N_4

We may simplify this machine by observing that no arrows point at states $\{1\}$ and $\{1,2\}$, so they may be removed without affecting the performance of the machine. Doing so yields the following figure.

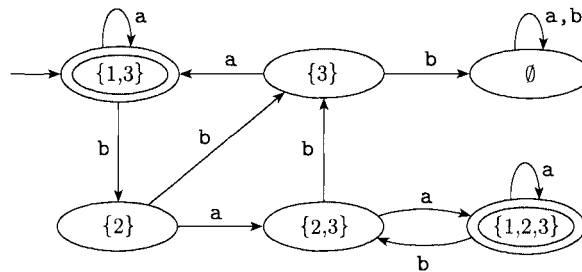


FIGURE 1.44
DFA D after removing unnecessary states

CLOSURE UNDER THE REGULAR OPERATIONS

Now we return to the closure of the class of regular languages under the regular operations that we began in Section 1.1. Our aim is to prove that the union, concatenation, and star of regular languages are still regular. We abandoned the original attempt to do so when dealing with the concatenation operation was too complicated. The use of nondeterminism makes the proofs much easier.

First, let's consider again closure under union. Earlier we proved closure under union by simulating deterministically both machines simultaneously via a Cartesian product construction. We now give a new proof to illustrate the technique of nondeterminism. Reviewing the first proof, appearing on page 45, may be worthwhile to see how much easier and more intuitive the new proof is.

THEOREM 1.45

The class of regular languages is closed under the union operation.

PROOF IDEA We have regular languages A_1 and A_2 and want to prove that $A_1 \cup A_2$ is regular. The idea is to take two NFAs, N_1 and N_2 for A_1 and A_2 , and combine them into one new NFA, N .

Machine N must accept its input if either N_1 or N_2 accepts this input. The new machine has a new start state that branches to the start states of the old machines with ϵ arrows. In this way the new machine nondeterministically guesses which of the two machines accepts the input. If one of them accepts the input, N will accept it, too.

We represent this construction in the following figure. On the left, we indicate the start and accept states of machines N_1 and N_2 with large circles and some additional states with small circles. On the right, we show how to combine N_1 and N_2 into N by adding additional transition arrows.

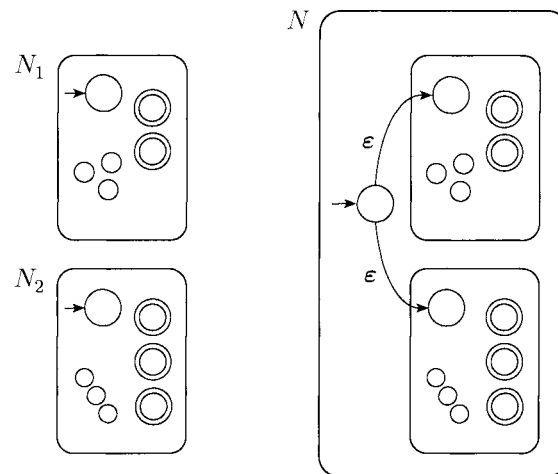


FIGURE 1.46

Construction of an NFA N to recognize $A_1 \cup A_2$

PROOF

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1 , and
 $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2 .

Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \cup A_2$.

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.
The states of N are all the states of N_1 and N_2 , with the addition of a new start state q_0 .
2. The state q_0 is the start state of N .
3. The accept states $F = F_1 \cup F_2$.
The accept states of N are all the accept states of N_1 and N_2 . That way N accepts if either N_1 accepts or N_2 accepts.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon. \end{cases}$$

Now we can prove closure under concatenation. Recall that earlier, without nondeterminism, completing the proof would have been difficult.

THEOREM 1.47

The class of regular languages is closed under the concatenation operation.

PROOF IDEA We have regular languages A_1 and A_2 and want to prove that $A_1 \circ A_2$ is regular. The idea is to take two NFAs, N_1 and N_2 for A_1 and A_2 , and combine them into a new NFA N as we did for the case of union, but this time in a different way, as shown in Figure 1.48.

Assign N 's start state to be the start state of N_1 . The accept states of N_1 have additional ε arrows that nondeterministically allow branching to N_2 whenever N_1 is in an accept state, signifying that it has found an initial piece of the input that constitutes a string in A_1 . The accept states of N are the accept states of N_2 only. Therefore it accepts when the input can be split into two parts, the first accepted by N_1 and the second by N_2 . We can think of N as nondeterministically guessing where to make the split.

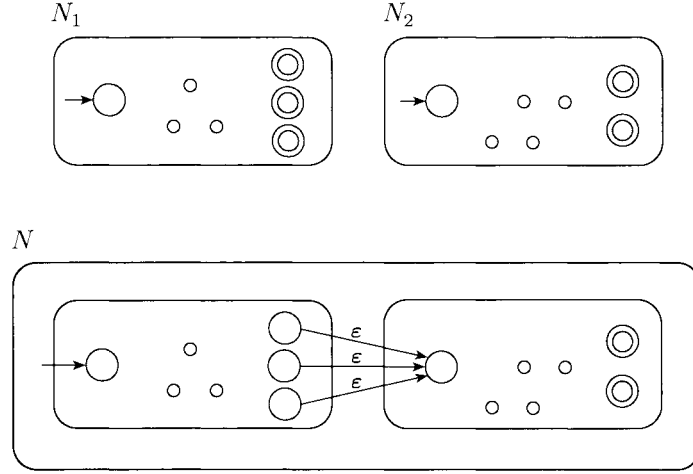


FIGURE 1.48
Construction of N to recognize $A_1 \circ A_2$

PROOF

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1 , and
 $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2 .

Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognize $A_1 \circ A_2$.

1. $Q = Q_1 \cup Q_2$.
The states of N are all the states of N_1 and N_2 .
2. The state q_1 is the same as the start state of N_1 .
3. The accept states F_2 are the same as the accept states of N_2 .
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\epsilon$,

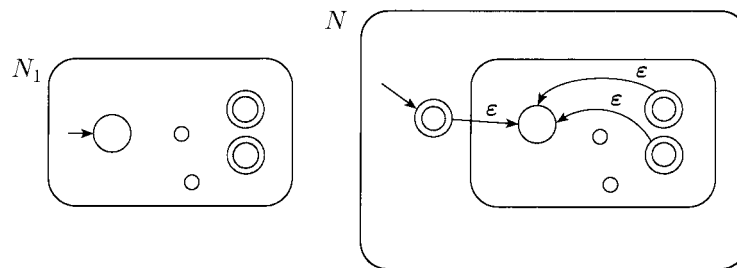
$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q, a) & q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q, a) \cup \{q_2\} & q \in F_1 \text{ and } a = \epsilon \\ \delta_2(q, a) & q \in Q_2. \end{cases}$$

THEOREM 1.49

The class of regular languages is closed under the star operation.

PROOF IDEA We have a regular language A_1 and want to prove that A_1^* also is regular. We take an NFA N_1 for A_1 and modify it to recognize A_1^* , as shown in the following figure. The resulting NFA N will accept its input whenever it can be broken into several pieces and N_1 accepts each piece.

We can construct N like N_1 with additional ϵ arrows returning to the start state from the accept states. This way, when processing gets to the end of a piece that N_1 accepts, the machine N has the option of jumping back to the start state to try to read in another piece that N_1 accepts. In addition we must modify N so that it accepts ϵ , which always is a member of A_1^* . One (slightly bad) idea is simply to add the start state to the set of accept states. This approach certainly adds ϵ to the recognized language, but it may also add other, undesired strings. Exercise 1.15 asks for an example of the failure of this idea. The way to fix it is to add a new start state, which also is an accept state, and which has an ϵ arrow to the old start state. This solution has the desired effect of adding ϵ to the language without adding anything else.

**FIGURE 1.50**

Construction of N to recognize A^*

PROOF Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1 . Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize A_1^* .

1. $Q = \{q_0\} \cup Q_1$.
The states of N are the states of N_1 plus a new start state.
2. The state q_0 is the new start state.
3. $F = \{q_0\} \cup F_1$.
The accept states are the old accept states plus the new start state.

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

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

1.3

REGULAR EXPRESSIONS

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

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

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

$$(0 \cup 1)0^*.$$

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

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

EXAMPLE 1.51

Another example of a regular expression is

$$(0 \cup 1)^*$$

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

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

FORMAL DEFINITION OF A REGULAR EXPRESSION**DEFINITION 1.52**

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

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

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

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

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

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

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

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

EXAMPLE 1.53

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

1. $0^*10^* = \{w \mid w \text{ contains a single } 1\}$.
2. $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}$.
3. $\Sigma^*001\Sigma^* = \{w \mid w \text{ contains the string } 001 \text{ as a substring}\}$.
4. $1^*(01^+)^* = \{w \mid \text{every } 0 \text{ in } w \text{ is followed by at least one } 1\}$.
5. $(\Sigma\Sigma)^* = \{w \mid w \text{ is a string of even length}\}$.⁵
6. $(\Sigma\Sigma\Sigma)^* = \{w \mid \text{the length of } w \text{ is a multiple of three}\}$.
7. $01 \cup 10 = \{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w \text{ starts and ends with the same symbol}\}$.
9. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.

The expression $0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$, so the concatenation operation adds either 0 or ε before every string in 1^* .

10. $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.

11. $1^*\emptyset = \emptyset$.

Concatenating the empty set to any set yields the empty set.

12. $\emptyset^* = \{\varepsilon\}$.

The star operation puts together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, giving only the empty string.

☞

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

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

$$R \cup \emptyset = R.$$

Adding the empty language to any other language will not change it.

$$R \circ \varepsilon = R.$$

Joining the empty string to any string will not change it.

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

$$R \cup \varepsilon \text{ may not equal } R.$$

For example, if $R = 0$, then $L(R) = \{0\}$ but $L(R \cup \varepsilon) = \{0, \varepsilon\}$.

$$R \circ \emptyset \text{ may not equal } R.$$

For example, if $R = 0$, then $L(R) = \{0\}$ but $L(R \circ \emptyset) = \emptyset$.

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

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

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

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

EQUIVALENCE WITH FINITE AUTOMATA

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

THEOREM 1.54

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

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

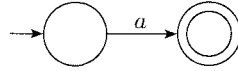
LEMMA 1.55

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

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

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

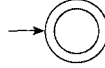
1. $R = a$ for some a in Σ . Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.



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

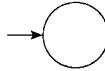
Formally, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.

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



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

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



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

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

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

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

EXAMPLE 1.56

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

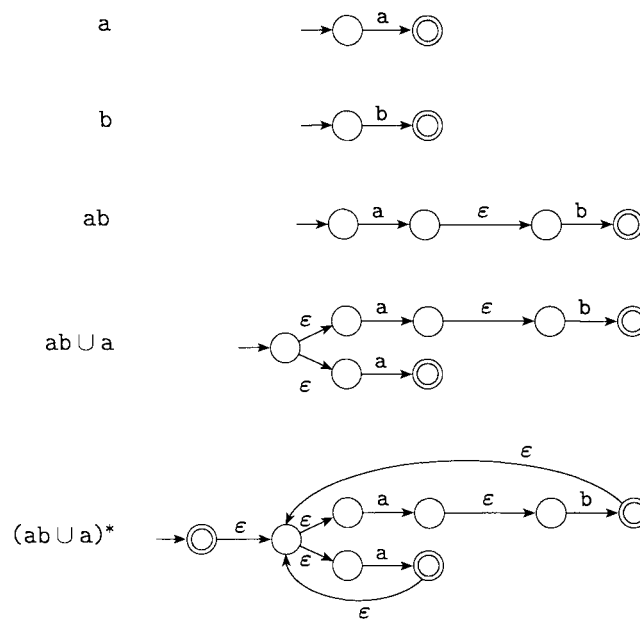
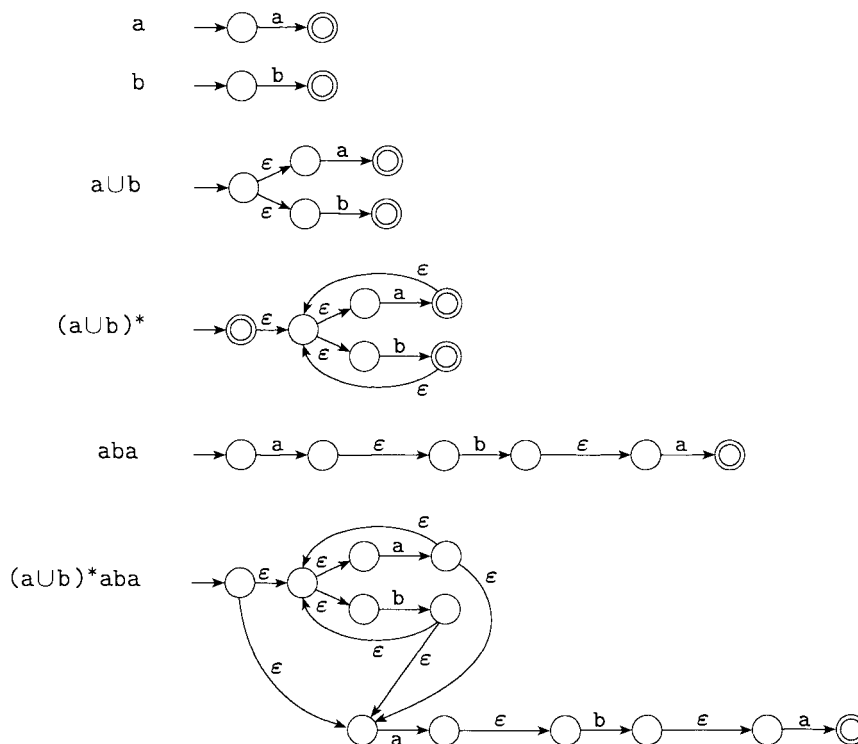


FIGURE 1.57

Building an NFA from the regular expression $(ab \cup a)^*$

EXAMPLE 1.58

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

**FIGURE 1.59**

Building an NFA from the regular expression $(a \cup b)^* aba$

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

LEMMA 1.60

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

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

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

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

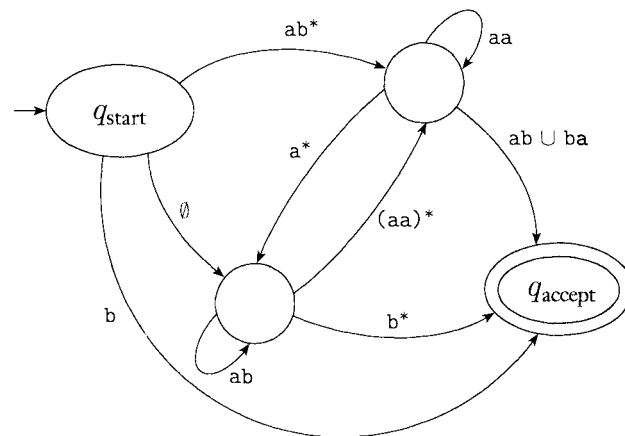


FIGURE 1.61
A generalized nondeterministic finite automaton

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

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

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

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

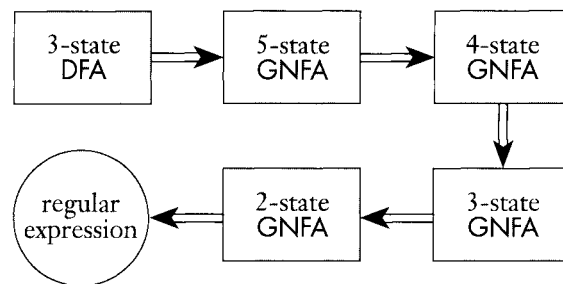


FIGURE 1.62

Typical stages in converting a DFA to a regular expression

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

After removing q_{rip} we repair the machine by altering the regular expressions that label each of the remaining arrows. The new labels compensate for the absence of q_{rip} by adding back the lost computations. The new label going from a state q_i to a state q_j is a regular expression that describes all strings that would

take the machine from q_i to q_j either directly or via q_{rip} . We illustrate this approach in Figure 1.63.

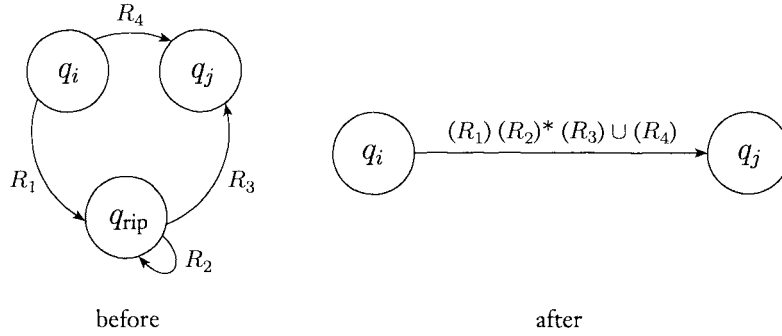


FIGURE 1.63

Constructing an equivalent GNFA with one fewer state

In the old machine if q_i goes to q_{rip} with an arrow labeled R_1 , q_{rip} goes to itself with an arrow labeled R_2 , q_{rip} goes to q_j with an arrow labeled R_3 , and q_i goes to q_j with an arrow labeled R_4 , then in the new machine the arrow from q_i to q_j gets the label

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

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

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

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

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

DEFINITION 1.64

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

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

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

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

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

$\text{CONVERT}(G)$:

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

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

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

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

for $R_1 = \delta(q_i, q_{\text{rip}})$, $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R_3 = \delta(q_{\text{rip}}, q_j)$, and $R_4 = \delta(q_i, q_j)$.

4. Compute $\text{CONVERT}(G')$ and return this value.

Next we prove that CONVERT returns a correct value.

CLAIM 1.65

For any GNFA G , $\text{CONVERT}(G)$ is equivalent to G .

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

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

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

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

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

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

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

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

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

.....

EXAMPLE 1.66

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

In Figure 1.67(b) we make a four-state GNFA by adding a new start state and a new accept state, called s and a instead of q_{start} and q_{accept} so that we can draw them conveniently. To avoid cluttering up the figure, we do not draw the arrows

labeled \emptyset , even though they are present. Note that we replace the label a, b on the self-loop at state 2 on the DFA with the label $a \cup b$ at the corresponding point on the GNFA. We do so because the DFA's label represents two transitions, one for a and the other for b , whereas the GNFA may have only a single transition going from 2 to itself.

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

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

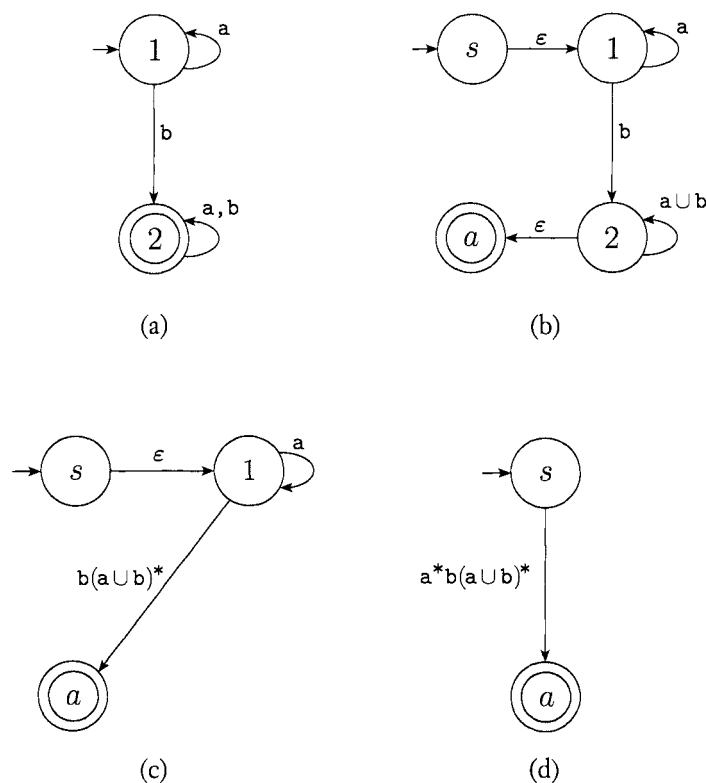


FIGURE 1.67

Converting a two-state DFA to an equivalent regular expression

EXAMPLE 1.68

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

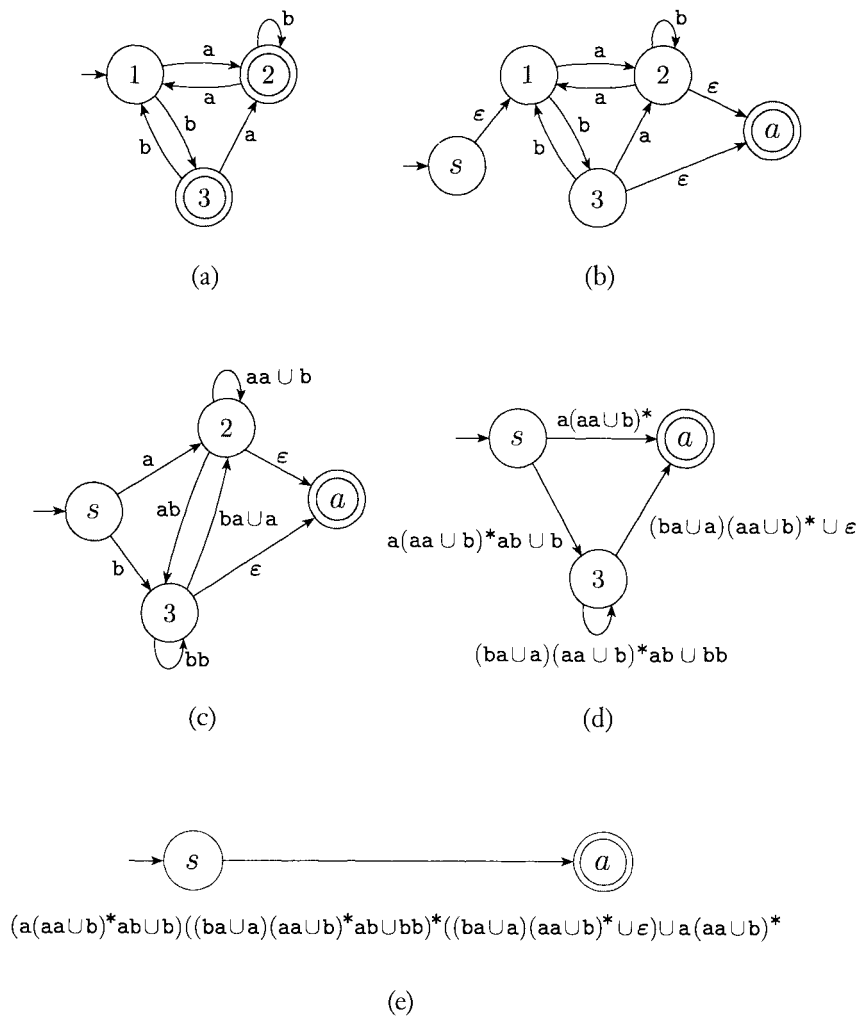


FIGURE 1.69
Converting a three-state DFA to an equivalent regular expression

1.4

NONREGULAR LANGUAGES

To understand the power of finite automata you must also understand their limitations. In this section we show how to prove that certain languages cannot be recognized by any finite automaton.

Let's take the language $B = \{0^n 1^n \mid n \geq 0\}$. If we attempt to find a DFA that recognizes B , we discover that the machine seems to need to remember how many 0s have been seen so far as it reads the input. Because the number of 0s isn't limited, the machine will have to keep track of an unlimited number of possibilities. But it cannot do so with any finite number of states.

Next, we present a method for proving that languages such as B are not regular. Doesn't the argument already given prove nonregularity because the number of 0s is unlimited? It does not. Just because the language appears to require unbounded memory doesn't mean that it is necessarily so. It does happen to be true for the language B , but other languages seem to require an unlimited number of possibilities, yet actually they are regular. For example, consider two languages over the alphabet $\Sigma = \{0,1\}$:

$C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$, and

$D = \{w \mid w \text{ has an equal number of occurrences of 01 and 10 as substrings}\}$.

At first glance a recognizing machine appears to need to count in each case, and therefore neither language appears to be regular. As expected, C is not regular, but surprisingly D is regular!⁶ Thus our intuition can sometimes lead us astray, which is why we need mathematical proofs for certainty. In this section we show how to prove that certain languages are not regular.

THE PUMPING LEMMA FOR REGULAR LANGUAGES

Our technique for proving nonregularity stems from a theorem about regular languages, traditionally called the *pumping lemma*. This theorem states that all regular languages have a special property. If we can show that a language does not have this property, we are guaranteed that it is not regular. The property states that all strings in the language can be "pumped" if they are at least as long as a certain special value, called the *pumping length*. That means each such string contains a section that can be repeated any number of times with the resulting string remaining in the language.

⁶See Problem 1.48.

THEOREM 1.70

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where, if s is any string in A of length at least p , then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Recall the notation where $|s|$ represents the length of string s , y^i means that i copies of y are concatenated together, and y^0 equals ϵ .

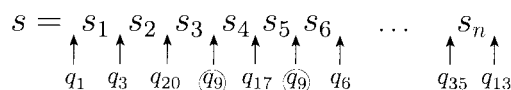
When s is divided into xyz , either x or z may be ϵ , but condition 2 says that $y \neq \epsilon$. Observe that without condition 2 the theorem would be trivially true. Condition 3 states that the pieces x and y together have length at most p . It is an extra technical condition that we occasionally find useful when proving certain languages to be nonregular. See Example 1.74 for an application of condition 3.

PROOF IDEA Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA that recognizes A . We assign the pumping length p to be the number of states of M . We show that any string s in A of length at least p may be broken into the three pieces xyz satisfying our three conditions. What if no strings in A are of length at least p ? Then our task is even easier because the theorem becomes *vacuously* true: Obviously the three conditions hold for all strings of length at least p if there aren't any such strings.

If s in A has length at least p , consider the sequence of states that M goes through when computing with input s . It starts with q_1 the start state, then goes to, say, q_3 , then, say, q_{20} , then q_9 , and so on, until it reaches the end of s in state q_{13} . With s in A , we know that M accepts s , so q_{13} is an accept state.

If we let n be the length of s , the sequence of states $q_1, q_3, q_{20}, q_9, \dots, q_{13}$ has length $n + 1$. Because n is at least p , we know that $n + 1$ is greater than p , the number of states of M . Therefore the sequence must contain a repeated state. This result is an example of the **pigeonhole principle**, a fancy name for the rather obvious fact that if p pigeons are placed into fewer than p holes, some hole has to have more than one pigeon in it.

The following figure shows the string s and the sequence of states that M goes through when processing s . State q_9 is the one that repeats.

**FIGURE 1.71**

Example showing state q_9 repeating when M reads s

We now divide s into the three pieces x , y , and z . Piece x is the part of s

appearing before q_9 , piece y is the part between the two appearances of q_9 , and piece z is the remaining part of s , coming after the second occurrence of q_9 . So x takes M from the state q_1 to q_9 , y takes M from q_9 back to q_9 and z takes M from q_9 to the accept state q_{13} , as shown in the following figure.

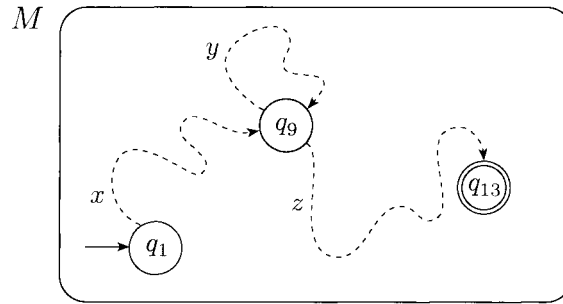


FIGURE 1.72

Example showing how the strings x , y , and z affect M

Let's see why this division of s satisfies the three conditions. Suppose that we run M on input $xyyz$. We know that x takes M from q_1 to q_9 , and then the first y takes it from q_9 back to q_9 , as does the second y , and then z takes it to q_{13} . With q_{13} being an accept state, M accepts input $xyyz$. Similarly, it will accept $xy^i z$ for any $i > 0$. For the case $i = 0$, $xy^i z = xz$, which is accepted for similar reasons. That establishes condition 1.

Checking condition 2, we see that $|y| > 0$, as it was the part of s that occurred between two different occurrences of state q_9 .

In order to get condition 3, we make sure that q_9 is the first repetition in the sequence. By the pigeonhole principle, the first $p + 1$ states in the sequence must contain a repetition. Therefore $|xy| \leq p$.

PROOF Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA recognizing A and p be the number of states of M .

Let $s = s_1 s_2 \cdots s_n$ be a string in A of length n , where $n \geq p$. Let r_1, \dots, r_{n+1} be the sequence of states that M enters while processing s , so $r_{i+1} = \delta(r_i, s_i)$ for $1 \leq i \leq n$. This sequence has length $n + 1$, which is at least $p + 1$. Among the first $p + 1$ elements in the sequence, two must be the same state, by the pigeonhole principle. We call the first of these r_j and the second r_l . Because r_l occurs among the first $p + 1$ places in a sequence starting at r_1 , we have $l \leq p + 1$. Now let $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{l-1}$, and $z = s_l \cdots s_n$.

As x takes M from r_1 to r_j , y takes M from r_j to r_j , and z takes M from r_j to r_{n+1} , which is an accept state, M must accept $xy^i z$ for $i \geq 0$. We know that $j \neq l$, so $|y| > 0$; and $l \leq p + 1$, so $|xy| \leq p$. Thus we have satisfied all conditions of the pumping lemma.

To use the pumping lemma to prove that a language B is not regular, first assume that B is regular in order to obtain a contradiction. Then use the pumping lemma to guarantee the existence of a pumping length p such that all strings of length p or greater in B can be pumped. Next, find a string s in B that has length p or greater but that cannot be pumped. Finally, demonstrate that s cannot be pumped by considering all ways of dividing s into x , y , and z (taking condition 3 of the pumping lemma into account if convenient) and, for each such division, finding a value i where $xy^iz \notin B$. This final step often involves grouping the various ways of dividing s into several cases and analyzing them individually. The existence of s contradicts the pumping lemma if B were regular. Hence B cannot be regular.

Finding s sometimes takes a bit of creative thinking. You may need to hunt through several candidates for s before you discover one that works. Try members of B that seem to exhibit the “essence” of B ’s nonregularity. We further discuss the task of finding s in some of the following examples.

EXAMPLE 1.73

Let B be the language $\{0^n 1^n \mid n \geq 0\}$. We use the pumping lemma to prove that B is not regular. The proof is by contradiction.

Assume to the contrary that B is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string $0^p 1^p$. Because s is a member of B and s has length more than p , the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in B . We consider three cases to show that this result is impossible.

1. The string y consists only of 0s. In this case the string $xyyz$ has more 0s than 1s and so is not a member of B , violating condition 1 of the pumping lemma. This case is a contradiction.
2. The string y consists only of 1s. This case also gives a contradiction.
3. The string y consists of both 0s and 1s. In this case the string $xyyz$ may have the same number of 0s and 1s, but they will be out of order with some 1s before 0s. Hence it is not a member of B , which is a contradiction.

Thus a contradiction is unavoidable if we make the assumption that B is regular, so B is not regular. Note that we can simplify this argument by applying condition 3 of the pumping lemma to eliminate cases 2 and 3.

In this example, finding the string s was easy, because any string in B of length p or more would work. In the next two examples some choices for s do not work, so additional care is required. □

EXAMPLE 1.74

Let $C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$. We use the pumping lemma to prove that C is not regular. The proof is by contradiction.

Assume to the contrary that C is regular. Let p be the pumping length given by the pumping lemma. As in Example 1.73, let s be the string $0^p 1^p$. With s being a member of C and having length more than p , the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^i z$ is in C . We would like to show that this outcome is impossible. But wait, it *is* possible! If we let x and z be the empty string and y be the string $0^p 1^p$, then $xy^i z$ always has an equal number of 0s and 1s and hence is in C . So it *seems* that s can be pumped.

Here condition 3 in the pumping lemma is useful. It stipulates that when pumping s it must be divided so that $|xy| \leq p$. That restriction on the way that s may be divided makes it easier to show that the string $s = 0^p 1^p$ we selected cannot be pumped. If $|xy| \leq p$, then y must consist only of 0s, so $xyyz \notin C$. Therefore s cannot be pumped. That gives us the desired contradiction.⁷

Selecting the string s in this example required more care than in Example 1.73. If we had chosen $s = (01)^p$ instead, we would have run into trouble because we need a string that *cannot* be pumped and that string *can* be pumped, even taking condition 3 into account. Can you see how to pump it? One way to do so sets $x = \varepsilon$, $y = 01$, and $z = (01)^{p-1}$. Then $xy^i z \in C$ for every value of i . If you fail on your first attempt to find a string that cannot be pumped, don't despair. Try another one!

An alternative method of proving that C is nonregular follows from our knowledge that B is nonregular. If C were regular, $C \cap 0^* 1^*$ also would be regular. The reasons are that the language $0^* 1^*$ is regular and that the class of regular languages is closed under intersection, which we proved in footnote 3 (page 46). But $C \cap 0^* 1^*$ equals B , and we know that B is nonregular from Example 1.73.

EXAMPLE 1.75

Let $F = \{ww \mid w \in \{0,1\}^*\}$. We show that F is nonregular, using the pumping lemma.

Assume to the contrary that F is regular. Let p be the pumping length given by the pumping lemma. Let s be the string $0^p 10^p 1$. Because s is a member of F and s has length more than p , the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, satisfying the three conditions of the lemma. We show that this outcome is impossible.

Condition 3 is once again crucial, because without it we could pump s if we let x and z be the empty string. With condition 3 the proof follows because y must consist only of 0s, so $xyyz \notin F$.

Observe that we chose $s = 0^p 10^p 1$ to be a string that exhibits the “essence” of the nonregularity of F , as opposed to, say, the string $0^p 0^p$. Even though $0^p 0^p$ is a member of F , it fails to demonstrate a contradiction because it can be pumped.

⁷We could have used condition 3 in Example 1.73, as well, to simplify its proof.

EXAMPLE 1.76

Here we demonstrate a nonregular unary language. Let $D = \{1^{n^2} \mid n \geq 0\}$. In other words, D contains all strings of 1s whose length is a perfect square. We use the pumping lemma to prove that D is not regular. The proof is by contradiction.

Assume to the contrary that D is regular. Let p be the pumping length given by the pumping lemma. Let s be the string 1^{p^2} . Because s is a member of D and s has length at least p , the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in D . As in the preceding examples, we show that this outcome is impossible. Doing so in this case requires a little thought about the sequence of perfect squares:

$$0, 1, 4, 9, 16, 25, 36, 49, \dots$$

Note the growing gap between successive members of this sequence. Large members of this sequence cannot be near each other.

Now consider the two strings xyz and xy^2z . These strings differ from each other by a single repetition of y , and consequently their lengths differ by the length of y . By condition 3 of the pumping lemma, $|xy| \leq p$ and thus $|y| \leq p$. We have $|xyz| = p^2$ and so $|xy^2z| \leq p^2 + p$. But $p^2 + p < p^2 + 2p + 1 = (p+1)^2$. Moreover, condition 2 implies that y is not the empty string and so $|xy^2z| > p^2$. Therefore the length of xy^2z lies strictly between the consecutive perfect squares p^2 and $(p+1)^2$. Hence this length cannot be a perfect square itself. So we arrive at the contradiction $xy^2z \notin D$ and conclude that D is not regular. \square

EXAMPLE 1.77

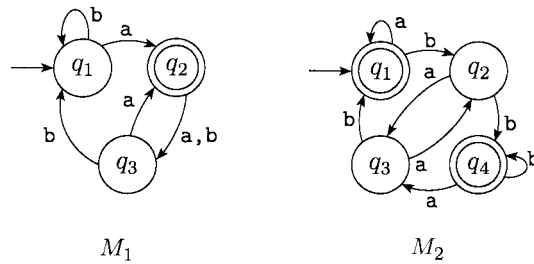
Sometimes “pumping down” is useful when we apply the pumping lemma. We use the pumping lemma to show that $E = \{0^i1^j \mid i > j\}$ is not regular. The proof is by contradiction.

Assume that E is regular. Let p be the pumping length for E given by the pumping lemma. Let $s = 0^{p+1}1^p$. Then s can be split into xyz , satisfying the conditions of the pumping lemma. By condition 3, y consists only of 0s. Let’s examine the string $xyyz$ to see whether it can be in E . Adding an extra copy of y increases the number of 0s. But, E contains all strings in 0^*1^* that have more 0s than 1s, so increasing the number of 0s will still give a string in E . No contradiction occurs. We need to try something else.

The pumping lemma states that $xy^iz \in E$ even when $i = 0$, so let’s consider the string $xy^0z = xz$. Removing string y decreases the number of 0s in s . Recall that s has just one more 0 than 1. Therefore xz cannot have more 0s than 1s, so it cannot be a member of E . Thus we obtain a contradiction. \square

EXERCISES

- ^A1.1 The following are the state diagrams of two DFAs, M_1 and M_2 . Answer the following questions about each of these machines.



- What is the start state?
 - What is the set of accept states?
 - What sequence of states does the machine go through on input **aabb**?
 - Does the machine accept the string **aabb**?
 - Does the machine accept the string ϵ ?
- ^A1.2 Give the formal description of the machines M_1 and M_2 pictured in Exercise 1.1.
- 1.3 The formal description of a DFA M is $(\{q_1, q_2, q_3, q_4, q_5\}, \{u, d\}, \delta, q_3, \{q_3\})$, where δ is given by the following table. Give the state diagram of this machine.

	u	d
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_2	q_4
q_4	q_3	q_5
q_5	q_4	q_5

- 1.4 Each of the following languages is the intersection of two simpler languages. In each part, construct DFAs for the simpler languages, then combine them using the construction discussed in footnote 3 (page 46) to give the state diagram of a DFA for the language given. In all parts $\Sigma = \{a, b\}$.
- $\{w \mid w \text{ has at least three } a\text{'s and at least two } b\text{'s}\}$
 - ^A $\{w \mid w \text{ has at exactly two } a\text{'s and at least two } b\text{'s}\}$
 - $\{w \mid w \text{ has an even number of } a\text{'s and one or two } b\text{'s}\}$
 - ^A $\{w \mid w \text{ has an even number of } a\text{'s and each } a \text{ is followed by at least one } b\}$
 - $\{w \mid w \text{ starts with an } a \text{ and has at most one } b\}$
 - $\{w \mid w \text{ has an odd number of } a\text{'s and ends with a } b\}$
 - $\{w \mid w \text{ has even length and an odd number of } a\text{'s}\}$

- 1.5 Each of the following languages is the complement of a simpler language. In each part, construct a DFA for the simpler language, then use it to give the state diagram of a DFA for the language given. In all parts $\Sigma = \{a, b\}$.
- $\{w \mid w \text{ does not contain the substring } ab\}$
 - $\{w \mid w \text{ does not contain the substring } baba\}$
 - $\{w \mid w \text{ contains neither the substrings } ab \text{ nor } ba\}$
 - $\{w \mid w \text{ is any string not in } a^*b^*\}$
 - $\{w \mid w \text{ is any string not in } (ab^+)^*\}$
 - $\{w \mid w \text{ is any string not in } a^* \cup b^*\}$
 - $\{w \mid w \text{ is any string that doesn't contain exactly two } a\text{'s}\}$
 - $\{w \mid w \text{ is any string except } a \text{ and } b\}$
- 1.6 Give state diagrams of DFAs recognizing the following languages. In all parts the alphabet is $\{0,1\}$
- $\{w \mid w \text{ begins with a } 1 \text{ and ends with a } 0\}$
 - $\{w \mid w \text{ contains at least three } 1\text{'s}\}$
 - $\{w \mid w \text{ contains the substring } 0101, \text{ i.e., } w = x0101y \text{ for some } x \text{ and } y\}$
 - $\{w \mid w \text{ has length at least } 3 \text{ and its third symbol is a } 0\}$
 - $\{w \mid w \text{ starts with } 0 \text{ and has odd length, or starts with } 1 \text{ and has even length}\}$
 - $\{w \mid w \text{ doesn't contain the substring } 110\}$
 - $\{w \mid \text{the length of } w \text{ is at most } 5\}$
 - $\{w \mid w \text{ is any string except } 11 \text{ and } 111\}$
 - $\{w \mid \text{every odd position of } w \text{ is a } 1\}$
 - $\{w \mid w \text{ contains at least two } 0\text{'s and at most one } 1\}$
 - $\{\epsilon, 0\}$
 - $\{w \mid w \text{ contains an even number of } 0\text{'s, or contains exactly two } 1\text{'s}\}$
 - The empty set
 - All strings except the empty string
- 1.7 Give state diagrams of NFAs with the specified number of states recognizing each of the following languages. In all parts the alphabet is $\{0,1\}$.
- The language $\{w \mid w \text{ ends with } 00\}$ with three states
 - The language of Exercise 1.6c with five states
 - The language of Exercise 1.6l with six states
 - The language $\{0\}$ with two states
 - The language $0^*1^*0^*$ with three states
 - The language $1^*(001^+)^*$ with three states
 - The language $\{\epsilon\}$ with one state
 - The language 0^* with one state
- 1.8 Use the construction given in the proof of Theorem 1.45 to give the state diagrams of NFAs recognizing the union of the languages described in
- Exercises 1.6a and 1.6b.
 - Exercises 1.6c and 1.6f.

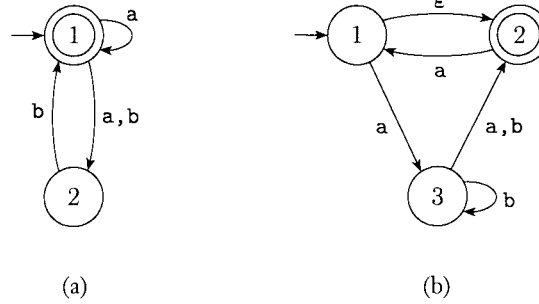
- 1.9 Use the construction given in the proof of Theorem 1.47 to give the state diagrams of NFAs recognizing the concatenation of the languages described in
- Exercises 1.6g and 1.6i.
 - Exercises 1.6b and 1.6m.
- 1.10 Use the construction given in the proof of Theorem 1.49 to give the state diagrams of NFAs recognizing the star of the language described in
- Exercise 1.6b.
 - Exercise 1.6j.
 - Exercise 1.6m.
- ^A1.11 Prove that every NFA can be converted to an equivalent one that has a single accept state.
- 1.12 Let $D = \{w \mid w \text{ contains an even number of a's and an odd number of b's and does not contain the substring } ab\}$. Give a DFA with five states that recognizes D and a regular expression that generates D . (Suggestion: Describe D more simply.)
- 1.13 Let F be the language of all strings over $\{0,1\}$ that do not contain a pair of 1s that are separated by an odd number of symbols. Give the state diagram of a DFA with 5 states that recognizes F . (You may find it helpful first to find a 4-state NFA for the complement of F .)
- 1.14
- Show that, if M is a DFA that recognizes language B , swapping the accept and nonaccept states in M yields a new DFA that recognizes the complement of B . Conclude that the class of regular languages is closed under complement.
 - Show by giving an example that, if M is an NFA that recognizes language C , swapping the accept and nonaccept states in M doesn't necessarily yield a new NFA that recognizes the complement of C . Is the class of languages recognized by NFAs closed under complement? Explain your answer.
- 1.15 Give a counterexample to show that the following construction fails to prove Theorem 1.49, the closure of the class of regular languages under the star operation.⁸ Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1 . Construct $N = (Q_1, \Sigma, \delta, q_1, F)$ as follows. N is supposed to recognize A_1^* .
- The states of N are the states of N_1 .
 - The start state of N is the same as the start state of N_1 .
 - $F = \{q_1\} \cup F_1$.
The accept states F are the old accept states plus its start state.
 - Define δ so that for any $q \in Q$ and any $a \in \Sigma_\epsilon$,

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

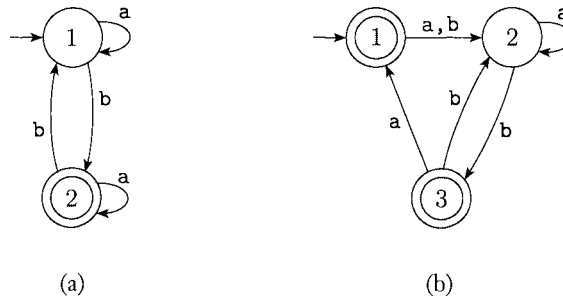
(Suggestion: Show this construction graphically, as in Figure 1.50.)

⁸In other words, you must present a finite automaton, N_1 , for which the constructed automaton N does not recognize the star of N_1 's language.

- 1.16 Use the construction given in Theorem 1.39 to convert the following two nondeterministic finite automata to equivalent deterministic finite automata.



- 1.17 a. Give an NFA recognizing the language $(01 \cup 001 \cup 010)^*$.
 b. Convert this NFA to an equivalent DFA. Give only the portion of the DFA that is reachable from the start state.
- 1.18 Give regular expressions generating the languages of Exercise 1.6.
- 1.19 Use the procedure described in Lemma 1.55 to convert the following regular expressions to nondeterministic finite automata.
- $(0 \cup 1)^*000(0 \cup 1)^*$
 - $((00)^*(11) \cup 01)^*$
 - \emptyset^*
- 1.20 For each of the following languages, give two strings that are members and two strings that are *not* members—a total of four strings for each part. Assume the alphabet $\Sigma = \{a, b\}$ in all parts.
- a^*b^*
 - $a(ba)^*b$
 - $a^* \cup b^*$
 - $(aaa)^*$
 - $\Sigma^*a\Sigma^*b\Sigma^*a\Sigma^*$
 - $aba \cup bab$
 - $(\epsilon \cup a)b$
 - $(a \cup ba \cup bb)\Sigma^*$
- 1.21 Use the procedure described in Lemma 1.60 to convert the following finite automata to regular expressions.

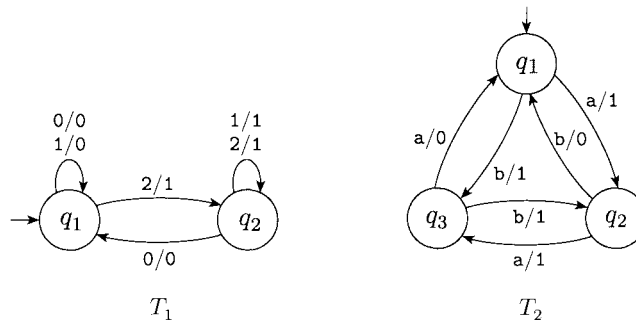


- 1.22 In certain programming languages, comments appear between delimiters such as `/#` and `#/`. Let C be the language of all valid delimited comment strings. A member of C must begin with `/#` and end with `#/` but have no intervening `#/`. For simplicity, we'll say that the comments themselves are written with only the symbols `a` and `b`; hence the alphabet of C is $\Sigma = \{a, b, /, \#\}$.

- Give a DFA that recognizes C .
- Give a regular expression that generates C .

^A1.23 Let B be any language over the alphabet Σ . Prove that $B = B^*$ iff $BB \subseteq B$.

- 1.24 A **finite state transducer** (FST) is a type of deterministic finite automaton whose output is a string and not just *accept* or *reject*. The following are state diagrams of finite state transducers T_1 and T_2 .



Each transition of an FST is labeled with two symbols, one designating the input symbol for that transition and the other designating the output symbol. The two symbols are written with a slash, $/$, separating them. In T_1 , the transition from q_1 to q_2 has input symbol 2 and output symbol 1. Some transitions may have multiple input–output pairs, such as the transition in T_1 from q_1 to itself. When an FST computes on an input string w , it takes the input symbols $w_1 \cdots w_n$ one by one and, starting at the start state, follows the transitions by matching the input labels with the sequence of symbols $w_1 \cdots w_n = w$. Every time it goes along a transition, it outputs the corresponding output symbol. For example, on input 2212011, machine T_1 enters the sequence of states $q_1, q_2, q_2, q_2, q_2, q_1, q_1, q_1$ and produces output 1111000. On input `abbb`, T_2 outputs 1011. Give the sequence of states entered and the output produced in each of the following parts.

- T_1 on input 011
 - T_1 on input 211
 - T_1 on input 121
 - T_1 on input 0202
 - T_2 on input `b`
 - T_2 on input `bbab`
 - T_2 on input `bbbbbb`
 - T_2 on input ϵ
- 1.25 Read the informal definition of the finite state transducer given in Exercise 1.24. Give a formal definition of this model, following the pattern in Definition 1.5 (page 35). Assume that an FST has an input alphabet Σ and an output alphabet Γ but not a set of accept states. Include a formal definition of the computation of an FST. (Hint: An FST is a 5-tuple. Its transition function is of the form $\delta: Q \times \Sigma \rightarrow Q \times \Gamma$.)

- $a(ab)^* \cup b$
- $a^+ \cup (ab)^+$
- $(a \cup b^+)a^+b^+$

[illegible]

- 1.32** Let

Σ_3 contains all size 3 columns of 0s and 1s. A string of symbols in Σ_3 gives three rows of 0s and 1s. Consider each row to be a binary number and let

For example,

Show that B is regular. (Hint: Working with $B^{\mathcal{R}}$ is easier. You may assume the result claimed in Problem 1.31.)

1.33 Let

$$\Sigma_2 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}.$$

Here, Σ_2 contains all columns of 0s and 1s of height two. A string of symbols in Σ_2 gives two rows of 0s and 1s. Consider each row to be a binary number and let

$$C = \{w \in \Sigma_2^* \mid \text{the bottom row of } w \text{ is three times the top row}\}.$$

For example, $\begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \in C$, but $\begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \notin C$. Show that C is regular. (You may assume the result claimed in Problem 1.31.)

1.34 Let Σ_2 be the same as in Problem 1.33. Consider each row to be a binary number and let

$$D = \{w \in \Sigma_2^* \mid \text{the top row of } w \text{ is a larger number than is the bottom row}\}.$$

For example, $\begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \in D$, but $\begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \notin D$. Show that D is regular.

1.35 Let Σ_2 be the same as in Problem 1.33. Consider the top and bottom rows to be strings of 0s and 1s and let

$$E = \{w \in \Sigma_2^* \mid \text{the bottom row of } w \text{ is the reverse of the top row of } w\}.$$

Show that E is not regular.

1.36 Let $B_n = \{a^k \mid k \text{ is a multiple of } n\}$. Show that for each $n \geq 1$, the language B_n is regular.

1.37 Let $C_n = \{x \mid x \text{ is a binary number that is a multiple of } n\}$. Show that for each $n \geq 1$, the language C_n is regular.

1.38 An *all-NFA* M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ that accepts $x \in \Sigma^*$ if *every* possible state that M could be in after reading input x is a state from F . Note, in contrast, that an ordinary NFA accepts a string if *some* state among these possible states is an accept state. Prove that all-NFAs recognize the class of regular languages.

1.39 The construction in Theorem 1.54 shows that every GNFA is equivalent to a GNFA with only two states. We can show that an opposite phenomenon occurs for DFAs. Prove that for every $k > 1$ a language $A_k \subseteq \{0,1\}^*$ exists that is recognized by a DFA with k states but not by one with only $k - 1$ states.

1.40 Say that string x is a *prefix* of string y if a string z exists where $xz = y$ and that x is a *proper prefix* of y if in addition $x \neq y$. In each of the following parts we define an operation on a language A . Show that the class of regular languages is closed under that operation.

a. $\text{NOPREFIX}(A) = \{w \in A \mid \text{no proper prefix of } w \text{ is a member of } A\}$.

b. $\text{NOEXTEND}(A) = \{w \in A \mid w \text{ is not the proper prefix of any string in } A\}$.

1.41 For languages A and B , let the *perfect shuffle* of A and B be the language

$$\{w \mid w = a_1 b_1 \cdots a_k b_k, \text{ where } a_1 \cdots a_k \in A \text{ and } b_1 \cdots b_k \in B, \text{ each } a_i, b_i \in \Sigma\}.$$

Show that the class of regular languages is closed under perfect shuffle.

1.42 For languages A and B , let the *shuffle* of A and B be the language

$$\{w \mid w = a_1 b_1 \cdots a_k b_k, \text{ where } a_1 \cdots a_k \in A \text{ and } b_1 \cdots b_k \in B, \text{ each } a_i, b_i \in \Sigma^*\}.$$

Show that the class of regular languages is closed under shuffle.

1.43 Let A be any language. Define $DROP-OUT(A)$ to be the language containing all strings that can be obtained by removing one symbol from a string in A . Thus, $DROP-OUT(A) = \{xz \mid xyz \in A \text{ where } x, z \in \Sigma^*, y \in \Sigma\}$. Show that the class of regular languages is closed under the DROP-OUT operation. Give both a proof by picture and a more formal proof by construction as in Theorem 1.47.

^A**1.44** Let B and C be languages over $\Sigma = \{0, 1\}$. Define

$$B \stackrel{1}{\leftarrow} C = \{w \in B \mid \text{for some } y \in C, \text{ strings } w \text{ and } y \text{ contain equal numbers of 1s}\}.$$

Show that the class of regular languages is closed under the $\stackrel{1}{\leftarrow}$ operation.

^{*}**1.45** Let $A/B = \{w \mid wx \in A \text{ for some } x \in B\}$. Show that if A is regular and B is any language then A/B is regular.

1.46 Prove that the following languages are not regular. You may use the pumping lemma and the closure of the class of regular languages under union, intersection, and complement.

- a. $\{0^n 1^m 0^n \mid m, n \geq 0\}$
- ^Ab. $\{0^m 1^n \mid m \neq n\}$
- c. $\{w \mid w \in \{0, 1\}^* \text{ is not a palindrome}\}$ ⁹
- d. $\{wtw \mid w, t \in \{0, 1\}^+\}$

1.47 Let $\Sigma = \{1, \#\}$ and let

$$Y = \{w \mid w = x_1 \# x_2 \# \cdots \# x_k \text{ for } k \geq 0, \text{ each } x_i \in 1^*, \text{ and } x_i \neq x_j \text{ for } i \neq j\}.$$

Prove that Y is not regular.

1.48 Let $\Sigma = \{0, 1\}$ and let

$$D = \{w \mid w \text{ contains an equal number of occurrences of the substrings } 01 \text{ and } 10\}.$$

Thus $101 \in D$ because 101 contains a single 01 and a single 10 , but $1010 \notin D$ because 1010 contains two 10 s and one 01 . Show that D is a regular language.

- 1.49**
- a. Let $B = \{1^k y \mid y \in \{0, 1\}^* \text{ and } y \text{ contains at least } k \text{ 1s, for } k \geq 1\}$. Show that B is a regular language.
 - b. Let $C = \{1^k y \mid y \in \{0, 1\}^* \text{ and } y \text{ contains at most } k \text{ 1s, for } k \geq 1\}$. Show that C isn't a regular language.

^A**1.50** Read the informal definition of the finite state transducer given in Exercise 1.24. Prove that no FST can output w^R for every input w if the input and output alphabets are $\{0, 1\}$.

1.51 Let x and y be strings and let L be any language. We say that x and y are **indistinguishable by L** if some string z exists whereby exactly one of the strings xz and yz is a member of L ; otherwise, for every string z , we have $xz \in L$ whenever $yz \in L$ and we say that x and y are **indistinguishable by L** . If x and y are indistinguishable by L we write $x \equiv_L y$. Show that \equiv_L is an equivalence relation.

⁹A **palindrome** is a string that reads the same forward and backward.

^{A*}1.52 **Myhill–Nerode theorem.** Refer to Problem 1.51. Let L be a language and let X be a set of strings. Say that X is *pairwise distinguishable by L* if every two distinct strings in X are distinguishable by L . Define the *index of L* to be the maximum number of elements in any set that is pairwise distinguishable by L . The index of L may be finite or infinite.

- Show that, if L is recognized by a DFA with k states, L has index at most k .
- Show that, if the index of L is a finite number k , it is recognized by a DFA with k states.
- Conclude that L is regular iff it has finite index. Moreover, its index is the size of the smallest DFA recognizing it.

1.53 Let $\Sigma = \{0, 1, +, =\}$ and

$$ADD = \{x=y+z \mid x, y, z \text{ are binary integers, and } x \text{ is the sum of } y \text{ and } z\}.$$

Show that ADD is not regular.

1.54 Consider the language $F = \{a^i b^j c^k \mid i, j, k \geq 0 \text{ and if } i = 1 \text{ then } j = k\}$.

- Show that F is not regular.
- Show that F acts like a regular language in the pumping lemma. In other words, give a pumping length p and demonstrate that F satisfies the three conditions of the pumping lemma for this value of p .
- Explain why parts (a) and (b) do not contradict the pumping lemma.

1.55 The pumping lemma says that every regular language has a pumping length p , such that every string in the language can be pumped if it has length p or more. If p is a pumping length for language A , so is any length $p' \geq p$. The *minimum pumping length* for A is the smallest p that is a pumping length for A . For example, if $A = 01^*$, the minimum pumping length is 2. The reason is that the string $s = 0$ is in A and has length 1 yet s cannot be pumped, but any string in A of length 2 or more contains a 1 and hence can be pumped by dividing it so that $x = 0$, $y = 1$, and z is the rest. For each of the following languages, give the minimum pumping length and justify your answer.

- | | |
|---|-------------------|
| ^A a. 0001^* | f. ε |
| ^A b. 0^*1^* | g. $1^*01^*01^*$ |
| c. $001 \cup 0^*1^*$ | h. $10(11^*0)^*0$ |
| ^A d. $0^*1^*0^*1^* \cup 10^*1$ | i. 1011 |
| e. $(01)^*$ | j. Σ^* |

*1.56 If A is a set of natural numbers and k is a natural number greater than 1, let

$$B_k(A) = \{w \mid w \text{ is the representation in base } k \text{ of some number in } A\}.$$

Here, we do not allow leading 0s in the representation of a number. For example, $B_2(\{3, 5\}) = \{11, 101\}$ and $B_3(\{3, 5\}) = \{10, 12\}$. Give an example of a set A for which $B_2(A)$ is regular but $B_3(A)$ is not regular. Prove that your example works.

- *1.57 If A is any language, let $A_{\frac{1}{2}-}$ be the set of all first halves of strings in A so that

$$A_{\frac{1}{2}-} = \{x \mid \text{for some } y, |x| = |y| \text{ and } xy \in A\}.$$

Show that, if A is regular, then so is $A_{\frac{1}{2}-}$.

- *1.58 If A is any language, let $A_{\frac{1}{3}-\frac{1}{3}}$ be the set of all strings in A with their middle thirds removed so that

$$A_{\frac{1}{3}-\frac{1}{3}} = \{xz \mid \text{for some } y, |x| = |y| = |z| \text{ and } xyz \in A\}.$$

Show that, if A is regular, then $A_{\frac{1}{3}-\frac{1}{3}}$ is not necessarily regular.

- *1.59 Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let h be a state of M called its “home”. A **synchronizing sequence** for M and h is a string $s \in \Sigma^*$ where $\delta(q, s) = h$ for every $q \in Q$. (Here we have extended δ to strings, so that $\delta(q, s)$ equals the state where M ends up when M starts at state q and reads input s .) Say that M is **synchronizable** if it has a synchronizing sequence for some state h . Prove that, if M is a k -state synchronizable DFA, then it has a synchronizing sequence of length at most k^3 . Can you improve upon this bound?
- 1.60 Let $\Sigma = \{a, b\}$. For each $k \geq 1$, let C_k be the language consisting of all strings that contain an a exactly k places from the right-hand end. Thus $C_k = \Sigma^* a \Sigma^{k-1}$. Describe an NFA with $k + 1$ states that recognizes C_k , both in terms of a state diagram and a formal description.
- 1.61 Consider the languages C_k defined in Problem 1.60. Prove that for each k , no DFA can recognize C_k with fewer than 2^k states.
- 1.62 Let $\Sigma = \{a, b\}$. For each $k \geq 1$, let D_k be the language consisting of all strings that have at least one a among the last k symbols. Thus $D_k = \Sigma^* a (\Sigma \cup \epsilon)^{k-1}$. Describe a DFA with at most $k + 1$ states that recognizes D_k , both in terms of a state diagram and a formal description.
- 1.63
- Let A be an infinite regular language. Prove that A can be split into two infinite disjoint regular subsets.
 - Let B and D be two languages. Write $B \Subset D$ if $B \subseteq D$ and D contains infinitely many strings that are not in B . Show that, if B and D are two regular languages where $B \Subset D$, then we can find a regular language C where $B \Subset C \Subset D$.
- 1.64 Let N be an NFA with k states that recognizes some language A .
- Show that, if A is nonempty, A contains some string of length at most k .
 - Show that, by giving an example, that part (a) is not necessarily true if you replace both A 's by \bar{A} .
 - Show that, if \bar{A} is nonempty, \bar{A} contains some string of length at most 2^k .
 - Show that the bound given in part (c) is nearly tight; that is, for each k , demonstrate an NFA recognizing a language A_k where \bar{A}_k is nonempty and where \bar{A}_k 's shortest member strings are of length exponential in k . Come as close to the bound in (c) as you can.

*1.65 Prove that, for each $n > 0$, a language B_n exists where

- a. B_n is recognizable by a NFA that has n states, and
- b. if $B_n = A_1 \cup \dots \cup A_k$, for regular languages A_i , then at least one of the A_i requires a DFA with exponentially many states.

SELECTED SOLUTIONS

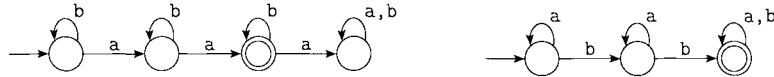
1.1 For M_1 : (a) q_1 ; (b) $\{q_2\}$; (c) q_1, q_2, q_3, q_1, q_1 ; (d) No; (e) No
 For M_2 : (a) q_1 ; (b) $\{q_1, q_4\}$; (c) q_1, q_1, q_1, q_2, q_4 ; (d) Yes; (e) Yes

1.2 $M_2 = (\{q_1, q_2, q_3\}, \{a, b\}, \delta_1, q_1, \{q_2\})$.
 $M_3 = (\{q_1, q_2, q_3, q_4\}, \{a, b\}, \delta_2, q_1, \{q_1, q_4\})$.
 The transition functions are

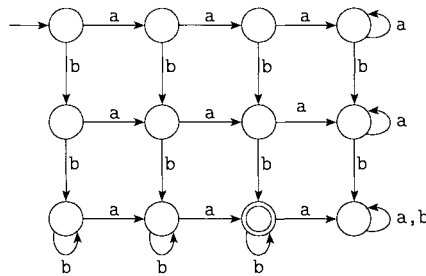
δ_1	a	b
q_1	q_2	q_1
q_2	q_3	q_3
q_3	q_2	q_1

δ_2	a	b
q_1	q_1	q_2
q_2	q_3	q_4
q_3	q_2	q_1
q_4	q_3	q_4

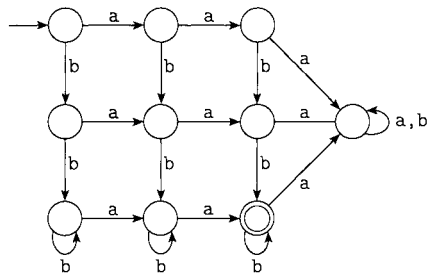
1.4 (b) The following are DFAs for the two languages $\{w \mid w \text{ has exactly two a's}\}$ and $\{w \mid w \text{ has at least two b's}\}$:



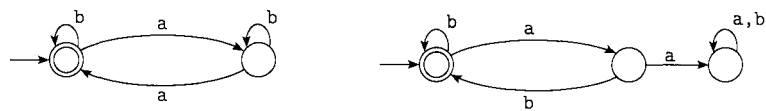
Combining them using the intersection construction gives the DFA:



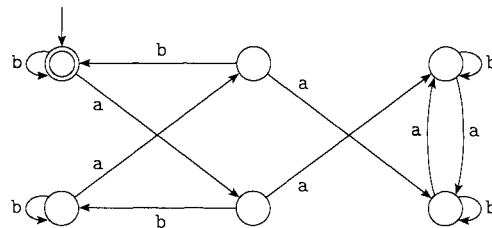
Though the problem doesn't request you to simplify the DFA, certain states can be combined to give



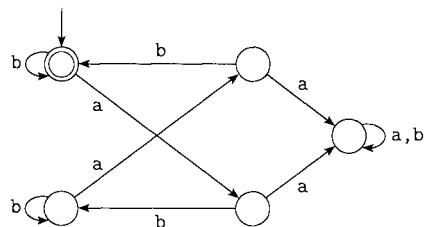
(d) These are DFAs for the two languages $\{w \mid w \text{ has an even number of } a\text{'s}\}$ and $\{w \mid \text{each } a \text{ is followed by at least one } b\}$:



Combining them using the intersection construction gives the DFA:



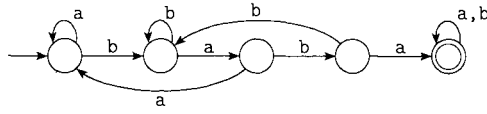
Though the problem doesn't request you to simplify the DFA, certain states can be combined to give



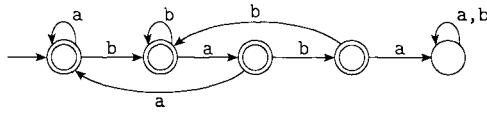
- 1.5 (a) The left-hand DFA recognizes $\{w \mid w \text{ contains } ab\}$. The right-hand DFA recognizes its complement, $\{w \mid w \text{ doesn't contain } ab\}$.



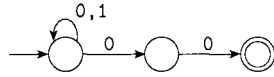
- (b) This DFA recognizes $\{w \mid w \text{ contains } baba\}$.



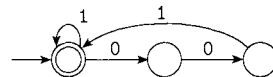
This DFA recognizes $\{w \mid w \text{ does not contain } baba\}$.



- 1.7 (a)



- (f)



- 1.11 Let $N = (Q, \Sigma, \delta, q_0, F)$ be any NFA. Construct an NFA N' with a single accept state that accepts the same language as N . Informally, N' is exactly like N except it has ε -transitions from the states corresponding to the accept states of N , to a new accept state, q_{accept} . State q_{accept} has no emerging transitions. More formally, $N' = (Q \cup \{q_{\text{accept}}\}, \Sigma, \delta', q_0, \{q_{\text{accept}}\})$, where for each $q \in Q$ and $a \in \Sigma$

$$\delta'(q, a) = \begin{cases} \delta(q, a) & \text{if } a \neq \varepsilon \text{ or } q \notin F \\ \delta(q, a) \cup \{q_{\text{accept}}\} & \text{if } a = \varepsilon \text{ and } q \in F \end{cases}$$

and $\delta'(q_{\text{accept}}, a) = \emptyset$ for each $a \in \Sigma_\varepsilon$.

- 1.23 We prove both directions of the “iff.”

(\rightarrow) Assume that $B = B^*$ and show that $BB \subseteq B$.

For every language $BB \subseteq B^*$ holds, so if $B = B^*$, then $BB \subseteq B$.

(\leftarrow) Assume that $BB \subseteq B$ and show that $B = B^*$.

For every language $B \subseteq B^*$, so we need to show only $B^* \subseteq B$. If $w \in B^*$, then $w = x_1x_2 \cdots x_k$ where each $x_i \in B$ and $k \geq 1$. Because $x_1, x_2 \in B$ and $BB \subseteq B$, we have $x_1x_2 \in B$. Similarly, because x_1x_2 is in B and x_3 is in B , we have $x_1x_2x_3 \in B$. Continuing in this way, $x_1 \cdots x_k \in B$. Hence $w \in B$, and so we may conclude that $B^* \subseteq B$.

The latter argument may be written formally as the following proof by induction. Assume that $BB \subseteq B$.

Claim: For each $k \geq 1$, if $x_1, \dots, x_k \in B$, then $x_1 \cdots x_k \in B$.

Basis: Prove for $k = 1$. This statement is obviously true.

Induction step: For each $k \geq 1$, assume that the claim is true for k and prove it to be true for $k + 1$.

If $x_1, \dots, x_k, x_{k+1} \in B$, then by the induction assumption, $x_1 \cdots x_k \in B$. Therefore $x_1 \cdots x_k x_{k+1} \in BB$, but $BB \subseteq B$, so $x_1 \cdots x_{k+1} \in B$. That proves the induction step and the claim. The claim implies that, if $BB \subseteq B$, then $B^* \subseteq B$.

1.29 (a) Assume that $A_1 = \{0^n 1^n 2^n \mid n \geq 0\}$ is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string $0^p 1^p 2^p$. Because s is a member of A_1 and s is longer than p , the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^i z$ is in A_1 . Consider two possibilities:

1. The string y consists only of 0s, only of 1s, or only of 2s. In these cases the string $xyyz$ will not have equal numbers of 0s, 1s, and 2s. Hence $xyyz$ is not a member of A_1 , a contradiction.
2. The string y consists of more than one kind of symbol. In this case $xyyz$ will have the 0s, 1s, or 2s out of order. Hence $xyyz$ is not a member of A_1 , a contradiction.

Either way we arrive at a contradiction. Therefore, A_1 is not regular.

(c) Assume that $A_3 = \{a^{2^n} \mid n \geq 0\}$ is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string a^{2^p} . Because s is a member of A_1 and s is longer than p , the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, satisfying the three conditions of the pumping lemma.

The third condition tells us that $|xy| \leq p$. Furthermore, $p < 2^p$ and so $|y| < 2^p$. Therefore $|xyyz| = |xyz| + |y| < 2^p + 2^p = 2^{p+1}$. The second condition requires $|y| > 1$ so $2^p < |xyyz| < 2^{p+1}$. The length of $xyyz$ cannot be a power of 2. Hence $xyyz$ is not a member of A_3 , a contradiction. Therefore, A_3 is not regular.

1.40 Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA recognizing A , where A is some regular language. Construct $M' = (Q', \Sigma, \delta', q_0', F')$ recognizing $\text{NOPREFIX}(A)$ as follows:

1. $Q' = Q$.
2. For $r \in Q'$ and $a \in \Sigma$ define $\delta'(r, a) = \begin{cases} \delta(r, a) & \text{if } r \notin F \\ \emptyset & \text{if } r \in F. \end{cases}$
3. $q_0' = q_0$.
4. $F' = F$.

- 1.44** Let $M_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$ and $M_C = (Q_C, \Sigma, \delta_C, q_C, F_C)$ be DFAs recognizing B and C respectively. Construct NFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes $B \stackrel{!}{\leftarrow} C$ as follows. To decide whether its input w is in $B \stackrel{!}{\leftarrow} C$, the machine M checks that $w \in B$, and in parallel, nondeterministically guesses a string y that contains the same number of 1s as contained in w and checks that $y \in C$.

1. $Q = Q_B \times Q_C$.
2. For $(q, r) \in Q$ and $a \in \Sigma$ define

$$\delta((q, r), a) = \begin{cases} \{(\delta_B(q, 0), r)\} & \text{if } a = 0 \\ \{(\delta_B(q, 1), \delta_C(r, 1))\} & \text{if } a = 1 \\ \{(q, \delta_C(r, 0))\} & \text{if } a = \varepsilon. \end{cases}$$

3. $q_0 = (q_B, q_C)$.
4. $F = F_B \times F_C$.

- 1.46 (b)** Let $B = \{0^m 1^n \mid m \neq n\}$. Observe that $\overline{B} \cap 0^* 1^* = \{0^k 1^k \mid k \geq 0\}$. If B were regular, then \overline{B} would be regular and so would $\overline{B} \cap 0^* 1^*$. But we already know that $\{0^k 1^k \mid k \geq 0\}$ isn't regular, so B cannot be regular.

Alternatively, we can prove B to be nonregular by using the pumping lemma directly, though doing so is trickier. Assume that $B = \{0^m 1^n \mid m \neq n\}$ is regular. Let p be the pumping length given by the pumping lemma. Observe that $p!$ is divisible by all integers from 1 to p , where $p! = p(p-1)(p-2) \cdots 1$. The string $s = 0^p 1^{p+p!} \in B$, and $|s| \geq p$. Thus the pumping lemma implies that s can be divided as xyz with $x = 0^a$, $y = 0^b$, and $z = 0^c 1^{p+p!}$, where $b \geq 1$ and $a+b+c = p$. Let s' be the string $xy^{i+1}z$, where $i = p!/b$. Then $y^i = 0^{p!}$ so $y^{i+1} = 0^{b+p!}$, and so $xyz = 0^{a+b+c+p!} 1^{p+p!}$. That gives $xyz = 0^{p+p!} 1^{p+p!} \notin B$, a contradiction.

- 1.50** Assume to the contrary that some FST T outputs w^R on input w . Consider the input strings 00 and 01. On input 00, T must output 00, and on input 01, T must output 10. In both cases the first input bit is a 0 but the first output bits differ. Operating in this way is impossible for an FST because it produces its first output bit before it reads its second input. Hence no such FST can exist.

- 1.52 (a)** We prove this assertion by contradiction. Let M be a k -state DFA that recognizes L . Suppose for a contradiction that L has index greater than k . That means some set X with more than k elements is pairwise distinguishable by L . Because M has k states, the pigeonhole principle implies that X contains two distinct strings x and y , where $\delta(q_0, x) = \delta(q_0, y)$. Here $\delta(q_0, x)$ is the state that M is in after starting in the start state q_0 and reading input string x . Then, for any string $z \in \Sigma^*$, $\delta(q_0, xz) = \delta(q_0, yz)$. Therefore either both xz and yz are in L or neither are in L . But then x and y aren't distinguishable by L , contradicting our assumption that X is pairwise distinguishable by L .

(b) Let $X = \{s_1, \dots, s_k\}$ be pairwise distinguishable by L . We construct DFA $M = (Q, \Sigma, \delta, q_0, F)$ with k states recognizing L . Let $Q = \{q_1, \dots, q_k\}$, and define $\delta(q_i, a)$ to be q_j , where $s_j \equiv_L s_i a$ (the relation \equiv_L is defined in Problem 1.51). Note that $s_j \equiv_L s_i a$ for some $s_j \in X$; otherwise, $X \cup s_i a$ would have $k+1$ elements and would be pairwise distinguishable by L , which would contradict the assumption that L has index k . Let $F = \{q_i \mid s_i \in L\}$. Let the start state q_0 be the q_i such that $s_i \equiv_L \varepsilon$. M is constructed so that, for any state q_i , $\{s \mid \delta(q_0, s) = q_i\} = \{s \mid s \equiv_L s_i\}$. Hence M recognizes L .

(c) Suppose that L is regular and let k be the number of states in a DFA recognizing L . Then from part (a) L has index at most k . Conversely, if L has index k , then by part (b) it is recognized by a DFA with k states and thus is regular. To show that the index of L is the size of the smallest DFA accepting it, suppose that L 's index is *exactly* k . Then, by part (b), there is a k -state DFA accepting L . That is the smallest such DFA because if it were any smaller, then we could show by part (a) that the index of L is less than k .

1.55 (a) The minimum pumping length is 4. The string 000 is in the language but cannot be pumped, so 3 is not a pumping length for this language. If s has length 4 or more, it contains 1s. By dividing s onto xyz , where x is 000 and y is the first 1 and z is everything afterward, we satisfy the pumping lemma's three conditions.

(b) The minimum pumping length is 1. The pumping length cannot be 0 because the string ε is in the language and it cannot be pumped. Every nonempty string in the language can be divided into xyz , where $x = \varepsilon$ and y is the first character and z is the remainder. This division satisfies the three conditions.

(d) The minimum pumping length is 3. The pumping length cannot be 2 because the string 11 is in the language and it cannot be pumped. Let s be a string in the language of length at least 3. If s is generated by $0^*1^*0^*1^*$, we can write it as xyz , where x is the empty string, y is the first symbol of s , and z is the remainder of s . Breaking s up in this way shows that it can be pumped. If s is generated by 10^*1 , we can write it as xyz , where $x = 1$ and $y = 0$ and z is the remainder of s . This division gives a way to pump s .

2

CONTEXT-FREE LANGUAGES

In Chapter 1 we introduced two different, though equivalent, methods of describing languages: *finite automata* and *regular expressions*. We showed that many languages can be described in this way but that some simple languages, such as $\{0^n 1^n \mid n \geq 0\}$, cannot.

In this chapter we present *context-free grammars*, a more powerful method of describing languages. Such grammars can describe certain features that have a recursive structure, which makes them useful in a variety of applications.

Context-free grammars were first used in the study of human languages. One way of understanding the relationship of terms such as *noun*, *verb*, and *preposition* and their respective phrases leads to a natural recursion because noun phrases may appear inside verb phrases and vice versa. Context-free grammars can capture important aspects of these relationships.

An important application of context-free grammars occurs in the specification and compilation of programming languages. A grammar for a programming language often appears as a reference for people trying to learn the language syntax. Designers of compilers and interpreters for programming languages often start by obtaining a grammar for the language. Most compilers and interpreters contain a component called a *parser* that extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution. A number of methodologies facilitate the construction of a parser once a context-free grammar is available. Some tools even automatically generate the parser from the grammar.

The collection of languages associated with context-free grammars are called the **context-free languages**. They include all the regular languages and many additional languages. In this chapter, we give a formal definition of context-free grammars and study the properties of context-free languages. We also introduce **pushdown automata**, a class of machines recognizing the context-free languages. Pushdown automata are useful because they allow us to gain additional insight into the power of context-free grammars.

2.1 CONTEXT-FREE GRAMMARS

CONTEXT-FREE GRAMMARS

The following is an example of a context-free grammar, which we call G_1 .

$$\begin{aligned} A &\rightarrow 0A1 \\ A &\rightarrow B \\ B &\rightarrow \# \end{aligned}$$

A grammar consists of a collection of **substitution rules**, also called **productions**. Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a **variable**. The string consists of variables and other symbols called **terminals**. The variable symbols often are represented by capital letters. The terminals are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols. One variable is designated as the **start variable**. It usually occurs on the left-hand side of the topmost rule. For example, grammar G_1 contains three rules. G_1 's variables are A and B , where A is the start variable. Its terminals are 0, 1, and #.

You use a grammar to describe a language by generating each string of that language in the following manner.

1. Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise.
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.
3. Repeat step 2 until no variables remain.

For example, grammar G_1 generates the string 000#111. The sequence of substitutions to obtain a string is called a **derivation**. A derivation of string 000#111 in grammar G_1 is

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

You may also represent the same information pictorially with a **parse tree**. An example of a parse tree is shown in Figure 2.1.

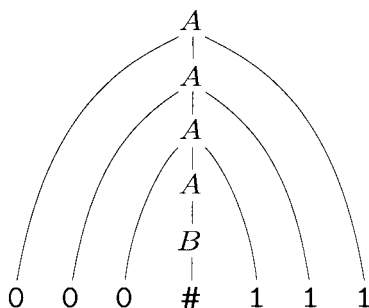


FIGURE 2.1
Parse tree for 000#111 in grammar G_1

All strings generated in this way constitute the *language of the grammar*. We write $L(G_1)$ for the language of grammar G_1 . Some experimentation with the grammar G_1 shows us that $L(G_1)$ is $\{0^n\#1^n \mid n \geq 0\}$. Any language that can be generated by some context-free grammar is called a *context-free language* (CFL). For convenience when presenting a context-free grammar, we abbreviate several rules with the same left-hand variable, such as $A \rightarrow 0A1$ and $A \rightarrow B$, into a single line $A \rightarrow 0A1 \mid B$, using the symbol “ \mid ” as an “or.”

The following is a second example of a context-free grammar, called G_2 , which describes a fragment of the English language.

$$\begin{aligned} \langle \text{SENTENCE} \rangle &\rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\ \langle \text{NOUN-PHRASE} \rangle &\rightarrow \langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \\ \langle \text{VERB-PHRASE} \rangle &\rightarrow \langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle \\ \langle \text{PREP-PHRASE} \rangle &\rightarrow \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \\ \langle \text{CMPLX-NOUN} \rangle &\rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\ \langle \text{CMPLX-VERB} \rangle &\rightarrow \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \\ \langle \text{ARTICLE} \rangle &\rightarrow \text{a} \mid \text{the} \\ \langle \text{NOUN} \rangle &\rightarrow \text{boy} \mid \text{girl} \mid \text{flower} \\ \langle \text{VERB} \rangle &\rightarrow \text{touches} \mid \text{likes} \mid \text{sees} \\ \langle \text{PREP} \rangle &\rightarrow \text{with} \end{aligned}$$

Grammar G_2 has 10 variables (the capitalized grammatical terms written inside brackets); 27 terminals (the standard English alphabet plus a space character); and 18 rules. Strings in $L(G_2)$ include

a boy sees
the boy sees a flower
a girl with a flower likes the boy

Each of these strings has a derivation in grammar G_2 . The following is a derivation of the first string on this list.

$$\begin{aligned}
\langle \text{SENTENCE} \rangle &\Rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \langle \text{CMPLX-NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow a \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow a \text{ boy } \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow a \text{ boy } \langle \text{CMPLX-VERB} \rangle \\
&\Rightarrow a \text{ boy } \langle \text{VERB} \rangle \\
&\Rightarrow a \text{ boy sees}
\end{aligned}$$

FORMAL DEFINITION OF A CONTEXT-FREE GRAMMAR

Let's formalize our notion of a context-free grammar (CFG).

DEFINITION 2.2

A *context-free grammar* is a 4-tuple (V, Σ, R, S) , where

1. V is a finite set called the *variables*,
2. Σ is a finite set, disjoint from V , called the *terminals*,
3. R is a finite set of *rules*, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

If u, v , and w are strings of variables and terminals, and $A \rightarrow w$ is a rule of the grammar, we say that uAv *yields* uwv , written $uAv \Rightarrow uwv$. Say that u *derives* v , written $u \Rightarrow^* v$, if $u = v$ or if a sequence u_1, u_2, \dots, u_k exists for $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k \Rightarrow v.$$

The *language of the grammar* is $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$.

In grammar G_1 , $V = \{A, B\}$, $\Sigma = \{0, 1, \#\}$, $S = A$, and R is the collection of the three rules appearing on page 100. In grammar G_2 ,

$$\begin{aligned}
V = \{ &\langle \text{SENTENCE} \rangle, \langle \text{NOUN-PHRASE} \rangle, \langle \text{VERB-PHRASE} \rangle, \\
&\langle \text{PREP-PHRASE} \rangle, \langle \text{CMPLX-NOUN} \rangle, \langle \text{CMPLX-VERB} \rangle, \\
&\langle \text{ARTICLE} \rangle, \langle \text{NOUN} \rangle, \langle \text{VERB} \rangle, \langle \text{PREP} \rangle \},
\end{aligned}$$

and $\Sigma = \{a, b, c, \dots, z, " \}$. The symbol " " is the blank symbol, placed invisibly after each word (a, boy, etc.), so the words won't run together.

Often we specify a grammar by writing down only its rules. We can identify the variables as the symbols that appear on the left-hand side of the rules and the terminals as the remaining symbols. By convention, the start variable is the variable on the left-hand side of the first rule.

EXAMPLES OF CONTEXT-FREE GRAMMARS

EXAMPLE 2.3

Consider grammar $G_3 = (\{S\}, \{a, b\}, R, S)$. The set of rules, R , is

$$S \rightarrow aSb \mid SS \mid \varepsilon.$$

This grammar generates strings such as $abab$, $aaabbbb$, and $aababb$. You can see more easily what this language is if you think of a as a left parenthesis “(” and b as a right parenthesis “)”. Viewed in this way, $L(G_3)$ is the language of all strings of properly nested parentheses.

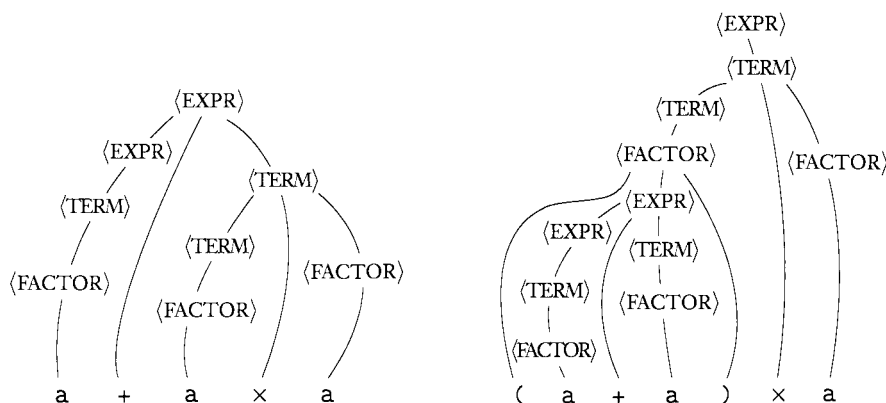
EXAMPLE 2.4

Consider grammar $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$.

V is $\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$ and Σ is $\{a, +, \times, (,)\}$. The rules are

$$\begin{aligned} \langle \text{EXPR} \rangle &\rightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\ \langle \text{TERM} \rangle &\rightarrow \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\ \langle \text{FACTOR} \rangle &\rightarrow (\langle \text{EXPR} \rangle) \mid a \end{aligned}$$

The two strings $a+axa$ and $(a+a)xa$ can be generated with grammar G_4 . The parse trees are shown in the following figure.

**FIGURE 2.5**

Parse trees for the strings $a+axa$ and $(a+a)xa$

A compiler translates code written in a programming language into another form, usually one more suitable for execution. To do so the compiler extracts the meaning of the code to be compiled in a process called *parsing*. One rep-

resentation of this meaning is the parse tree for the code, in the context-free grammar for the programming language. We discuss an algorithm that parses context-free languages later in Theorem 7.16 and in Problem 7.43.

Grammar G_4 describes a fragment of a programming language concerned with arithmetic expressions. Observe how the parse trees in Figure 2.5 “group” the operations. The tree for $a+axa$ groups the \times operator and its operands (the second two a ’s) as one operand of the $+$ operator. In the tree for $(a+a) \times a$, the grouping is reversed. These groupings fit the standard precedence of multiplication before addition and the use of parentheses to override the standard precedence. Grammar G_4 is designed to capture these precedence relations.

DESIGNING CONTEXT-FREE GRAMMARS

As with the design of finite automata, discussed in Section 1.1 (page 41), the design of context-free grammars requires creativity. Indeed, context-free grammars are even trickier to construct than finite automata because we are more accustomed to programming a machine for specific tasks than we are to describing languages with grammars. The following techniques are helpful, singly or in combination, when you’re faced with the problem of constructing a CFG.

First, many CFLs are the union of simpler CFLs. If you must construct a CFG for a CFL that you can break into simpler pieces, do so and then construct individual grammars for each piece. These individual grammars can be easily merged into a grammar for the original language by combining their rules and then adding the new rule $S \rightarrow S_1 \mid S_2 \mid \cdots \mid S_k$, where the variables S_i are the start variables for the individual grammars. Solving several simpler problems is often easier than solving one complicated problem.

For example, to get a grammar for the language $\{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$, first construct the grammar

$$S_1 \rightarrow 0S_11 \mid \epsilon$$

for the language $\{0^n 1^n \mid n \geq 0\}$ and the grammar

$$S_2 \rightarrow 1S_20 \mid \epsilon$$

for the language $\{1^n 0^n \mid n \geq 0\}$ and then add the rule $S \rightarrow S_1 \mid S_2$ to give the grammar

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow 0S_11 \mid \epsilon \\ S_2 &\rightarrow 1S_20 \mid \epsilon. \end{aligned}$$

Second, constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language. You can convert any DFA into an equivalent CFG as follows. Make a variable R_i for each state q_i of the DFA. Add the rule $R_i \rightarrow aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA. Add the rule $R_i \rightarrow \epsilon$ if q_i is an accept state of the DFA. Make R_0 the start variable of the grammar, where q_0 is the start state of the machine. Verify on your own that the resulting CFG generates the same language that the DFA recognizes.

Third, certain context-free languages contain strings with two substrings that are “linked” in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring. This situation occurs in the language $\{0^n 1^n \mid n \geq 0\}$ because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s. You can construct a CFG to handle this situation by using a rule of the form $R \rightarrow uRv$, which generates strings wherein the portion containing the u ’s corresponds to the portion containing the v ’s.

Finally, in more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures. That situation occurs in the grammar that generates arithmetic expressions in Example 2.4. Any time the symbol a appears, an entire parenthesized expression might appear recursively instead. To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.

AMBIGUITY

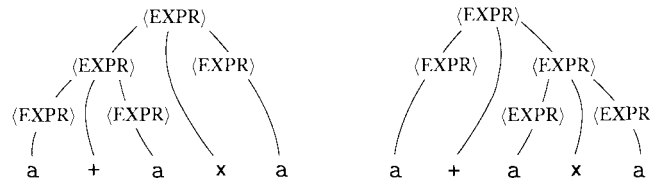
Sometimes a grammar can generate the same string in several different ways. Such a string will have several different parse trees and thus several different meanings. This result may be undesirable for certain applications, such as programming languages, where a given program should have a unique interpretation.

If a grammar generates the same string in several different ways, we say that the string is derived *ambiguously* in that grammar. If a grammar generates some string ambiguously we say that the grammar is *ambiguous*.

For example, consider grammar G_5 :

$$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid a$$

This grammar generates the string $a+a \times a$ ambiguously. The following figure shows the two different parse trees.

**FIGURE 2.6**

The two parse trees for the string $a+axa$ in grammar G_5

This grammar doesn't capture the usual precedence relations and so may group the $+$ before the \times or vice versa. In contrast grammar G_4 generates exactly the same language, but every generated string has a unique parse tree. Hence G_4 is unambiguous, whereas G_5 is ambiguous.

Grammar G_2 (page 101) is another example of an ambiguous grammar. The sentence *the girl touches the boy with the flower* has two different derivations. In Exercise 2.8 you are asked to give the two parse trees and observe their correspondence with the two different ways to read that sentence.

Now we formalize the notion of ambiguity. When we say that a grammar generates a string ambiguously, we mean that the string has two different parse trees, not two different derivations. Two derivations may differ merely in the order in which they replace variables yet not in their overall structure. To concentrate on structure we define a type of derivation that replaces variables in a fixed order. A derivation of a string w in a grammar G is a **leftmost derivation** if at every step the leftmost remaining variable is the one replaced. The derivation preceding Definition 2.2 (page 102) is a leftmost derivation.

DEFINITION 2.7

A string w is derived **ambiguously** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is **ambiguous** if it generates some string ambiguously.

Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language. Some context-free languages, however, can be generated only by ambiguous grammars. Such languages are called **inherently ambiguous**. Problem 2.29 asks you to prove that the language $\{a^i b^j c^k \mid i = j \text{ or } j = k\}$ is inherently ambiguous.

CHOMSKY NORMAL FORM

When working with context-free grammars, it is often convenient to have them in simplified form. One of the simplest and most useful forms is called the

Chomsky normal form. Chomsky normal form is useful in giving algorithms for working with context-free grammars, as we do in Chapters 4 and 7.

DEFINITION 2.8

A context-free grammar is in *Chomsky normal form* if every rule is of the form

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \end{aligned}$$

where a is any terminal and A , B , and C are any variables—except that B and C may not be the start variable. In addition we permit the rule $S \rightarrow \epsilon$, where S is the start variable.

THEOREM 2.9

Any context-free language is generated by a context-free grammar in Chomsky normal form.

PROOF IDEA We can convert any grammar G into Chomsky normal form. The conversion has several stages wherein rules that violate the conditions are replaced with equivalent ones that are satisfactory. First, we add a new start variable. Then, we eliminate all ϵ rules of the form $A \rightarrow \epsilon$. We also eliminate all *unit rules* of the form $A \rightarrow B$. In both cases we patch up the grammar to be sure that it still generates the same language. Finally, we convert the remaining rules into the proper form.

PROOF First, we add a new start variable S_0 and the rule $S_0 \rightarrow S$, where S was the original start variable. This change guarantees that the start variable doesn't occur on the right-hand side of a rule.

Second, we take care of all ϵ rules. We remove an ϵ -rule $A \rightarrow \epsilon$, where A is not the start variable. Then for each occurrence of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted. In other words, if $R \rightarrow uAv$ is a rule in which u and v are strings of variables and terminals, we add rule $R \rightarrow uv$. We do so for each *occurrence* of an A , so the rule $R \rightarrow uAvAw$ causes us to add $R \rightarrow uvAw$, $R \rightarrow uAvw$, and $R \rightarrow uvw$. If we have the rule $R \rightarrow A$, we add $R \rightarrow \epsilon$ unless we had previously removed the rule $R \rightarrow \epsilon$. We repeat these steps until we eliminate all ϵ rules not involving the start variable.

Third, we handle all unit rules. We remove a unit rule $A \rightarrow B$. Then, whenever a rule $B \rightarrow u$ appears, we add the rule $A \rightarrow u$ unless this was a unit rule previously removed. As before, u is a string of variables and terminals. We repeat these steps until we eliminate all unit rules.

Finally, we convert all remaining rules into the proper form. We replace each rule $A \rightarrow u_1u_2 \cdots u_k$, where $k \geq 3$ and each u_i is a variable or terminal symbol,

with the rules $A \rightarrow u_1 A_1$, $A_1 \rightarrow u_2 A_2$, $A_2 \rightarrow u_3 A_3, \dots$, and $A_{k-2} \rightarrow u_{k-1} u_k$. The A_i 's are new variables. If $k = 2$, we replace any terminal u_i in the preceding rule(s) with the new variable U_i and add the rule $U_i \rightarrow u_i$.

EXAMPLE 2.10

Let G_6 be the following CFG and convert it to Chomsky normal form by using the conversion procedure just given. The series of grammars presented illustrates the steps in the conversion. Rules shown in bold have just been added. Rules shown in gray have just been removed.

1. The original CFG G_6 is shown on the left. The result of applying the first step to make a new start variable appears on the right.

$$\begin{array}{ll}
 S \rightarrow ASA \mid aB & S_0 \rightarrow S \\
 A \rightarrow B \mid S & S \rightarrow ASA \mid aB \\
 B \rightarrow b \mid \varepsilon & A \rightarrow B \mid S \\
 & B \rightarrow b \mid \varepsilon
 \end{array}$$

2. Remove ε rules $B \rightarrow \varepsilon$, shown on the left, and $A \rightarrow \varepsilon$, shown on the right.

$$\begin{array}{ll}
 S_0 \rightarrow S & S_0 \rightarrow S \\
 S \rightarrow ASA \mid aB \mid a & S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \mid S \\
 A \rightarrow B \mid S \mid \varepsilon & A \rightarrow B \mid S \\
 B \rightarrow b \mid \varepsilon & B \rightarrow b
 \end{array}$$

- 3a. Remove unit rules $S \rightarrow S$, shown on the left, and $S_0 \rightarrow S$, shown on the right.

$$\begin{array}{ll}
 S_0 \rightarrow S & S_0 \rightarrow S \mid ASA \mid aB \mid a \mid SA \mid AS \\
 S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \mid S & S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \\
 A \rightarrow B \mid S & A \rightarrow B \mid S \\
 B \rightarrow b & B \rightarrow b
 \end{array}$$

- 3b. Remove unit rules $A \rightarrow B$ and $A \rightarrow S$.

$$\begin{array}{ll}
 S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS & S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS \\
 S \rightarrow ASA \mid aB \mid a \mid SA \mid AS & S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \\
 A \rightarrow B \mid S \mid b & A \rightarrow S \mid b \mid ASA \mid aB \mid a \mid SA \mid AS \\
 B \rightarrow b & B \rightarrow b
 \end{array}$$

4. Convert the remaining rules into the proper form by adding additional variables and rules. The final grammar in Chomsky normal form is equivalent to G_6 , which follows. (Actually the procedure given in Theorem 2.9 produces several variables U_i along with several rules $U_i \rightarrow a$. We simplified the resulting grammar by using a single variable U and rule $U \rightarrow a$.)

$$\begin{aligned} S_0 &\rightarrow AA_1 \mid UB \mid a \mid SA \mid AS \\ S &\rightarrow AA_1 \mid UB \mid a \mid SA \mid AS \\ A &\rightarrow b \mid AA_1 \mid UB \mid a \mid SA \mid AS \\ A_1 &\rightarrow SA \\ U &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

2.2 PUSHDOWN AUTOMATA

In this section we introduce a new type of computational model called *pushdown automata*. These automata are like nondeterministic finite automata but have an extra component called a *stack*. The stack provides additional memory beyond the finite amount available in the control. The stack allows pushdown automata to recognize some nonregular languages.

Pushdown automata are equivalent in power to context-free grammars. This equivalence is useful because it gives us two options for proving that a language is context free. We can give either a context-free grammar generating it or a pushdown automaton recognizing it. Certain languages are more easily described in terms of generators, whereas others are more easily described in terms of recognizers.

The following figure is a schematic representation of a finite automaton. The control represents the states and transition function, the tape contains the input string, and the arrow represents the input head, pointing at the next input symbol to be read.

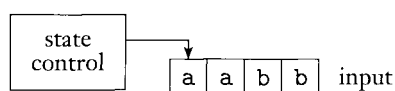


FIGURE 2.11
Schematic of a finite automaton

With the addition of a stack component we obtain a schematic representation of a pushdown automaton, as shown in the following figure.

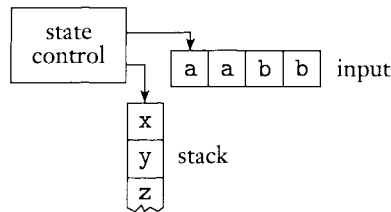


FIGURE 2.12
Schematic of a pushdown automaton

A pushdown automaton (PDA) can write symbols on the stack and read them back later. Writing a symbol “pushes down” all the other symbols on the stack. At any time the symbol on the top of the stack can be read and removed. The remaining symbols then move back up. Writing a symbol on the stack is often referred to as *pushing* the symbol, and removing a symbol is referred to as *popping* it. Note that all access to the stack, for both reading and writing, may be done only at the top. In other words a stack is a “last in, first out” storage device. If certain information is written on the stack and additional information is written afterward, the earlier information becomes inaccessible until the later information is removed.

Plates on a cafeteria serving counter illustrate a stack. The stack of plates rests on a spring so that when a new plate is placed on top of the stack, the plates below it move down. The stack on a pushdown automaton is like a stack of plates, with each plate having a symbol written on it.

A stack is valuable because it can hold an unlimited amount of information. Recall that a finite automaton is unable to recognize the language $\{0^n 1^n \mid n \geq 0\}$ because it cannot store very large numbers in its finite memory. A PDA is able to recognize this language because it can use its stack to store the number of 0s it has seen. Thus the unlimited nature of a stack allows the PDA to store numbers of unbounded size. The following informal description shows how the automaton for this language works.

Read symbols from the input. As each 0 is read, push it onto the stack. As soon as 1s are seen, pop a 0 off the stack for each 1 read. If reading the input is finished exactly when the stack becomes empty of 0s, accept the input. If the stack becomes empty while 1s remain or if the 1s are finished while the stack still contains 0s or if any 0s appear in the input following 1s, reject the input.

As mentioned earlier, pushdown automata may be nondeterministic. Deterministic and nondeterministic pushdown automata are *not* equivalent in power.

Nondeterministic pushdown automata recognize certain languages which no deterministic pushdown automata can recognize, though we will not prove this fact. We give languages requiring nondeterminism in Examples 2.16 and 2.18. Recall that deterministic and nondeterministic finite automata do recognize the same class of languages, so the pushdown automata situation is different. We focus on nondeterministic pushdown automata because these automata are equivalent in power to context-free grammars.

FORMAL DEFINITION OF A PUSHDOWN AUTOMATON

The formal definition of a pushdown automaton is similar to that of a finite automaton, except for the stack. The stack is a device containing symbols drawn from some alphabet. The machine may use different alphabets for its input and its stack, so now we specify both an input alphabet Σ and a stack alphabet Γ .

At the heart of any formal definition of an automaton is the transition function, which describes its behavior. Recall that $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$. The domain of the transition function is $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$. Thus the current state, next input symbol read, and top symbol of the stack determine the next move of a pushdown automaton. Either symbol may be ϵ , causing the machine to move without reading a symbol from the input or without reading a symbol from the stack.

For the range of the transition function we need to consider what to allow the automaton to do when it is in a particular situation. It may enter some new state and possibly write a symbol on the top of the stack. The function δ can indicate this action by returning a member of Q together with a member of Γ_ϵ , that is, a member of $Q \times \Gamma_\epsilon$. Because we allow nondeterminism in this model, a situation may have several legal next moves. The transition function incorporates nondeterminism in the usual way, by returning a set of members of $Q \times \Gamma_\epsilon$, that is, a member of $\mathcal{P}(Q \times \Gamma_\epsilon)$. Putting it all together, our transition function δ takes the form $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$.

DEFINITION 2.13

A *pushdown automaton* is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q , Σ , Γ , and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows. It accepts input w if w can be written as $w = w_1 w_2 \cdots w_m$, where each $w_i \in \Sigma_\varepsilon$ and sequences of states $r_0, r_1, \dots, r_m \in Q$ and strings $s_0, s_1, \dots, s_m \in \Gamma^*$ exist that satisfy the following three conditions. The strings s_i represent the sequence of stack contents that M has on the accepting branch of the computation.

1. $r_0 = q_0$ and $s_0 = \varepsilon$. This condition signifies that M starts out properly, in the start state and with an empty stack.
2. For $i = 0, \dots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol.
3. $r_m \in F$. This condition states that an accept state occurs at the input end.

EXAMPLES OF PUSHDOWN AUTOMATA

EXAMPLE 2.14

The following is the formal description of the PDA (page 110) that recognizes the language $\{0^n 1^n \mid n \geq 0\}$. Let M_1 be $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where

$$Q = \{q_1, q_2, q_3, q_4\},$$

$$\Sigma = \{0, 1\},$$

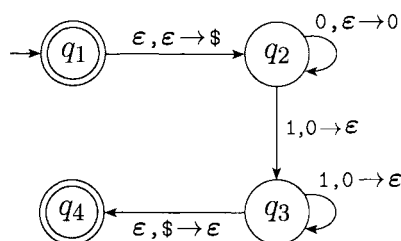
$$\Gamma = \{0, \$\},$$

$$F = \{q_1, q_4\}, \text{ and}$$

δ is given by the following table, wherein blank entries signify \emptyset .

Input:	0			1			ε		
Stack:	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$		$\{(q_3, \varepsilon)\}$				
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									

We can also use a state diagram to describe a PDA, as shown in the Figures 2.15, 2.17, and 2.19. Such diagrams are similar to the state diagrams used to describe finite automata, modified to show how the PDA uses its stack when going from state to state. We write " $a, b \rightarrow c$ " to signify that when the machine is reading an a from the input it may replace the symbol b on the top of the stack with a c . Any of a , b , and c may be ε . If a is ε , the machine may make this transition without reading any symbol from the input. If b is ε , the machine may make this transition without reading and popping any symbol from the stack. If c is ε , the machine does not write any symbol on the stack when going along this transition.

**FIGURE 2.15**

State diagram for the PDA M_1 that recognizes $\{0^n 1^n \mid n \geq 0\}$

The formal definition of a PDA contains no explicit mechanism to allow the PDA to test for an empty stack. This PDA is able to get the same effect by initially placing a special symbol $\$$ on the stack. Then if it ever sees the $\$$ again, it knows that the stack effectively is empty. Subsequently, when we refer to testing for an empty stack in an informal description of a PDA, we implement the procedure in the same way.

Similarly, PDAs cannot test explicitly for having reached the end of the input string. This PDA is able to achieve that effect because the accept state takes effect only when the machine is at the end of the input. Thus from now on, we assume that PDAs can test for the end of the input, and we know that we can implement it in the same manner.

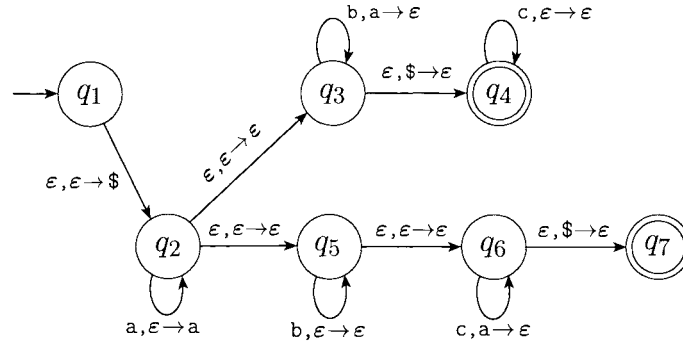
EXAMPLE 2.16

This example illustrates a pushdown automaton that recognizes the language

$$\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}.$$

Informally the PDA for this language works by first reading and pushing the a's. When the a's are done the machine has all of them on the stack so that it can match them with either the b's or the c's. This maneuver is a bit tricky because the machine doesn't know in advance whether to match the a's with the b's or the c's. Nondeterminism comes in handy here.

Using its nondeterminism, the PDA can guess whether to match the a's with the b's or with the c's, as shown in the following figure. Think of the machine as having two branches of its nondeterminism, one for each possible guess. If either of them match, that branch accepts and the entire machine accepts. In fact we could show, though we do not do so, that nondeterminism is *essential* for recognizing this language with a PDA.

**FIGURE 2.17**

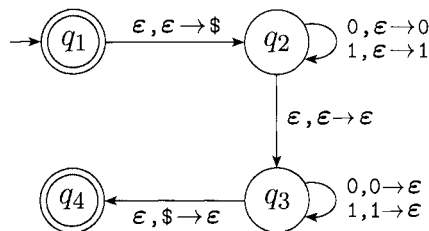
State diagram for PDA M_2 that recognizes $\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$

EXAMPLE 2.18

In this example we give a PDA M_3 recognizing the language $\{ww^R \mid w \in \{0,1\}^*\}$. Recall that w^R means w written backwards. The informal description of the PDA follows.

Begin by pushing the symbols that are read onto the stack. At each point nondeterministically guess that the middle of the string has been reached and then change into popping off the stack for each symbol read, checking to see that they are the same. If they were always the same symbol and the stack empties at the same time as the input is finished, accept; otherwise reject.

The following is the diagram of this machine.

**FIGURE 2.19**

State diagram for the PDA M_3 that recognizes $\{ww^R \mid w \in \{0,1\}^*\}$

EQUIVALENCE WITH CONTEXT-FREE GRAMMARS

In this section we show that context-free grammars and pushdown automata are equivalent in power. Both are capable of describing the class of context-free languages. We show how to convert any context-free grammar into a pushdown automaton that recognizes the same language and vice versa. Recalling that we defined a context-free language to be any language that can be described with a context-free grammar, our objective is the following theorem.

THEOREM 2.20

A language is context free if and only if some pushdown automaton recognizes it.

As usual for “if and only if” theorems, we have two directions to prove. In this theorem, both directions are interesting. First, we do the easier forward direction.

LEMMA 2.21

If a language is context free, then some pushdown automaton recognizes it.

PROOF IDEA Let A be a CFL. From the definition we know that A has a CFG, G , generating it. We show how to convert G into an equivalent PDA, which we call P .

The PDA P that we now describe will work by accepting its input w , if G generates that input, by determining whether there is a derivation for w . Recall that a derivation is simply the sequence of substitutions made as a grammar generates a string. Each step of the derivation yields an *intermediate string* of variables and terminals. We design P to determine whether some series of substitutions using the rules of G can lead from the start variable to w .

One of the difficulties in testing whether there is a derivation for w is in figuring out which substitutions to make. The PDA’s nondeterminism allows it to guess the sequence of correct substitutions. At each step of the derivation one of the rules for a particular variable is selected nondeterministically and used to substitute for that variable.

The PDA P begins by writing the start variable on its stack. It goes through a series of intermediate strings, making one substitution after another. Eventually it may arrive at a string that contains only terminal symbols, meaning that it has used the grammar to derive a string. Then P accepts if this string is identical to the string it has received as input.

Implementing this strategy on a PDA requires one additional idea. We need to see how the PDA stores the intermediate strings as it goes from one to another. Simply using the stack for storing each intermediate string is tempting. However, that doesn’t quite work because the PDA needs to find the variables in the intermediate string and make substitutions. The PDA can access only the top

symbol on the stack and that may be a terminal symbol instead of a variable. The way around this problem is to keep only *part* of the intermediate string on the stack: the symbols starting with the first variable in the intermediate string. Any terminal symbols appearing before the first variable are matched immediately with symbols in the input string. The following figure shows the PDA P .

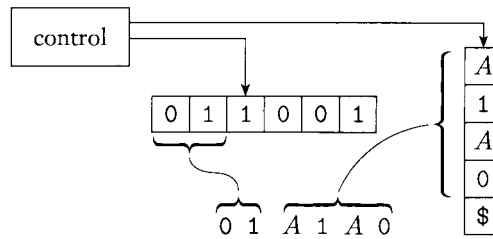


FIGURE 2.22
 P representing the intermediate string $01A1A0$

The following is an informal description of P .

1. Place the marker symbol $\$$ and the start variable on the stack.
2. Repeat the following steps forever.
 - a. If the top of stack is a variable symbol A , nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.
 - b. If the top of stack is a terminal symbol a , read the next symbol from the input and compare it to a . If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
 - c. If the top of stack is the symbol $\$$, enter the accept state. Doing so accepts the input if it has all been read.

PROOF We now give the formal details of the construction of the pushdown automaton $P = (Q, \Sigma, \Gamma, \delta, q_1, F)$. To make the construction clearer we use shorthand notation for the transition function. This notation provides a way to write an entire string on the stack in one step of the machine. We can simulate this action by introducing additional states to write the string one symbol at a time, as implemented in the following formal construction.

Let q and r be states of the PDA and let a be in Σ_ϵ and s be in Γ_ϵ . Say that we want the PDA to go from q to r when it reads a and pops s . Furthermore we want it to push the entire string $u = u_1 \cdots u_l$ on the stack at the same time. We can implement this action by introducing new states q_1, \dots, q_{l-1} and setting the

transition function as follows

$$\begin{aligned} \delta(q, a, s) &\text{ to contain } (q_1, u_l), \\ \delta(q_1, \varepsilon, \varepsilon) &= \{(q_2, u_{l-1})\}, \\ \delta(q_2, \varepsilon, \varepsilon) &= \{(q_3, u_{l-2})\}, \\ &\vdots \\ \delta(q_{l-1}, \varepsilon, \varepsilon) &= \{(r, u_1)\}. \end{aligned}$$

We use the notation $(r, u) \in \delta(q, a, s)$ to mean that when q is the state of the automaton, a is the next input symbol, and s is the symbol on the top of the stack, the PDA may read the a and pop the s , then push the string u onto the stack and go on to the state r . The following figure shows this implementation.

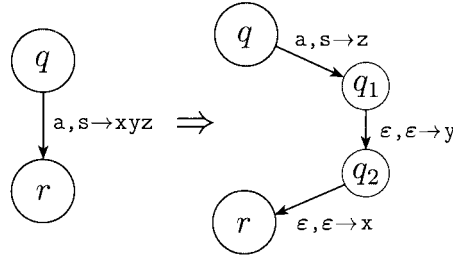


FIGURE 2.23
Implementing the shorthand $(r, xyz) \in \delta(q, a, s)$

The states of P are $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\} \cup E$, where E is the set of states we need for implementing the shorthand just described. The start state is q_{start} . The only accept state is q_{accept} .

The transition function is defined as follows. We begin by initializing the stack to contain the symbols $\$$ and S , implementing step 1 in the informal description: $\delta(q_{\text{start}}, \varepsilon, \varepsilon) = \{(q_{\text{loop}}, S\$)\}$. Then we put in transitions for the main loop of step 2.

First, we handle case (a) wherein the top of the stack contains a variable. Let $\delta(q_{\text{loop}}, \varepsilon, A) = \{(q_{\text{loop}}, w) \mid \text{where } A \rightarrow w \text{ is a rule in } R\}$.

Second, we handle case (b) wherein the top of the stack contains a terminal. Let $\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \varepsilon)\}$.

Finally, we handle case (c) wherein the empty stack marker $\$$ is on the top of the stack. Let $\delta(q_{\text{loop}}, \varepsilon, \$) = \{(q_{\text{accept}}, \varepsilon)\}$.

The state diagram is shown in Figure 2.24

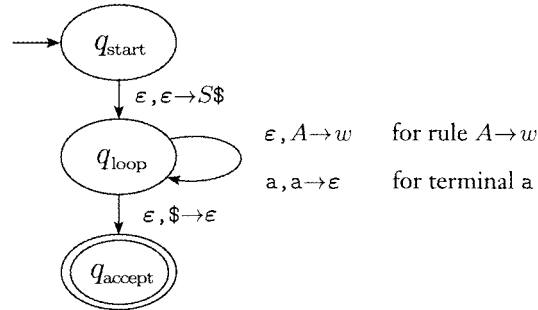


FIGURE 2.24
State diagram of P

That completes the proof of Lemma 2.21.

EXAMPLE 2.25

We use the procedure developed in Lemma 2.21 to construct a PDA P_1 from the following CFG G .

$$\begin{aligned} S &\rightarrow aTb \mid b \\ T &\rightarrow Ta \mid \epsilon \end{aligned}$$

The transition function is shown in the following diagram.

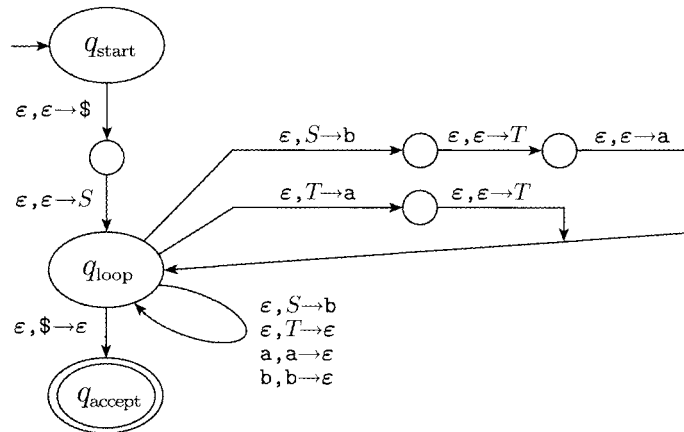


FIGURE 2.26
State diagram of P_1

Now we prove the reverse direction of Theorem 2.20. For the forward direction we gave a procedure for converting a CFG into a PDA. The main idea was to design the automaton so that it simulates the grammar. Now we want to give a procedure for going the other way: converting a PDA into a CFG. We design the grammar to simulate the automaton. This task is a bit tricky because “programming” an automaton is easier than “programming” a grammar.

LEMMA 2.27

If a pushdown automaton recognizes some language, then it is context free.

PROOF IDEA We have a PDA P , and we want to make a CFG G that generates all the strings that P accepts. In other words, G should generate a string if that string causes the PDA to go from its start state to an accept state.

To achieve this outcome we design a grammar that does somewhat more. For each pair of states p and q in P the grammar will have a variable A_{pq} . This variable generates all the strings that can take P from p with an empty stack to q with an empty stack. Observe that such strings can also take P from p to q , regardless of the stack contents at p , leaving the stack at q in the same condition as it was at p .

First, we simplify our task by modifying P slightly to give it the following three features.

1. It has a single accept state, q_{accept} .
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time.

Giving P features 1 and 2 is easy. To give it feature 3, we replace each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state, and we replace each transition that neither pops nor pushes with a two transition sequence that pushes then pops an arbitrary stack symbol.

To design G so that A_{pq} generates all strings that take P from p to q , starting and ending with an empty stack, we must understand how P operates on these strings. For any such string x , P 's first move on x must be a push, because every move is either a push or a pop and P can't pop an empty stack. Similarly, the last move on x must be a pop, because the stack ends up empty.

Two possibilities occur during P 's computation on x . Either the symbol popped at the end is the symbol that was pushed at the beginning, or not. If so, the stack is empty only at the beginning and end of P 's computation on x . If not, the initially pushed symbol must get popped at some point before the end of x and thus the stack becomes empty at this point. We simulate the former possibility with the rule $A_{pq} \rightarrow aA_{rs}b$, where a is the input read at the first move, b is the input read at the last move, r is the state following p , and s is the state preceding q . We simulate the latter possibility with the rule $A_{pq} \rightarrow A_{pr}A_{rq}$, where r is the state when the stack becomes empty.

PROOF Say that $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ and construct G . The variables of G are $\{A_{pq} \mid p, q \in Q\}$. The start variable is $A_{q_0, q_{\text{accept}}}$. Now we describe G 's rules.

- For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε) , put the rule $A_{pq} \rightarrow aA_{rs}b$ in G .
- For each $p, q, r \in Q$, put the rule $A_{pq} \rightarrow A_{pr}A_{rq}$ in G .
- Finally, for each $p \in Q$, put the rule $A_{pp} \rightarrow \varepsilon$ in G .

You may gain some insight for this construction from the following figures.

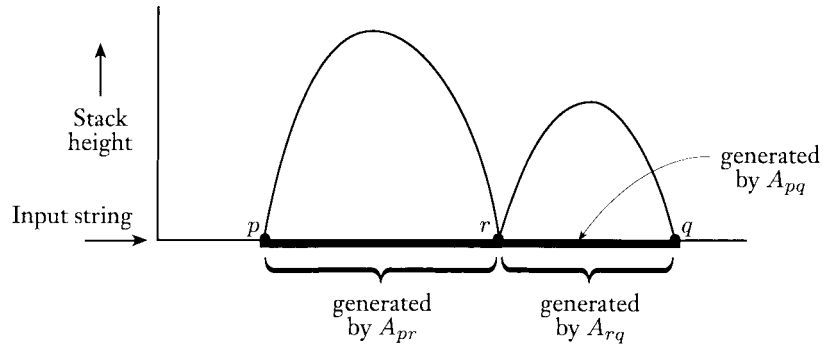


FIGURE 2.28
PDA computation corresponding to the rule $A_{pq} \rightarrow A_{pr}A_{rq}$

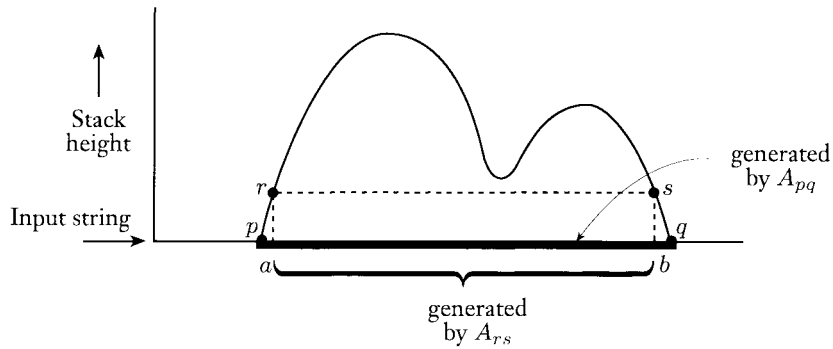


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

Now we prove that this construction works by demonstrating that A_{pq} generates x if and only if (iff) x can bring P from p with empty stack to q with empty stack. We consider each direction of the iff as a separate claim.

CLAIM 2.30

If A_{pq} generates x , then x can bring P from p with empty stack to q with empty stack.

We prove this claim by induction on the number of steps in the derivation of x from A_{pq} .

Basis: The derivation has 1 step.

A derivation with a single step must use a rule whose right-hand side contains no variables. The only rules in G where no variables occur on the right-hand side are $A_{pp} \rightarrow \varepsilon$. Clearly, input ε takes P from p with empty stack to p with empty stack so the basis is proved.

Induction step: Assume true for derivations of length at most k , where $k \geq 1$, and prove true for derivations of length $k + 1$.

Suppose that $A_{pq} \xRightarrow{*} x$ with $k + 1$ steps. The first step in this derivation is either $A_{pq} \Rightarrow aA_{rs}b$ or $A_{pq} \Rightarrow A_{pr}A_{rq}$. We handle these two cases separately.

In the first case, consider the portion y of x that A_{rs} generates, so $x = ayb$. Because $A_{rs} \xRightarrow{*} y$ with k steps, the induction hypothesis tells us that P can go from r on empty stack to s on empty stack. Because $A_{pq} \rightarrow aA_{rs}b$ is a rule of G , $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε) , for some stack symbol t . Hence, if P starts at p with an empty stack, after reading a it can go to state r and push t onto the stack. Then reading string y can bring it to s and leave t on the stack. Then after reading b it can go to state q and pop t off the stack. Therefore x can bring it from p with empty stack to q with empty stack.

In the second case, consider the portions y and z of x that A_{pr} and A_{rq} respectively generate, so $x = yz$. Because $A_{pr} \xRightarrow{*} y$ in at most k steps and $A_{rq} \xRightarrow{*} z$ in at most k steps, the induction hypothesis tells us that y can bring P from p to r , and z can bring P from r to q , with empty stacks at the beginning and end. Hence x can bring it from p with empty stack to q with empty stack. This completes the induction step.

CLAIM 2.31

If x can bring P from p with empty stack to q with empty stack, A_{pq} generates x .

We prove this claim by induction on the number of steps in the computation of P that goes from p to q with empty stacks on input x .

Basis: The computation has 0 steps.

If a computation has 0 steps, it starts and ends at the same state—say, p . So we must show that $A_{pp} \xRightarrow{*} x$. In 0 steps, P only has time to read the empty string, so $x = \epsilon$. By construction, G has the rule $A_{pp} \rightarrow \epsilon$, so the basis is proved.

Induction step: Assume true for computations of length at most k , where $k \geq 0$, and prove true for computations of length $k + 1$.

Suppose that P has a computation wherein x brings p to q with empty stacks in $k + 1$ steps. Either the stack is empty only at the beginning and end of this computation, or it becomes empty elsewhere, too.

In the first case, the symbol that is pushed at the first move must be the same as the symbol that is popped at the last move. Call this symbol t . Let a be the input read in the first move, b be the input read in the last move, r be the state after the first move, and s be the state before the last move. Then $\delta(p, a, \epsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ϵ) , and so rule $A_{pq} \rightarrow aA_{rs}b$ is in G .

Let y be the portion of x without a and b , so $x = ayb$. Input y can bring P from r to s without touching the symbol t that is on the stack and so P can go from r with an empty stack to s with an empty stack on input y . We have removed the first and last steps of the $k + 1$ steps in the original computation on x so the computation on y has $(k + 1) - 2 = k - 1$ steps. Thus the induction hypothesis tells us that $A_{rs} \xRightarrow{*} y$. Hence $A_{pq} \xRightarrow{*} x$.

In the second case, let r be a state where the stack becomes empty other than at the beginning or end of the computation on x . Then the portions of the computation from p to r and from r to q each contain at most k steps. Say that y is the input read during the first portion and z is the input read during the second portion. The induction hypothesis tells us that $A_{pr} \xRightarrow{*} y$ and $A_{rq} \xRightarrow{*} z$. Because rule $A_{pq} \rightarrow A_{pr}A_{rq}$ is in G , $A_{pq} \xRightarrow{*} x$, and the proof is complete.

That completes the proof of Lemma 2.27 and of Theorem 2.20.

We have just proved that pushdown automata recognize the class of context-free languages. This proof allows us to establish a relationship between the regular languages and the context-free languages. Because every regular language is recognized by a finite automaton and every finite automaton is automatically a pushdown automaton that simply ignores its stack, we now know that every regular language is also a context-free language.

COROLLARY 2.32

Every regular language is context free.

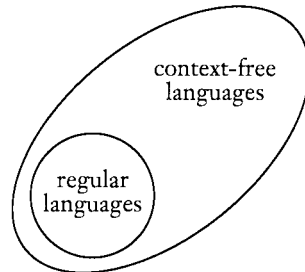


FIGURE 2.33
Relationship of the regular and context-free languages

2.3

NON-CONTEXT-FREE LANGUAGES

In this section we present a technique for proving that certain languages are not context free. Recall that in Section 1.4 we introduced the pumping lemma for showing that certain languages are not regular. Here we present a similar pumping lemma for context-free languages. It states that every context-free language has a special value called the *pumping length* such that all longer strings in the language can be “pumped.” This time the meaning of *pumped* is a bit more complex. It means that the string can be divided into five parts so that the second and the fourth parts may be repeated together any number of times and the resulting string still remains in the language.

THE PUMPING LEMMA FOR CONTEXT-FREE LANGUAGES

THEOREM 2.34

Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p , then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^i xy^i z \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

When s is being divided into $uvxyz$, condition 2 says that either v or y is not the empty string. Otherwise the theorem would be trivially true. Condition 3

states that the pieces v , x , and y together have length at most p . This technical condition sometimes is useful in proving that certain languages are not context free.

PROOF IDEA Let A be a CFL and let G be a CFG that generates it. We must show that any sufficiently long string s in A can be pumped and remain in A . The idea behind this approach is simple.

Let s be a very long string in A . (We make clear later what we mean by “very long.”) Because s is in A , it is derivable from G and so has a parse tree. The parse tree for s must be very tall because s is very long. That is, the parse tree must contain some long path from the start variable at the root of the tree to one of the terminal symbols at a leaf. On this long path some variable symbol R must repeat because of the pigeonhole principle. As the following figure shows, this repetition allows us to replace the subtree under the second occurrence of R with the subtree under the first occurrence of R and still get a legal parse tree. Therefore, we may cut s into five pieces $uvxyz$ as the figure indicates, and we may repeat the second and fourth pieces and obtain a string still in the language. In other words, $uv^i xy^i z$ is in A for any $i \geq 0$.

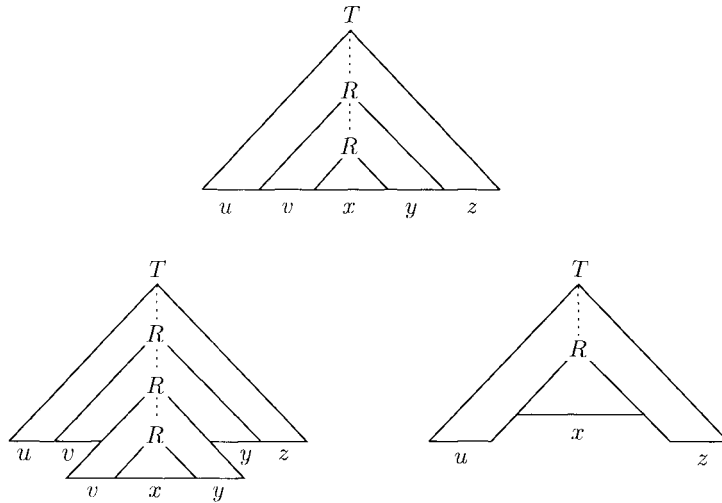


FIGURE 2.35
Surgery on parse trees

Let's now turn to the details to obtain all three conditions of the pumping lemma. We also show how to calculate the pumping length p .

PROOF Let G be a CFG for CFL A . Let b be the maximum number of symbols in the right-hand side of a rule (assume at least 2). In any parse tree using this grammar we know that a node can have no more than b children. In other words, at most b leaves are 1 step from the start variable; at most b^2 leaves are within 2 steps of the start variable; and at most b^h leaves are within h steps of the start variable. So, if the height of the parse tree is at most h , the length of the string generated is at most b^h . Conversely, if a generated string is at least $b^h + 1$ long, each of its parse trees must be at least $h + 1$ high.

Say $|V|$ is the number of variables in G . We set p , the pumping length, to be $b^{|V|+1}$. Now if s is a string in A and its length is p or more, its parse tree must be at least $|V| + 1$ high, because $b^{|V|+1} \geq b^{|V|} + 1$.

To see how to pump any such string s , let τ be one of its parse trees. If s has several parse trees, choose τ to be a parse tree that has the smallest number of nodes. We know that τ must be at least $|V| + 1$ high, so it must contain a path from the root to a leaf of length at least $|V| + 1$. That path has at least $|V| + 2$ nodes; one at a terminal, the others at variables. Hence that path has at least $|V| + 1$ variables. With G having only $|V|$ variables, some variable R appears more than once on that path. For convenience later, we select R to be a variable that repeats among the lowest $|V| + 1$ variables on this path.

We divide s into $uvxyz$ according to Figure 2.35. Each occurrence of R has a subtree under it, generating a part of the string s . The upper occurrence of R has a larger subtree and generates vxy , whereas the lower occurrence generates just x with a smaller subtree. Both of these subtrees are generated by the same variable, so we may substitute one for the other and still obtain a valid parse tree. Replacing the smaller by the larger repeatedly gives parse trees for the strings $uv^i xy^i z$ at each $i > 1$. Replacing the larger by the smaller generates the string uxz . That establishes condition 1 of the lemma. We now turn to conditions 2 and 3.

To get condition 2 we must be sure that both v and y are not ϵ . If they were, the parse tree obtained by substituting the smaller subtree for the larger would have fewer nodes than τ does and would still generate s . This result isn't possible because we had already chosen τ to be a parse tree for s with the smallest number of nodes. That is the reason for selecting τ in this way.

In order to get condition 3 we need to be sure that vxy has length at most p . In the parse tree for s the upper occurrence of R generates vxy . We chose R so that both occurrences fall within the bottom $|V| + 1$ variables on the path, and we chose the longest path in the parse tree, so the subtree where R generates vxy is at most $|V| + 1$ high. A tree of this height can generate a string of length at most $b^{|V|+1} = p$.

For some tips on using the pumping lemma to prove that languages are not context free, review the text preceding Example 1.73 (page 80) where we discuss the related problem of proving nonregularity with the pumping lemma for regular languages.

EXAMPLE 2.36

Use the pumping lemma to show that the language $B = \{a^n b^n c^n \mid n \geq 0\}$ is not context free.

We assume that B is a CFL and obtain a contradiction. Let p be the pumping length for B that is guaranteed to exist by the pumping lemma. Select the string $s = a^p b^p c^p$. Clearly s is a member of B and of length at least p . The pumping lemma states that s can be pumped, but we show that it cannot. In other words, we show that no matter how we divide s into $uvxyz$, one of the three conditions of the lemma is violated.

First, condition 2 stipulates that either v or y is nonempty. Then we consider one of two cases, depending on whether substrings v and y contain more than one type of alphabet symbol.

1. When both v and y contain only one type of alphabet symbol, v does not contain both a's and b's or both b's and c's, and the same holds for y . In this case the string uv^2xy^2z cannot contain equal numbers of a's, b's, and c's. Therefore it cannot be a member of B . That violates condition 1 of the lemma and is thus a contradiction.
2. When either v or y contain more than one type of symbol uv^2xy^2z may contain equal numbers of the three alphabet symbols but not in the correct order. Hence it cannot be a member of B and a contradiction occurs.

One of these cases must occur. Because both cases result in a contradiction, a contradiction is unavoidable. So the assumption that B is a CFL must be false. Thus we have proved that B is not a CFL. \square

EXAMPLE 2.37

Let $C = \{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$. We use the pumping lemma to show that C is not a CFL. This language is similar to language B in Example 2.36, but proving that it is not context free is a bit more complicated.

Assume that C is a CFL and obtain a contradiction. Let p be the pumping length given by the pumping lemma. We use the string $s = a^p b^p c^p$ that we used earlier, but this time we must “pump down” as well as “pump up.” Let $s = uvxyz$ and again consider the two cases that occurred in Example 2.36.

1. When both v and y contain only one type of alphabet symbol, v does not contain both a's and b's or both b's and c's, and the same holds for y . Note that the reasoning used previously in case 1 no longer applies. The reason is that C contains strings with unequal numbers of a's, b's, and c's as long as the numbers are not decreasing. We must analyze the situation more carefully to show that s cannot be pumped. Observe that because v and y contain only one type of alphabet symbol, one of the symbols a, b, or c doesn't appear in v or y . We further subdivide this case into three subcases according to which symbol does not appear.

- a. *The a's do not appear.* Then we try pumping down to obtain the string $uv^0xy^0z = uxz$. That contains the same number of a's as s does, but it contains fewer b's or fewer c's. Therefore it is not a member of C , and a contradiction occurs.
 - b. *The b's do not appear.* Then either a's or c's must appear in v or y because both can't be the empty string. If a's appear, the string uv^2xy^2z contains more a's than b's, so it is not in C . If c's appear, the string uv^0xy^0z contains more b's than c's, so it is not in C . Either way a contradiction occurs.
 - c. *The c's do not appear.* Then the string uv^2xy^2z contains more a's or more b's than c's, so it is not in C , and a contradiction occurs.
2. When either v or y contain more than one type of symbol, uv^2xy^2z will not contain the symbols in the correct order. Hence it cannot be a member of C , and a contradiction occurs.

Thus we have shown that s cannot be pumped in violation of the pumping lemma and that C is not context free. \square

EXAMPLE 2.38

Let $D = \{ww \mid w \in \{0,1\}^*\}$. Use the pumping lemma to show that D is not a CFL. Assume that D is a CFL and obtain a contradiction. Let p be the pumping length given by the pumping lemma.

This time choosing string s is less obvious. One possibility is the string 0^p10^p1 . It is a member of D and has length greater than p , so it appears to be a good candidate. But this string *can* be pumped by dividing it as follows, so it is not adequate for our purposes.

$$\overbrace{000 \cdots 000}^{0^p 1} \underbrace{0}_u \underbrace{1}_v \underbrace{0}_x \overbrace{000 \cdots 0001}^{0^p 1} \underbrace{1}_y \underbrace{000 \cdots 0001}_z$$

Let's try another candidate for s . Intuitively, the string $0^p1^p0^p1^p$ seems to capture more of the "essence" of the language D than the previous candidate did. In fact, we can show that this string does work, as follows.

We show that the string $s = 0^p1^p0^p1^p$ cannot be pumped. This time we use condition 3 of the pumping lemma to restrict the way that s can be divided. It says that we can pump s by dividing $s = uvxyz$, where $|vxy| \leq p$.

First, we show that the substring vxy must straddle the midpoint of s . Otherwise, if the substring occurs only in the first half of s , pumping s up to uv^2xy^2z moves a 1 into the first position of the second half, and so it cannot be of the form ww . Similarly, if vxy occurs in the second half of s , pumping s up to uv^2xy^2z moves a 0 into the last position of the first half, and so it cannot be of the form ww .

But if the substring vxy straddles the midpoint of s , when we try to pump s down to uxz it has the form $0^p1^i0^j1^p$, where i and j cannot both be p . This string is not of the form ww . Thus s cannot be pumped, and D is not a CFL. \square

EXERCISES

- 2.1 Recall the CFG G_4 that we gave in Example 2.4. For convenience, let's rename its variables with single letters as follows.

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T \times F \mid F \\ F &\rightarrow (E) \mid a \end{aligned}$$

Give parse trees and derivations for each string.

- | | |
|----------|------------|
| a. a | c. $a+a+a$ |
| b. $a+a$ | d. $((a))$ |

- 2.2 a. Use the languages $A = \{a^m b^n c^n \mid m, n \geq 0\}$ and $B = \{a^n b^n c^m \mid m, n \geq 0\}$ together with Example 2.36 to show that the class of context-free languages is not closed under intersection.
- b. Use part (a) and DeMorgan's law (Theorem 0.20) to show that the class of context-free languages is not closed under complementation.

- ^A2.3 Answer each part for the following context-free grammar G .

$$\begin{aligned} R &\rightarrow XRX \mid S \\ S &\rightarrow aTb \mid bTa \\ T &\rightarrow XTX \mid X \mid \epsilon \\ X &\rightarrow a \mid b \end{aligned}$$

- | | |
|--|--|
| a. What are the variables of G ? | i. True or False: $T \xRightarrow{*} T$. |
| b. What are the terminals of G ? | j. True or False: $XXX \xRightarrow{*} aba$. |
| c. Which is the start variable of G ? | k. True or False: $X \xRightarrow{*} aba$. |
| d. Give three strings in $L(G)$. | l. True or False: $T \xRightarrow{*} XX$. |
| e. Give three strings <i>not</i> in $L(G)$. | m. True or False: $T \xRightarrow{*} XXX$. |
| f. True or False: $T \Rightarrow aba$. | n. True or False: $S \xRightarrow{*} \epsilon$. |
| g. True or False: $T \xRightarrow{*} aba$. | o. Give a description in English of $L(G)$. |
| h. True or False: $T \Rightarrow T$. | |
- 2.4 Give context-free grammars that generate the following languages. In all parts the alphabet Σ is $\{0,1\}$.
- ^Aa. $\{w \mid w \text{ contains at least three 1s}\}$
- b. $\{w \mid w \text{ starts and ends with the same symbol}\}$
- c. $\{w \mid \text{the length of } w \text{ is odd}\}$
- ^Ad. $\{w \mid \text{the length of } w \text{ is odd and its middle symbol is a 0}\}$
- e. $\{w \mid w = w^R, \text{ that is, } w \text{ is a palindrome}\}$
- f. The empty set

- 2.5 Give informal descriptions and state diagrams of pushdown automata for the languages in Exercise 2.4.
- 2.6 Give context-free grammars generating the following languages.
- ^Aa. The set of strings over the alphabet $\{a, b\}$ with more a's than b's
 - b. The complement of the language $\{a^n b^n \mid n \geq 0\}$
 - ^Ac. $\{w \# x \mid w^R \text{ is a substring of } x \text{ for } w, x \in \{0, 1\}^*\}$
 - d. $\{x_1 \# x_2 \# \cdots \# x_k \mid k \geq 1, \text{ each } x_i \in \{a, b\}^*, \text{ and for some } i \text{ and } j, x_i = x_j^R\}$
- ^A2.7 Give informal English descriptions of PDAs for the languages in Exercise 2.6.
- ^A2.8 Show that the string `the girl touches the boy with the flower` has two different leftmost derivations in grammar G_2 on page 101. Describe in English the two different meanings of this sentence.
- 2.9 Give a context-free grammar that generates the language

$$A = \{a^i b^j c^k \mid i = j \text{ or } j = k \text{ where } i, j, k \geq 0\}.$$

Is your grammar ambiguous? Why or why not?

- 2.10 Give an informal description of a pushdown automaton that recognizes the language A in Exercise 2.9.
- 2.11 Convert the CFG G_4 given in Exercise 2.1 to an equivalent PDA, using the procedure given in Theorem 2.20.
- 2.12 Convert the CFG G given in Exercise 2.3 to an equivalent PDA, using the procedure given in Theorem 2.20.
- 2.13 Let $G = (V, \Sigma, R, S)$ be the following grammar. $V = \{S, T, U\}$; $\Sigma = \{0, \#\}$; and R is the set of rules:

$$\begin{aligned} S &\rightarrow TT \mid U \\ T &\rightarrow 0T \mid T0 \mid \# \\ U &\rightarrow 0U00 \mid \# \end{aligned}$$

- a. Describe $L(G)$ in English.
 - b. Prove that $L(G)$ is not regular.
- 2.14 Convert the following CFG into an equivalent CFG in Chomsky normal form, using the procedure given in Theorem 2.9.

$$\begin{aligned} A &\rightarrow BAB \mid B \mid \varepsilon \\ B &\rightarrow 00 \mid \varepsilon \end{aligned}$$

- 2.15 Give a counterexample to show that the following construction fails to prove that the class of context-free languages is closed under star. Let A be a CFL that is generated by the CFG $G = (V, \Sigma, R, S)$. Add the new rule $S \rightarrow SS$ and call the resulting grammar G' . This grammar is supposed to generate A^* .
- 2.16 Show that the class of context-free languages is closed under the regular operations, union, concatenation, and star.
- 2.17 Use the results of Problem 2.16 to give another proof that every regular language is context free, by showing how to convert a regular expression directly to an equivalent context-free grammar.

PROBLEMS

- ^A2.18 a. Let C be a context-free language and R be a regular language. Prove that the language $C \cap R$ is context free.
 b. Use part (a) to show that the language $A = \{w \mid w \in \{a, b, c\}^* \text{ and contains equal numbers of } a\text{'s, } b\text{'s, and } c\text{'s}\}$ is not a CFL.

*2.19 Let CFG G be

$$\begin{aligned} S &\rightarrow aSb \mid bY \mid Ya \\ Y &\rightarrow bY \mid aY \mid \varepsilon \end{aligned}$$

Give a simple description of $L(G)$ in English. Use that description to give a CFG for $\overline{L(G)}$, the complement of $L(G)$.

- 2.20 Let $A/B = \{w \mid wx \in A \text{ for some } x \in B\}$. Show that, if A is context free and B is regular, then A/B is context free.
- *2.21 Let $\Sigma = \{a, b\}$. Give a CFG generating the language of strings with twice as many a 's as b 's. Prove that your grammar is correct.
- *2.22 Let $C = \{x\#y \mid x, y \in \{0, 1\}^* \text{ and } x \neq y\}$. Show that C is a context-free language.
- *2.23 Let $D = \{xy \mid x, y \in \{0, 1\}^* \text{ and } |x| = |y| \text{ but } x \neq y\}$. Show that D is a context-free language.
- *2.24 Let $E = \{a^i b^j \mid i \neq j \text{ and } 2i \neq j\}$. Show that E is a context-free language.
- 2.25 For any language A , let $SUFFIX(A) = \{v \mid uv \in A \text{ for some string } u\}$. Show that the class of context-free languages is closed under the *SUFFIX* operation.
- 2.26 Show that, if G is a CFG in Chomsky normal form, then for any string $w \in L(G)$ of length $n \geq 1$, exactly $2n - 1$ steps are required for any derivation of w .
- *2.27 Let $G = (V, \Sigma, R, \langle \text{STMT} \rangle)$ be the following grammar.

$$\begin{aligned} \langle \text{STMT} \rangle &\rightarrow \langle \text{ASSIGN} \rangle \mid \langle \text{IF-THEN} \rangle \mid \langle \text{IF-THEN-ELSE} \rangle \\ \langle \text{IF-THEN} \rangle &\rightarrow \text{if condition then } \langle \text{STMT} \rangle \\ \langle \text{IF-THEN-ELSE} \rangle &\rightarrow \text{if condition then } \langle \text{STMT} \rangle \text{ else } \langle \text{STMT} \rangle \\ \langle \text{ASSIGN} \rangle &\rightarrow a:=1 \end{aligned}$$

$$\Sigma = \{\text{if, condition, then, else, } a:=1\}.$$

$$V = \{\langle \text{STMT} \rangle, \langle \text{IF-THEN} \rangle, \langle \text{IF-THEN-ELSE} \rangle, \langle \text{ASSIGN} \rangle\}$$

G is a natural-looking grammar for a fragment of a programming language, but G is ambiguous.

- a. Show that G is ambiguous.
 b. Give a new unambiguous grammar for the same language.
- *2.28 Give unambiguous CFGs for the following languages.
- a. $\{w \mid \text{in every prefix of } w \text{ the number of } a\text{'s is at least the number of } b\text{'s}\}$
 b. $\{w \mid \text{the number of } a\text{'s and } b\text{'s in } w \text{ are equal}\}$
 c. $\{w \mid \text{the number of } a\text{'s is at least the number of } b\text{'s}\}$
- *2.29 Show that the language A in Exercise 2.9 is inherently ambiguous.

- 2.30 Use the pumping lemma to show that the following languages are not context free.
- a. $\{0^n 1^n 0^n 1^n \mid n \geq 0\}$
 - ^Ab. $\{0^n \# 0^{2n} \# 0^{3n} \mid n \geq 0\}$
 - ^Ac. $\{w \# t \mid w \text{ is a substring of } t, \text{ where } w, t \in \{a, b\}^*\}$
 - d. $\{t_1 \# t_2 \# \cdots \# t_k \mid k \geq 2, \text{ each } t_i \in \{a, b\}^*, \text{ and } t_i = t_j \text{ for some } i \neq j\}$
- 2.31 Let B be the language of all palindromes over $\{0, 1\}$ containing an equal number of 0s and 1s. Show that B is not context free.
- 2.32 Let $\Sigma = \{1, 2, 3, 4\}$ and $C = \{w \in \Sigma^* \mid \text{in } w, \text{ the number of 1s equals the number of 2s, and the number of 3s equals the number of 4s}\}$. Show that C is not context free.
- *2.33 Show that $F = \{a^i b^j \mid i = kj \text{ for some positive integer } k\}$ is not context free.
- 2.34 Consider the language $B = L(G)$, where G is the grammar given in Exercise 2.13. The pumping lemma for context-free languages, Theorem 2.34, states the existence of a pumping length p for B . What is the minimum value of p that works in the pumping lemma? Justify your answer.
- 2.35 Let G be a CFG in Chomsky normal form that contains b variables. Show that, if G generates some string with a derivation having at least 2^b steps, $L(G)$ is infinite.
- 2.36 Give an example of a language that is not context free but that acts like a CFL in the pumping lemma. Prove that your example works. (See the analogous example for regular languages in Problem 1.54.)
- *2.37 Prove the following stronger form of the pumping lemma, wherein *both* pieces v and y must be nonempty when the string s is broken up.
- If A is a context-free language, then there is a number k where, if s is any string in A of length at least k , then s may be divided into five pieces, $s = uvxyz$, satisfying the conditions:
- a. for each $i \geq 0$, $uv^i xy^i z \in A$,
 - b. $v \neq \epsilon$ and $y \neq \epsilon$, and
 - c. $|vxy| \leq k$.
- ^A2.38 Refer to Problem 1.41 for the definition of the perfect shuffle operation. Show that the class of context-free languages is not closed under perfect shuffle.
- 2.39 Refer to Problem 1.42 for the definition of the shuffle operation. Show that the class of context-free languages is not closed under shuffle.
- *2.40 Say that a language is **prefix-closed** if the prefix of any string in the language is also in the language. Let C be an infinite, prefix-closed, context-free language. Show that C contains an infinite regular subset.
- *2.41 Read the definitions of $NOPREFIX(A)$ and $NOEXTEND(A)$ in Problem 1.40.
- a. Show that the class of CFLs is not closed under $NOPREFIX$ operation.
 - b. Show that the class of CFLs is not closed under $NOEXTEND$ operation.
- *2.42 Let $\Sigma = \{1, \#\}$ and $Y = \{w \mid w = t_1 \# t_2 \# \cdots \# t_k \text{ for } k \geq 0, \text{ each } t_i \in 1^*, \text{ and } t_i \neq t_j \text{ whenever } i \neq j\}$. Prove that Y is not context free.

- *2.45 Let $A = \{wtw^{\mathcal{R}} \mid w, t \in \{0, 1\}^* \text{ and } |w| = |t|\}$. Prove that A is not a context-free language.

SELECTED SOLUTIONS

- 2.3 (a) R, X, S, T ; (b) a, b ; (c) R ; (d) Three strings in G are ab, ba , and aab ;
 (e) Three strings not in G are a, b , and ϵ ; (f) False; (g) True; (h) False;
 (i) True; (j) True; (k) False; (l) True; (m) True; (n) False; (o) $L(G)$ consists
 of all strings over a and b that are not palindromes.
- 2.4 (a) $S \rightarrow R1R1R1R$ (d) $S \rightarrow 0 \mid 0S0 \mid 0S1 \mid 1S0 \mid 1S1$
 $R \rightarrow 0R \mid 1R \mid \epsilon$
- 2.6 (a) $S \rightarrow TaT$ (c) $S \rightarrow TX$
 $T \rightarrow TT \mid aTb \mid bTa \mid a \mid \epsilon$ $T \rightarrow 0T0 \mid 1T1 \mid \#X$
 T generates all strings with at least as many a 's as b 's, and S forces an extra a . $X \rightarrow 0X \mid 1X \mid \epsilon$
- 2.7 (a) The PDA uses its stack to count the number of a 's minus the number of b 's. It enters an accepting state whenever this count is positive. In more detail, it operates as follows. The PDA scans across the input. If it sees a b and its top stack symbol is an a , it pops the stack. Similarly, if it scans an a and its top stack symbol is a b , it pops the stack. In all other cases, it pushes the input symbol onto the stack. After the PDA finishes the input, if a is on top of the stack, it accepts. Otherwise it rejects.
- (c) The PDA scans across the input string and pushes every symbol it reads until it reads a $\#$. If $\#$ is never encountered, it rejects. Then, the PDA skips over part of the input, nondeterministically deciding when to stop skipping. At that point, it compares the next input symbols with the symbols it pops off the stack. At any disagreement, or if the input finishes while the stack is nonempty, this branch of the computation rejects. If the stack becomes empty, the machine reads the rest of the input and accepts.

2.8 Here is one derivation:

$\langle \text{SENTENCE} \rangle \Rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \Rightarrow$
 $\langle \text{CMPLX-NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \Rightarrow$
 $\langle \text{CMPLX-NOUN} \rangle \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle \Rightarrow$
 $\langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle \Rightarrow$
 The boy $\langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \langle \text{PREP-PHRASE} \rangle \Rightarrow$
 The boy $\langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \Rightarrow$
 The boy touches $\langle \text{NOUN-PHRASE} \rangle \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \Rightarrow$
 The boy touches $\langle \text{CMPLX-NOUN} \rangle \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \Rightarrow$
 The boy touches $\langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \Rightarrow$
 The boy touches the girl with $\langle \text{CMPLX-NOUN} \rangle \Rightarrow$
 The boy touches the girl with $\langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \Rightarrow$
 The boy touches the girl with the flower

Here is another derivation:

$\langle \text{SENTENCE} \rangle \Rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \Rightarrow$
 $\langle \text{CMPLX-NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \Rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \Rightarrow$
 The boy $\langle \text{VERB-PHRASE} \rangle \Rightarrow$ The boy $\langle \text{CMPLX-VERB} \rangle \Rightarrow$
 The boy $\langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \Rightarrow$
 The boy touches $\langle \text{NOUN-PHRASE} \rangle \Rightarrow$
 The boy touches $\langle \text{CMPLX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \Rightarrow$
 The boy touches $\langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \Rightarrow$
 The boy touches the girl $\langle \text{PREP-PHRASE} \rangle \Rightarrow$
 The boy touches the girl $\langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \Rightarrow$
 The boy touches the girl with $\langle \text{CMPLX-NOUN} \rangle \Rightarrow$
 The boy touches the girl with $\langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \Rightarrow$
 The boy touches the girl with the flower

Each of these derivations corresponds to a different English meaning. In the first derivation, the sentence means that the boy used the flower to touch the girl. In the second derivation, the girl is holding the flower when the boy touches her.

2.18 (a) Let C be a context-free language and R be a regular language. Let P be the PDA that recognizes C , and D be the DFA that recognizes R . If Q is the set of states of P and Q' is the set of states of D , we construct a PDA P' that recognizes $C \cap R$ with the set of states $Q \times Q'$. P' will do what P does and also keep track of the states of D . It accepts a string w if and only if it stops at a state $q \in F_P \times F_D$, where F_P is the set of accept states of P and F_D is the set of accept states of D . Since $C \cap R$ is recognized by P' , it is context free.

(b) Let R be the regular language $a^*b^*c^*$. If A were a CFL then $A \cap R$ would be a CFL by part (a). However, $A \cap R = \{a^n b^n c^n \mid n \geq 0\}$, and Example 2.36 proves that $A \cap R$ is not context free. Thus A is not a CFL.

2.30 (b) Let $B = \{0^n \# 0^{2n} \# 0^{3n} \mid n \geq 0\}$. Let p be the pumping length given by the pumping lemma. Let $s = 0^p \# 0^{2p} \# 0^{3p}$. We show that $s = uvxyz$ cannot be pumped.

Neither v nor y can contain $\#$, otherwise xv^2wy^2z contains more than two $\#$ s. Therefore, if we divide s into three segments by $\#$'s: 0^p , 0^{2p} , and 0^{3p} , at least one of the segments is not contained within either v or y . Hence xv^2wy^2z is not in B because the 1 : 2 : 3 length ratio of the segments is not maintained.

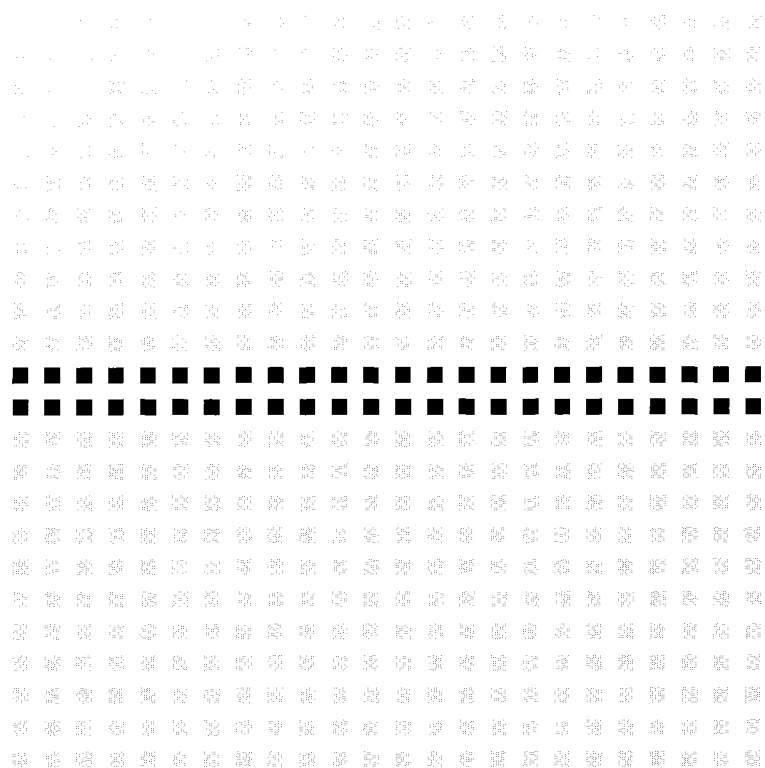
(c) Let $C = \{w\#t \mid w \text{ is a substring of } t, \text{ where } w, t \in \{a, b\}^*\}$. Let p be the pumping length given by the pumping lemma. Let $s = a^p b^p \# a^p b^p$. We show that the string $s = uvxyz$ cannot be pumped.

Neither v nor y can contain $\#$, otherwise uv^0xy^0z does not contain $\#$ and therefore is not in C . If both v and y are nonempty and occur on the left-hand side of the $\#$, the string uv^2xy^2z cannot be in C because it is longer on the left-hand side of the $\#$. Similarly, if both strings occur on the right-hand side of the $\#$, the string uv^0xy^0z cannot be in C because it is again longer on the left-hand side of the $\#$. If only one of v and y is nonempty (both cannot be nonempty), treat them as if both occurred on the same side of the $\#$ as above.

The only remaining case is where both v and y are nonempty and straddle the $\#$. But then v consists of b 's and y consists of a 's because of the third pumping lemma condition $|vxy| \leq p$. Hence, uv^2xy^2z contains more b 's on the left-hand side of the $\#$, so it cannot be a member of C .

- 2.38** Let A be the language $\{0^k 1^k \mid k \geq 0\}$ and let B be the language $\{a^k b^{3k} \mid k \geq 0\}$. The perfect shuffle of A and B is the language $C = \{(0a)^k (0b)^k (1b)^{2k} \mid k \geq 0\}$. Languages A and B are easily seen to be CFLs, but C is not a CFL, as follows. If C were a CFL, let p be the pumping length given by the pumping lemma, and let s be the string $(0a)^p (0b)^p (1b)^{2p}$. Because s is longer than p and $s \in C$, we can divide $s = uvxyz$ satisfying the pumping lemma's three conditions. Strings in C contain twice as many 1s as a 's. In order for uv^2xy^2z to have that property, the string vxy must contain both 1s and a 's. But that is impossible, because they are separated by $2p$ symbols yet the third condition says that $|vxy| \leq p$. Hence C is not context free.

PART TWO



C O M P U T A B I L I T Y T H E O R Y

