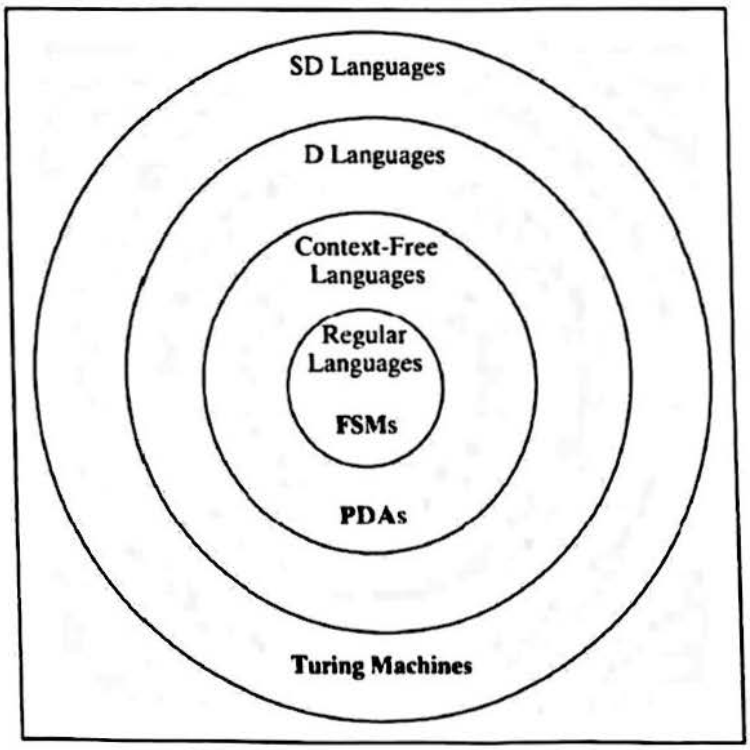
PART II

FINITE STATE MACHINES AND REGULAR LANGUAGES

In this section, we begin our exploration of the language hierarchy. We will start in the inner circle, which corresponds to the class of regular languages.

We will explore three techniques, which we will prove are equivalent, for defining the regular languages:

- Finite state machines.
- Regular languages.
- Regular grammars.



CHAPTER 5

Finite State Machines

The simplest and most efficient computational device that we will consider is the finite state machine (or FSM).

EXAMPLE 5.1 A Vending Machine

Consider the problem of deciding when to dispense a drink from a vending machine. To simplify the problem a bit, we'll pretend that it were still possible to buy a drink for $.25 and we will assume that vending machines do not take pennies. The solution that we will present for this problem can straightforwardly be extended to modern, high-priced machines.

The vending machine controller will receive a sequence of inputs, each of which corresponds to one of the following events:

- A coin is deposited into the machine. We can use the symbols N (for nickel), D (for dime), and Q (for quarter) to represent these events.
- The coin return button is pushed. We can use the symbol R (for return) to represent this event.
- A drink button is pushed and a drink is dispensed. We can use the symbol S (for soda) for this event.

After any finite sequence of inputs, the controller will be in either:

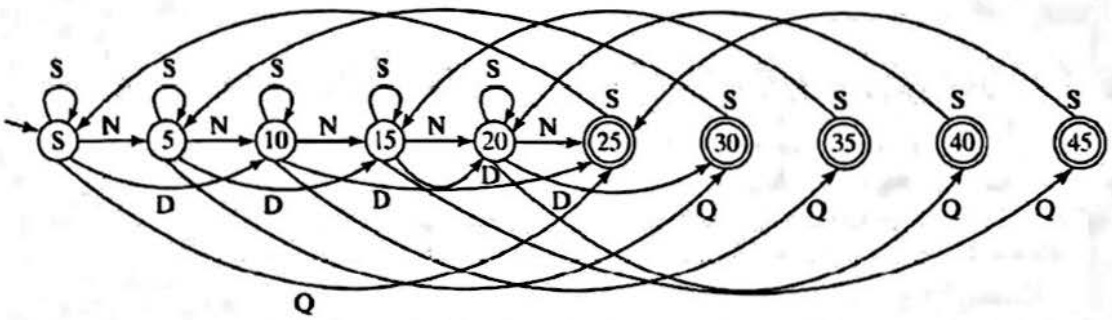
- A dispensing state, in which it is willing to dispense a drink if a drink button is pushed.
- A nondispensing state, in which not enough money has been inserted into the machine.

While there is no bound on the length of the input sequence that a drink machine may see in a week, there is only a finite amount of history that its controller must remember in order to do its job. It needs only to be able to answer

the question, "Has enough money been inserted, since the last time a drink was dispensed, to purchase the next drink?" It is of course possible for someone to keep inserting money without ever pushing a dispense-drink button. But we can design a controller that will simply reject any money that comes in after the amount required to buy a drink has been recorded and before a drink has actually been dispensed. We will however assume that our goal is to design a customer-friendly drink machine. For example, the thirsty customer may have only dimes. So we'll build a machine that will accept up to \$.45. If more than the necessary \$.25 is inserted before a dispensing button is pushed, our machine will remember the difference and leave a "credit" in the machine. So, for example, if a customer inserts three dimes and then asks for drink, the machine will remember the balance of \$.05.

Notice that the drink controller does not need to remember the actual sequence of coins that it has received. It need only remember the total *value* of the coins that have been inserted since the last drink was dispensed.

The drink controller that we have just described needs 10 states, corresponding to the possible values of the credit that the customer has in the machine: 0, 5, 10, 15, 20, 25, 30, 35, 40, and 45 cents. The main structure of the controller is then:



The state that is labeled *S* is the start state. Transitions from one state to the next are shown as arrows and labeled with the event that causes them to take place. As coins are deposited, the controller's state changes to reflect the amount of money that has been deposited. When the drink button is pushed (indicated as *S* in the diagram) and the customer has a credit of less than \$.25, nothing happens. The machine's state does not change. If the drink button is pushed and the customer has a credit of \$.25 or more, the credit is decremented by \$.25 and a drink is dispensed. The drink-dispensing states, namely those that correspond to "enough money", can be thought of as goal or accepting states. We have shown them in the diagram with double circles.

Not all of the required transitions have been shown in the diagram. It would be too difficult to read. We must add to the ones shown all of the following:

- From each of the accepting states, a transition back to itself labeled with each coin value. These transitions correspond to our decision to reject additional coins once the machine has been fed the price of a drink.

EXAMPLE 5.1 *(Continued)*

- From each state, a transition back to the start state labeled *R*. These transitions will be taken whenever the customer pushes the coin return button. They correspond to the machine returning all of the money that it has accumulated since the last drink was dispensed.

The drink controller that we have just described is an example of a finite state machine. We can think of it as a device to solve a problem (dispense drinks). Or we can think of it as a device to recognize a language (the "enough money" language that consists of the set of strings, such as NDD, that drive the machine to an accepting state in which a drink can be dispensed). In most of the rest of this chapter, we will take the language recognition perspective. But it does also make sense to imagine a finite state machine that actually acts in the world (for example, by outputting a coin or a drink). We will return to that idea in Section 5.9.

> The history of finite state machines substantially predates modern computers. (P. 1)

5.1 Deterministic Finite State Machines

A ***finite state machine*** (or FSM) is a computational device whose input is a string and whose output is one of two values that we can call *Accept* and *Reject*. FSMs are also sometimes called finite state automata or FSAs.

If M is an FSM, an input string is fed to M one character at a time, left to right. Each time it receives a character, M considers its current state and the new character and chooses a next state. One or more of M's states may be marked as accepting states. If M runs out of input and is in an accepting state, it accepts. If, however, M runs out of input and is not in an accepting state, it rejects. The number of steps that M executes on input w is exactly equal to $|w|$, so M always halts and either accepts or rejects.

We begin by defining the class of FSMs whose behavior is deterministic. In such machines, there is always exactly one move that can be made at each step; that move is determined by the current state and the next input character. In Section 5.4, we will relax this restriction and introduce nondeterministic FSMs (also called NDFSMs), in which there may, at various points in the computation, be more than one move from which the machine may choose. We will continue to use the term FSM to include both deterministic and nondeterministic FSMs.

> A telephone switching circuit can easily be modeled as a DFSM.

Formally, a ***deterministic FSM*** (or ***DFSM***) M is a quintuple $(K, \Sigma, \delta, s, A)$, where:

- K is a finite set of states,
- Σ is the input alphabet,

- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states, and
- δ is the transition function. It maps from:

$$\underset{\text{state}}{K} \times \underset{\text{input symbol}}{\Sigma} \text{ to } \underset{\text{state}}{K}.$$

A ***configuration*** of a DFSM M is an element of $K \times \Sigma^*$. Think of it as a snapshot of M. It captures the two things that can make a difference to M's future behavior:

- Its current state.
- The input that is still left to read.

The ***initial configuration*** of a DFSM M, on input w, is (s_M, w), where s_M is the start state of M. (We can use the subscript notation to refer to components of a machine M's definition, although, when the context makes it clear what machine we are talking about, we may omit the subscript.)

The transition function δ defines the operation of a DFSM M one step at a time. We can use it to define the sequence of configurations that M will enter. We start by defining the relation *yields-in-one-step*, written $|-_M$. *Yields-in-one-step* relates *configuration*$_1$ to *configuration*$_2$ iff M can move from *configuration*$_1$ to *configuration*$_2$ in one step. Let c be any element of Σ and let w be any element of Σ^*. Then,

$$(q_1, cw) |-_M (q_2, w) \text{ iff } ((q_1, c), q_2) \in \delta.$$

We can now define the relation *yields*, written $|-_M^*$ to be the reflexive, transitive closure of $|-_M$. So configuration C_1 yields configuration C_2 iff M can go from C_1 to C_2 in zero or more steps. In this case, we will write:

$$C_1 |-_M^* C_2.$$

A ***computation*** by M is a finite sequence of configurations $C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that:

- C_0 is an initial configuration,
- C_n is of the form (q, ε), for some state $q \in K_M$ (i.e., the entire input string has been read), and
- $C_0 |-_M C_1 |-_M C_2 |-_M \ldots |-_M C_n$.

Let w be an element of Σ^*. Then we will say that:

- M ***accepts*** w iff $(s, w) |-_M^* (q, \varepsilon)$, for some $q \in A_M$. Any configuration (q, ε), for some $q \in A_M$, is called an ***accepting configuration*** of M.
- M ***rejects*** w iff $(s, w) |-_M^* (q, \varepsilon)$, for some $q \notin A_M$. Any configuration (q, ε), for some $q \notin A_M$, is called an ***rejecting configuration*** of M.

M halts whenever it enters either an accepting or a rejecting configuration. It will do so immediately after reading the last character of its input.

The ***language accepted by*** M, denoted $L(M)$, is the set of all strings accepted by M.

EXAMPLE 5.2 A Simple Language of a's and b's

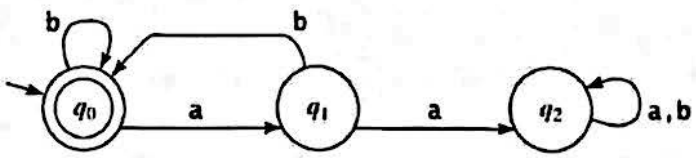
Let $L = \{w \in \{a, b\}^* : \text{every a is immediately followed by a b}\}$. L can be accepted by the DFSM $M = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_0\})$, where:

$$\begin{aligned}\delta = \{&((q_0, a), q_1),\\ &((q_0, b), q_0),\\ &((q_1, a), q_2),\\ &((q_1, b), q_0),\\ &((q_2, a), q_2),\\ &((q_2, b), q_2))\}.\end{aligned}$$

The tuple notation that we have just used for δ is quite hard to read. We will generally find it useful to draw δ as a transition diagram instead. When we do that, we will use two conventions:

1. The start state will be indicated with an unlabeled arrow pointing into it.
2. The accepting states will be indicated with double circles.

With those conventions, a DFSM can be completely specified by a transition diagram. So M is:



We will use the notation a, b as a shorthand for two transitions, one labeled a and one labeled b.

As an example of M's operation, consider the input string abbabab. M's computation is the sequence of configurations: $(q_0, \text{abbabab})$, (q_1, bbabab), (q_0, babab), (q_0, abab), (q_1, bab), (q_0, ab), (q_1, b), (q_0, ε). Since q_0 is an accepting state, M accepts.

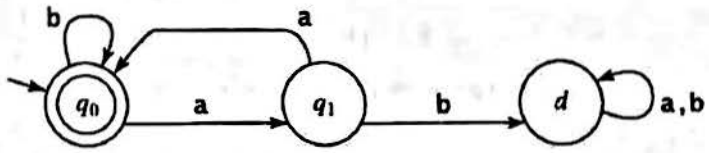
If we look at the three states in M, the machine that we just built, we see that they are of three different sorts:

1. State q_0 is an accepting state. Every string that drives M to state q_0 is in L.
2. State q_1 is not an accepting state. But every string that drives M to state q_1 could turn out to be in L if it is followed by an appropriate continuation string, in this case, one that starts with a b.

3. State q_2 is what we will call a ***dead state***. Once M enters state q_2, it will never leave. State q_2 is not an accepting state, so any string that drives M to state q_2 has already been determined not to be in L, *no matter what comes next*. We will often name our dead states d.

EXAMPLE 5.3 Even Length Regions of a's

Let $L = \{w \in \{a, b\}^* : \text{every a region in } w \text{ is of even length}\}$. L can be accepted by the DFSM M:



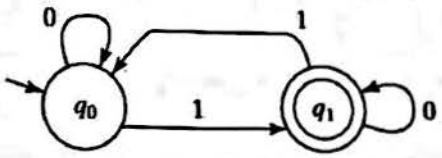
If M sees a b in state q_1, then there has been an a region whose length is odd. So, no matter what happens next, M must reject. So it goes to the dead state d.

> A useful way to prototype a complex system is as a finite state machine. See P. 4 for one example: the controller for a soccer-playing robot.

Because objects of other data types are encoded in computer memories as binary strings, it is important to be able to check key properties of such strings.

EXAMPLE 5.4 Checking for Odd Parity

Let $L = \{w \in \{0, 1\}^* : w \text{ has odd parity}\}$. A binary string has odd parity iff the number of 1's in it is odd. So L can be accepted by the DFSM M:



One of the most important properties of finite state machines is that they are guaranteed to halt on any input string of finite length. While this may seem obvious, it is worth noting since, as we'll see later, more powerful computational models may not share this property.

THEOREM 5.1 DFSMs Halt

Theorem: Every DFSM M, on input w, halts after $|w|$ steps.

Proof: On input w, M executes some computation $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \ldots \vdash_M C_n$, where C_0 is an initial configuration and C_n is of the form (q, ε), for some state $q \in K_M$. C_n is either an accepting or a rejecting configuration, so M will halt when it reaches C_n. Each step in the computation consumes one character of w. So $n = |w|$. Thus M will halt after $|w|$ steps.

5.2 The Regular Languages

We have now built DFSMs to accept four languages:

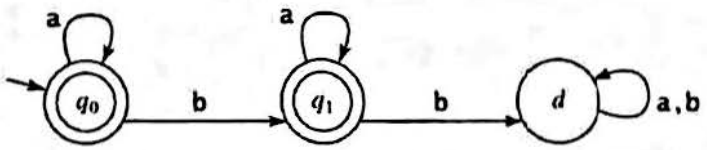
- "enough money to buy a drink",
- $\{w \in \{a, b\}^* :$ every a is immediately followed by a b$\}$,
- $\{w \in \{a, b\}^* :$ every a region in w is of even length$\}$, and
- binary strings with odd parity.

These four languages are typical of a large class of languages that can be accepted by finite state machines.

We define the set of ***regular languages*** to be exactly those that can be accepted by some DFSM.

EXAMPLE 5.5 No More Than One b

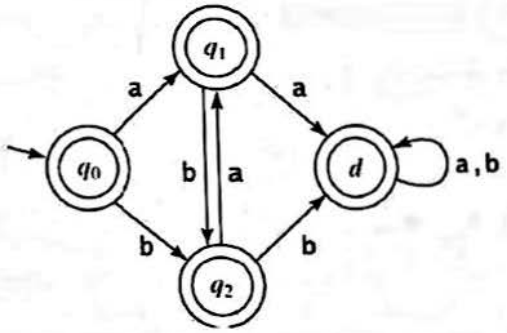
Let $L = \{w \in \{a, b\}^* :$ w contains no more than one b$\}$. L is regular because it can be accepted by the DFSM M:



Any string with more than one b will drive M to the dead state d. All other strings will drive M to either q_0 or q_1, both of which are accepting states.

EXAMPLE 5.6 No Two Consecutive Characters Are the Same

Let $L = \{w \in \{a, b\}^* :$ no two consecutive characters are the same$\}$. L is regular because it can be accepted by the DFSM M:



The start state, q_0, is the only state in which both a and b are legal inputs. M will be in state q_1 whenever the consecutive characters rule has not been violated and the last character it has read was a. At that point, the only legal next character is b. M will be in state q_2 whenever the consecutive characters rule has not been violated and the last character it has read was b. At that point, the only legal next character is a. Any other inputs drive M to d.

Simple languages of a's and b's, like the ones in the last two examples, are useful for practice in designing DFSMs. But the real power of the DFSM model comes from the fact that the languages that arise in many real-world applications are regular.

> The language of universal resource identifiers (URIs), used to describe objects on the World Wide Web, is regular. (I.3.1)

To describe less trivial languages will sometimes require DFSMs that are hard to draw if we include the dead state. In those cases, we will omit it from our diagrams. This doesn't mean that it doesn't exist. δ is a function that must be defined for all (state, input) pairs. It just means that we won't bother to draw the dead state. Instead, our convention will be that if there is no transition specified for some (state, input) pair, then that pair drives the machine to a dead state.

EXAMPLE 5.7 Floating Point Numbers

Let FLOAT = $\{w : w$ is the string representation of a floating point number$\}$. Assume the following syntax for floating point numbers:

- A floating point number is an optional sign, followed by a decimal number, followed by an optional exponent.
- A decimal number may be of the form x or $x.y$, where x and y are nonempty strings of decimal digits.

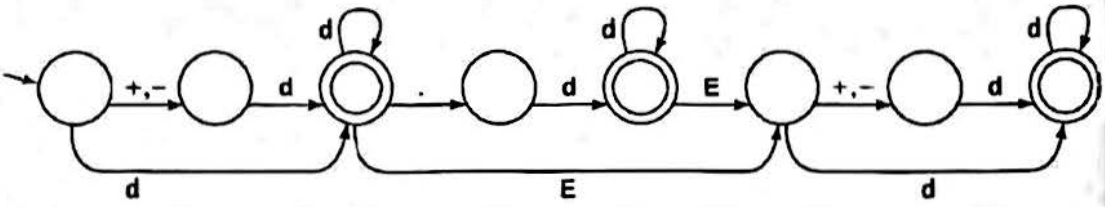
EXAMPLE 5.7 *(Continued)*

- An exponent begins with E and is followed by an optional sign and then an integer.
- An integer is a nonempty string of decimal digits.

So, for example, these strings represent floating point numbers:

+3.0, 3.0, 0.3E1, 0.3E+1, −0.3E+1, −3E8

FLOAT is regular because it can be accepted by the DFSM:



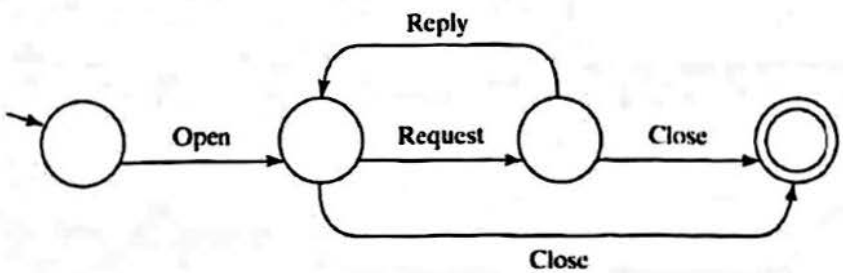
In this diagram, we have used the shorthand d to stand for any one of the decimal digits (0 − 9). And we have omitted the dead state to avoid arrows crossing over each other.

EXAMPLE 5.8 A Simple Communication Protocol

Let L be a language that contains all the legal sequences of messages that can be exchanged between a client and a server using a simple communication protocol. We will actually consider only a very simplified version of such a protocol, but the idea can be extended to a more realistic model.

Let Σ_L = {Open, Request, Reply, Close}. Every string in L begins with Open and ends with Close. In addition, every Request, except possibly the last, must be followed by Reply and no unsolicited Reply's may occur.

L is regular because it can be accepted by the DFSM:



Note that we have again omitted the dead state.

More realistic communication protocols can also be modeled as FSMs. (I.1)

5.3 Designing Deterministic Finite State Machines

Given some language L, how should we go about designing a DFSM to accept L? In general, as in any design task, there is no magic bullet. But there are two related things that it is helpful to think about:

- Imagine any DFSM M that accepts L. As a string w is being read by M, what properties of the part of w that has been seen so far are going to have any bearing on the ultimate answer that M needs to produce? Those are the properties that M needs to record. So, for example, in the "enough money" machine, all that matters is the amount of money since the last drink was dispensed. Which coins came in and the order in which they were deposited make no difference.
- If L is infinite but M has a finite number of states, strings must "cluster". In other words, multiple different strings will all drive M to the same state. Once they have done that, none of their differences matter anymore. If they've driven M to the same state, they share a fate. No matter what comes next, either all of them cause M to accept or all of them cause M to reject. In Section 5.7 we will show that the smallest DFSM for any language L is the one that has exactly one state for every group of initial substrings that share a common fate. For now, however, it helps to think about what those clusters are. We'll do that in our next example.

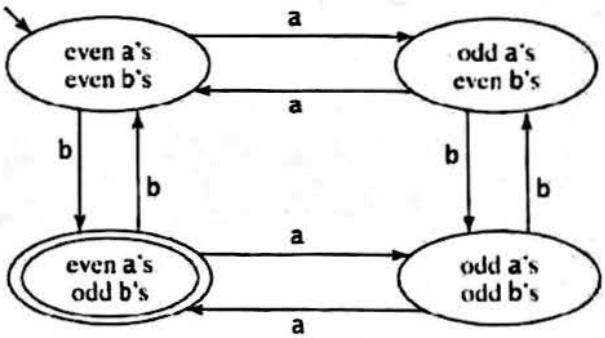
A building security system can be described as a DFSM that sounds an alarm if given an input sequence that signals an intruder. (J.1)

EXAMPLE 5.9 Even a's, Odd b's

Let $L = \{w \in \{a, b\}^* : w$ contains an even number of a's and an odd number of b's$\}$. To design a DFSM M to accept L, we need to decide what history matters. Since M's goal is to separate strings with even a's and odd b's from strings that fail to meet at least one of those requirements, all it needs to remember is whether the count of a's so far is even or odd and whether the count of b's is even or odd. So, since there are two clusters based on the number of a's so far (even and odd) and two clusters based on the number of b's, there are four distinct clusters.

That suggests that we need a four-state DFSM. Often it helps to name the states with a description of the clusters to which they correspond. The following DFSM M accepts L:

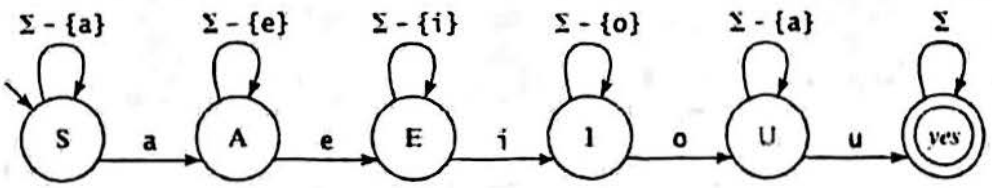
EXAMPLE 5.9 *(Continued)*



Notice that, once we have designed a machine that analyzes an input string with respect to some set of properties we care about, it is relatively easy to build a different machine that accepts strings based on different values of those properties. For example, to change M so that it accepts exactly the strings with both even a's and even b's, all we need to do is to change the accepting state.

EXAMPLE 5.10 All the Vowels in Alphabetical Order

Let $L = \{w \in \{a - z\}^* :$ all five vowels, a, e, i, o, and u, occur in w in alphabetical order$\}$. So L contains words like `abstemious`, `facetious`, and `sacrilegious`. But it does not contain `tenacious`, which does contain all the vowels, but not in the correct order. It is hard to write a clear, elegant program to accept L. But designing a DFSM is simple. The following machine M does the job. In this description of M, let the label "$\Sigma - \{a\}$" mean "all elements of Σ except a" and let the label "Σ" mean "all elements of Σ":

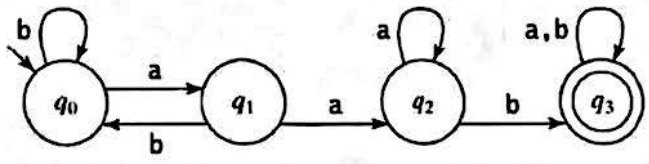


Notice that the state that we have labeled *yes* functions exactly opposite to the way in which the dead state works. If M ever reaches *yes*, it has decided to accept no matter what comes next.

Sometimes an easy way to design an FSM to accept a language L is to begin by designing an FSM to accept the complement of L. Then, as a final step, we swap the accepting and the nonaccepting states.

EXAMPLE 5.11 A Substring that Doesn't Occur

Let $L = \{w \in \{a, b\}^* : w$ does not contain the substring aab$\}$. It is straightforward to design an FSM that looks for the substring aab. So we can begin building a machine to accept L by building the following machine to accept $\neg L$:



Then we can convert this machine into one that accepts L by making states q_0, q_1, and q_2 accepting and state q_3 nonaccepting.

In Section 8.3 we'll show that the regular languages are closed under complement (i.e., the complement of every regular language is also regular). The proof will be by construction and the last step of the construction will be to swap accepting and nonaccepting states, just as we did in the last example.

Sometimes the usefulness of the DFSM model, as we have so far defined it, breaks down before its formal power does. There are some regular languages that seem quite simple when we state them but that can only be accepted by DFSMs of substantial complexity.

EXAMPLE 5.12 The Missing Letter Language

Let $\Sigma = \{a, b, c, d\}$. Let $L_{Missing} = \{w :$ there is a symbol $a_i \in \Sigma$ not appearing in $w\}$. $L_{Missing}$ is regular. We can begin writing out a DFSM M to accept it. We will need the following states:

- The start state: all letters are still missing.

After one character has been read, M could be in any one of:

- a read, so b, c, and d still missing.
- b read, so a, c, and d still missing.
- c read, so a, b, and d still missing.
- d read, so a, b, and c still missing.

After a second character has been read, M could be in any of the previous states or one of:

- a and b read, so c and d still missing.
- a and c read, so b and d still missing.
- and so forth. There are six of these.

EXAMPLE 5.12 *(Continued)*

After a third character has been read, M could be in any of the previous states or one of:

- a and b and c read, so d missing.
- a and b and d read, so c missing.
- a and c and d read, so b missing.
- b and c and d read, so a missing.

After a fourth character has been read, M could be in any of the previous states or:

- All characters read, so nothing is missing.

Every state except the last is an accepting state. M is complicated but it would be possible to write it out. Now imagine that Σ were the entire English alphabet. It would still be possible to write out a DFSM to accept $L_{Missing}$, but it would be so complicated it would be hard to get it right. The DFSM model is no longer very useful.

5.4 Nondeterministic FSMs

To solve the problem that we just encountered in the missing letter example, we will modify our definition of an FSM to allow nondeterminism. Recall our discussion of nondeterminism in Section 4.2. We will now introduce our first specific use of the ideas we discussed there. We'll see that we can easily build a nondeterministic FSM M to accept $L_{Missing}$. Any string in $L_{Missing}$ must be missing at least one letter. We'll design M so that it simply guesses at which letter that is. If there is a missing letter, then at least one of M's guesses will be right and the corresponding path will accept. So M will accept.

5.4.1 What Is a Nondeterministic FSM?

A nondeterministic FSM (or NDFSM) M is a quintuple $(K, \Sigma, \Delta, s, A)$, where:

- K is a finite set of states,
- Σ is an alphabet,
- $s \in K$ is the start state,
- $A \subseteq$ is the set of final states, and
- Δ is the transition relation. It is a finite subset of: $(K \times (\Sigma \cup \{\varepsilon\})) \times K$.

In other words, each element of Δ contains a (state, input symbol or ε) pair, and a new state.

We define configuration, initial configuration, accepting configuration, *yields-in-one-step*, *yields*, and computation analogously to the way that we defined them for DFSMs.

Let w be an element of Σ^*. Then we will say that:

- ***M accepts*** w iff *at least one* of its computations accepts.
- ***M rejects*** w iff none of its computations accepts.

The ***language accepted by*** M, denoted $L(M)$, is the set of all strings accepted by M.

There are two key differences between DFSMs and NDFSMs. In every configuration, a DFSM can make exactly one move. However, because Δ can be an arbitrary relation (that may not also be a function), that is not necessarily true for an NDFSM. Instead:

- An NDFSM M may enter a configuration in which there are still input symbols left to read but from which *no* moves are available. Since any sequence of moves that leads to such a configuration cannot ever reach an accepting configuration, M will simply halt without accepting. This situation is possible because Δ is not a function. So there can be (state, input) pairs for which no next state is defined.
- An NDFSM M may enter a configuration from which *two or more* competing moves are possible. The competition can come from either or both of the following properties of the transition relation of an NDFSM:
  - An NDFSM M may have one or more transitions that are labeled ε, rather than being labeled with a character from Σ. An ε-transition out of state q may (but need not) be followed, without consuming any input, whenever M is in state q. So an ε-transition from a state q competes with all other transitions out of q. One way to think about the usefulness of ε-transitions is that they enable M to guess at the correct path before it actually sees the input. Wrong guesses will generate paths that will fail but that can be ignored.
  - Out of some state q, there may be more than one transition with a given label. These competing transitions give M another way to guess at a correct path.

Consider the fragment, shown in Figure 5.1, of an NDFSM M. If M is in state q_0 and the next input character is an a, then there are three moves that M could make:

1. It can take the ε-transition to q_1 before it reads the next input character,
2. It can read the next input character and take the transition to q_2, or
3. It can read the next input character and take the transition to q_3.

One way to envision the operation of M is as a tree, as shown in Figure 5.2. Each node in the tree corresponds to a configuration of M. Each path from the root corresponds to a sequence of moves that M might make. Each path that leads to a configuration in which the entire input string has been read corresponds to a computation of M.

An alternative is to imagine following all paths through M in parallel. Think of M as being in a *set* of states at each step of its computation. If, when M runs out of input, the set of states that it is in contains at least one accepting state, then M will accept.

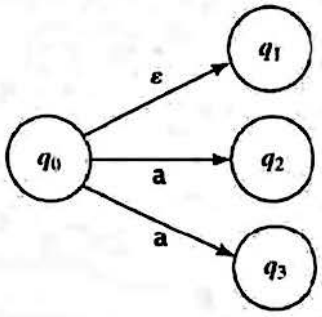


FIGURE 5.1 An NDFSM with two kinds of nondeterminism.

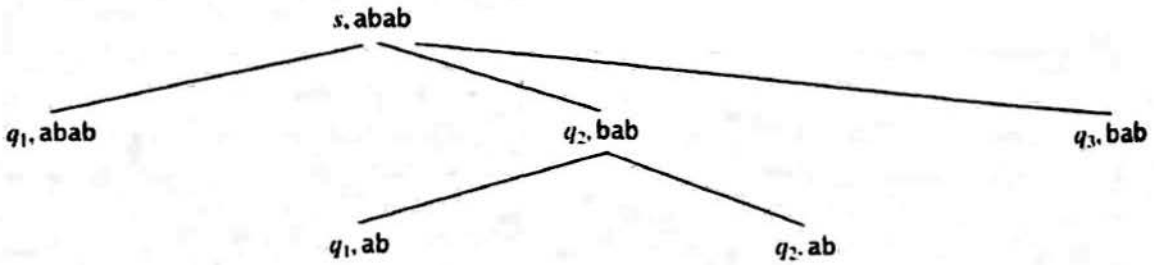
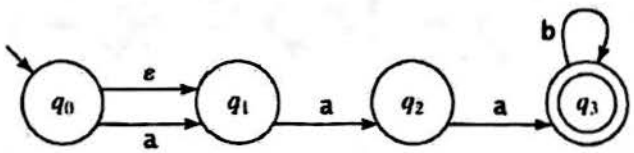


FIGURE 5.2 Viewing nondeterminism as search through a space of computation paths.

EXAMPLE 5.13 An Optional Initial a

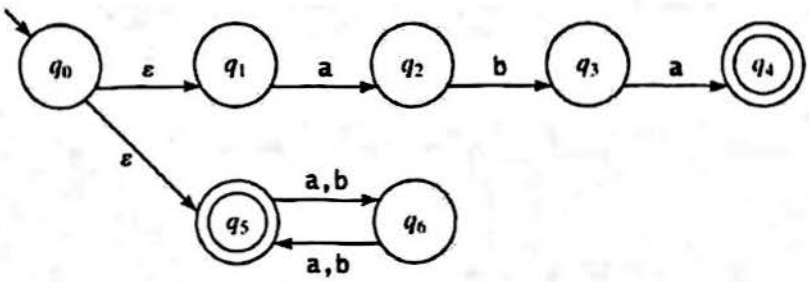
Let $L = \{w \in \{a, b\}^* : w$ is made up of an optional a followed by aa followed by zero or more b's$\}$. The following NDFSM M accepts L:



M may (but is not required to) follow the ε-transition from state q_0 to state q_1 before it reads the first input character. In effect, it must guess whether or not the optional a is present.

EXAMPLE 5.14 Two Different Sublanguages

Let $L = \{w \in \{a, b\}^* : w =$ aba or $|w|$ is even$\}$. An easy way to build an FSM to accept this language is to build FSMs for each of the individual sublanguages and then "glue" them together with ε-transitions. In essence, the machine guesses, when processing a string, which sublanguage the string might be in. So we have:

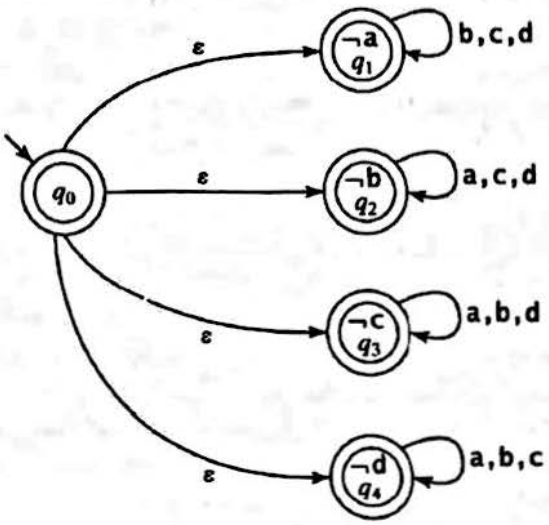


The upper machine accepts $\{w \in \{a, b\}^* : w =$ aba$\}$. The lower one accepts $\{w \in \{a, b\}^* : |w|$ is even$\}$.

By exploiting nondeterminism, it may be possible to build a simple FSM to accept a language for which the smallest deterministic FSM is complex. A good example of a language for which this is true is the missing letter language that we considered in Example 5.12.

EXAMPLE 5.15 The Missing Letter Language, Again

Let $\Sigma = \{a, b, c, d\}$. $L_{Missing} = \{w$: there is a symbol $a_i \in \Sigma$ not appearing in $w\}$. The following simple NDFSM M accepts $L_{Missing}$:



M works by guessing which letter is going to be the missing one. If any of its guesses is right, it will accept. If all of them are wrong, then all paths will fail and M will reject.

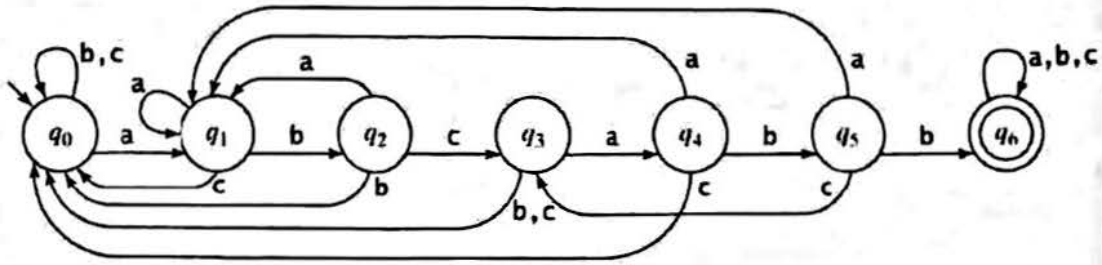
5.4.2 NDFSMs for Pattern and Substring Matching

Nondeterministic FSMs are a particularly effective way to define simple machines to search a text string for one or more patterns or substrings.

EXAMPLE 5.16 Exploiting Nondeterminism for Keyword Matching

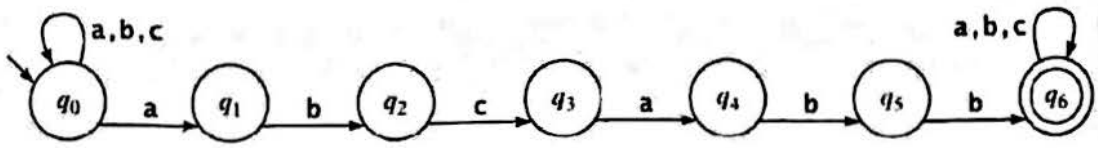
Let $L = \{w \in \{a, b, c\}^* : \exists x, y \in \{a, b, c\}^* (w = x\ abcabb\ y)\}$. In other words, w must contain at least one occurrence of the substring abcabb. The following DFSM M_1 accepts L:

EXAMPLE 5.16 (*Continued*)



While M_1 works, and it works efficiently, designing machines like M_1 and getting them right is hard. The spaghetti-like transitions are necessary because, whenever a match fails, it is possible that another partial match has already been found.

But now consider the following NDFSM M_2, which also accepts L:



The idea here is that, whenever M_2 sees an a, it may guess that it is at the beginning of the pattern abcabb. Or, on any input character (including a), it may guess that it is not yet at the beginning of the pattern (so it stays in q_0). If it ever reaches q_6, it will stay there until it has finished reading the input. Then it will accept.

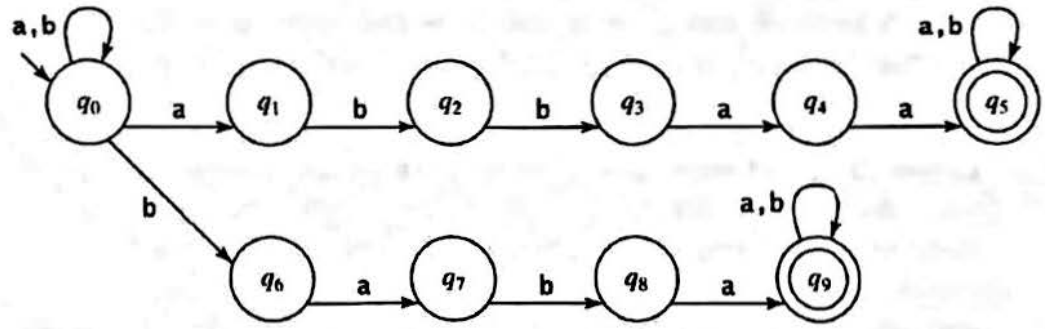
Of course, practical string search engines need to be small and deterministic. But NDFSMs like the one we just built can be used as the basis for constructing such efficient search machines. In Section 5.4.4, we will describe an algorithm that converts an arbitrary NDFSM into an equivalent DFSM. It is likely that that machine will have more states than it needs. But, in Section 5.7, we will present an algorithm that takes an arbitrary DFSM and produces an equivalent minimal one (i.e., one with the smallest number of states). So one effective way to build a correct and efficient string-searching machine is to build a simple NDFSM, convert it to an equivalent DFSM, and then minimize the result. One alternative to this three-step process is the Knuth-Morris-Pratt string search algorithm, which we will present in Example 27.5.

> String searching is a fundamental operation in every word processing or text editing system.

Now suppose that we have not one pattern but several. Hand crafting a DFSM may be even more difficult. One alternative is to use a specialized, keyword-search FSM-building algorithm that we will present in Section 6.2.4. Another is to build a simple NDFSM, as we show in the next example.

EXAMPLE 5.17 Multiple Keywords

Let $L = \{w \in \{a,b\}^* : \exists x, y \in \{a,b\}^* ((w = x\,\text{abbaa}\ y) \vee (w = x\,\text{baba}\ y))\}$. In other words, w contains at least one occurrence of the substring abbaa or the substring baba. The following NDFSM M accepts L:

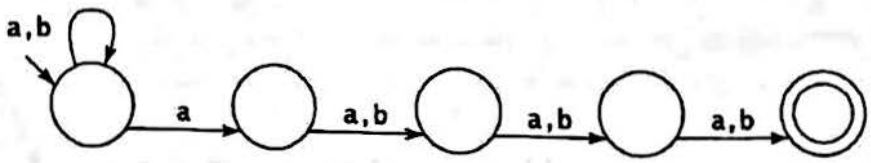


The idea here is that, whenever M sees an a, it may guess that it is at the beginning of the substring abbaa. Whenever it sees a b, it may guess that it is at the beginning of the substring baba. Alternatively, on either a or b, it may guess that it is not yet at the beginning of either substring (so it stays in q_0).

NDFSMs are also a natural way to search for other kinds of patterns, as we can see in the next example.

EXAMPLE 5.18 Other Kinds of Patterns

Let $L = \{w \in \{a, b\}^* :$ the fourth from the last character is a$\}$. The following NDFSM M accepts L:



The idea here is that, whenever it sees an a, one of M's paths guesses that it is the fourth from the last character (and so proceeds along the path that will read the last three remaining characters). The other path guesses that it is not (and so stays in the start state).

It is enlightening to try designing DFSMs for the last two examples. We leave that as an exercise. If you try it, you'll appreciate the value of the NDFSM model as a high-level tool for describing complex systems.

5.4.3 Analyzing Nondeterministic FSMs

Given an NDFSM M, such as any of the ones we have just considered, how can we analyze it to determine what strings it accepts? One way is to do a depth-first search of the paths through the machine. Another is to imagine tracing the execution of the original NDFSM M by following all paths in parallel. To do that, think of M as being in a set of states at each step of its computation. For example, consider again the NDFSM that we built for Example 5.17. You may find it useful to trace the process we are about to describe by using several fingers. Or, when fingers run out, use a coin on each active state. Initially, M is in q_0. If it sees an a, it can loop to state q_0 or go to q_1. So we will think of it as being in the set of states $\{q_0, q_1\}$ (thus we need two fingers or two coins). Suppose it sees a b next. From q_0, it can go to q_0 or q_6. From q_1, it can go to q_2. So, after seeing the string ab, M is in $\{q_0, q_2, q_6\}$ (three fingers or three coins). Suppose it sees a b next. From q_0, it can go to q_0 or q_6. From q_2, it can go to q_3. From q_6, it can go nowhere. So, after seeing abb, M is in $\{q_0, q_3, q_6\}$. And so forth. If, when all the input has been read, M is in at least one accepting state (in this case, q_5 or q_9), then it accepts. Otherwise it rejects.

Handling ε-Transitions

But how shall we handle ε-transitions? The construction that we just sketched assumes that all paths have read the same number of input symbols. But if, from some state q, one transition is labeled ε and another is labeled with some element of Σ, M consumes no input as it takes the first transition and one input symbol as it takes the second transition. To solve this problem, we introduce the function *eps*: $K_M \rightarrow \mathcal{P}(K_M)$. We define $eps(q)$, where q is some state in M, to be the set of states of M that are reachable from q by following zero or more ε-transitions. Formally:

$$eps(q) = \{p \in K : (q, w) \vdash_M^* (p, w)\}.$$

Alternatively, $eps(q)$ is the closure of $\{q\}$ under the relation $\{(p, r)$: there is a transition $(p, \varepsilon, r) \in \Delta\}$. The following algorithm computes *eps*:

eps(*q*: state) =

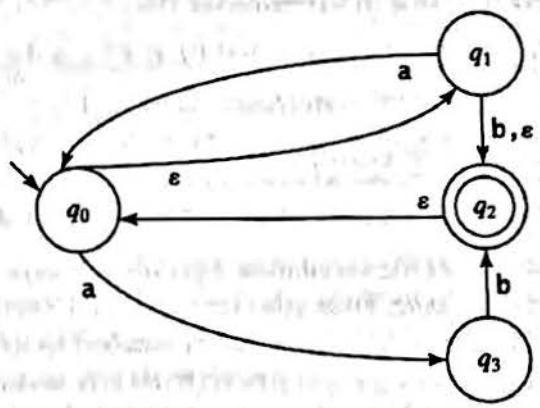
1. *result* = $\{q\}$.
2. While there exists some $p \in$ *result* and some $r \notin$ *result* and some transition $(p, \varepsilon, r) \in \Delta$ do: Insert r into *result*.
3. Return *result*.

This algorithm is guaranteed to halt because, each time through the loop, it adds an element to *result*. It must halt when there are no elements left to add. Since there is only a finite number of candidate elements, namely the finite set of states in M, and no element can be added more than once, the algorithm must eventually run out of elements to add, at which point it must halt. It correctly computes $eps(q)$ because, by the condition associated with the while loop:

- It can add no element that is not reachable from q following only ε-transitions.
- It will add all elements that are reachable from q following only ε-transitions.

EXAMPLE 5.19 Computing *eps*

Consider the following NDFSM M:



To compute $eps(q_0)$, we initially set *result* to $\{q_0\}$. Then q_1 is added, producing $\{q_0, q_1\}$. Then q_2 is added, producing $\{q_0, q_1, q_2\}$. There is an ε-transition from q_2 to q_0, but q_0 is already in *result*. So the computation of $eps(q_0)$ halts.

The result of running *eps* on each of the states of M is:

$$eps(q_0) = \{q_0, q_1, q_2\}.$$
$$eps(q_1) = \{q_0, q_1, q_2\}.$$
$$eps(q_2) = \{q_0, q_1, q_2\}.$$
$$eps(q_3) = \{q_3\}.$$

Example 5.19 illustrates clearly why we chose to define the *eps* function, rather than treating ε-transitions like other transitions and simply following them whenever we could. The machine we had to consider in that example contains what we might choose to call an ***ε-loop:*** a loop that can be traversed by following only ε-transitions. Since such transitions consume no input, there is no limit to the number of times the loop could be traversed. So, if we were not careful, it would be easy to write a simulation algorithm that did not halt. The algorithm that we presented for *eps* halts whenever it runs out of unvisited states to add, which must eventually happen since the set of states is finite.

A Simulation Algorithm

With the *eps* function in hand, we can now define an algorithm for tracing all paths in parallel through an NDFSM M:

ndfsmsimulate (M: NDFSM, w: string) =

1. *current-state* = $eps(s)$. /*Start in the set that contains M's start state and any other states that can be reached from it following only ε-transitions.

2. While any input symbols in w remain to be read do:
   2.1. c = get-next-symbol(w).
   2.2. *next-state* = $\emptyset$.
   2.3. For each state q in *current-state* do:
      For each state p such that $(q, c, p) \in \Delta$ do:
         next-state = *next-state* $\cup$ *eps* (p).
   2.4. *current-state* = *next-state*.
3. If *current-state* contains any states in A, accept. Else reject.

Step 2.3 is the core of the simulation algorithm. It says: Follow every arc labeled c from every state in *current-state*. Then compute *next-state* (and thus the new value of *current-state*) so that it includes every state that is reached in that process, plus every state that can be reached by following ε-transitions from any of those states. For more on how this step can be implemented, see the more detailed description of *ndfsmsimulate* that we present in Section 5.6.2.

5.4.4 The Equivalence of Nondeterministic and Deterministic FSMs

In this section, we explore the relationship between the DFSM and NDFSM models that we have just defined.

THEOREM 5.2 If There is a DFSM for *L*, There is an NDFSM for *L*

Theorem: For every DFSM there is an equivalent NDFSM.

Proof: Let M be a DFSM that accepts some language L. M is also an NDFSM that happens to contain no ε-transitions and whose transition relation happens to be a function. So the NDFSM that we claim must exist is simply M.

But what about the other direction? The nondeterministic model that we have just introduced makes it substantially easier to build FSMs to accept some kinds of languages, particularly those that involve looking for instances of complex patterns. But real computers are deterministic. What does the existence of an NDFSM to accept a language L tell us about the existence of a deterministic program to accept L? The answer is given by the following theorem:

THEOREM 5.3 If There is an NDFSM for *L*, There is a DFSM for *L*

Theorem: Given an NDFSM $M = (K, \Sigma, \Delta, s, A)$ that accepts some language L, there exists an equivalent DFSM that accepts L.

Proof: The proof is by construction of an equivalent DFSM M'. The construction is based on the function *eps* and on the simulation algorithm that we described in the last section. The states of M' will correspond to sets of states in M. So $M' = (K', \Sigma, \delta', s', A')$, where:

- K' contains one state for each element of $\mathcal{P}(K)$.
- $s' = eps(s)$.
- $A' = \{Q \subseteq K : Q \cap A \neq \emptyset\}$.
- $\delta'(Q, c) = \cup \{eps(p) : \exists q \in Q ((q, c, p) \in \Delta)\}$.

We should note the following things about this definition:

- In principle, there is one state in K' for each element of $\mathcal{P}(K)$. However, in most cases, many of those states will be unreachable from s' (and thus unnecessary). So we will present a construction algorithm that creates states only as it needs to.
- We'll name each state in K' with the element of $\mathcal{P}(K)$ to which it corresponds. That will make it relatively straightforward to see how the construction works. But keep in mind that those labels are just names. We could have called them anything.
- To decide whether a state in K' is an accepting state, we see whether it corresponds to an element of $\mathcal{P}(K)$ that contains at least one element of A, i.e., one accepting state from K.
- M' accepts whenever it runs out of input and is in a state that contains at least one accepting state of M. Thus it implements the definition of an NDFSM, which accepts iff at least one path through it accepts.
- The definition of δ' corresponds to step 2.3 of the simulation algorithm we presented above.

The following algorithm computes M' given M:

ndfsmtodfsm(M: NDFSM) =

1. For each state q in K do:
   Compute $eps(q)$. /* These values will be used below.
2. $s' = eps(s)$.
3. Compute δ':
   a. *active-states* $= \{s'\}$. /* We will build a list of all states that are reachable from the start state. Each element of *active-states* is a set of states drawn from K.
   b. $\delta' = \emptyset$.
   c. While there exists some element Q of *active-states* for which δ' has not yet been computed do:
      For each character c in Σ do:
         new-state $= \emptyset$.
         For each state q in Q do:

For each state p such that $(q, c, p) \in \Delta$ do:

new-state = *new-state* $\cup$ *eps*(p).

Add the transition (Q, c, *new-state*) to δ'.

If *new-state* $\notin$ *active-states* then insert it into *active-states*.

4. $K' =$ *active-states*.
5. $A' = \{Q \in K' : Q \cap A \neq \emptyset\}$.

The core of *ndfsmtodfsm* is the loop in step 3.3. At each step through it, we pick a state that we know is reachable from the start state but from which we have not yet computed transitions. Call it Q. Then compute the paths from Q for each element c of the input alphabet as follows: Q is a set of states in the original NDFSM M. So consider each element q of Q. Find all transitions from q labeled c. For each state p that is reached by such a transition, find all additional states that are reachable by following only ε-transitions from p. Let *new-state* be the set that contains all of those states. Now we know that whenever M' is in Q and it reads a c, it should go to *new-state*.

The algorithm *ndfsmtodfsm* halts on all inputs and constructs a DFSM M' that accepts exactly $L(M)$, the language accepted by M.

> A rigorous construction proof requires a proof that the construction algorithm is correct. We will generally omit the details of such proofs. But we show them for this case as an example of what these proofs look like. (Appendix C)

The algorithm *ndfsmtodfsm* is important for two reasons:

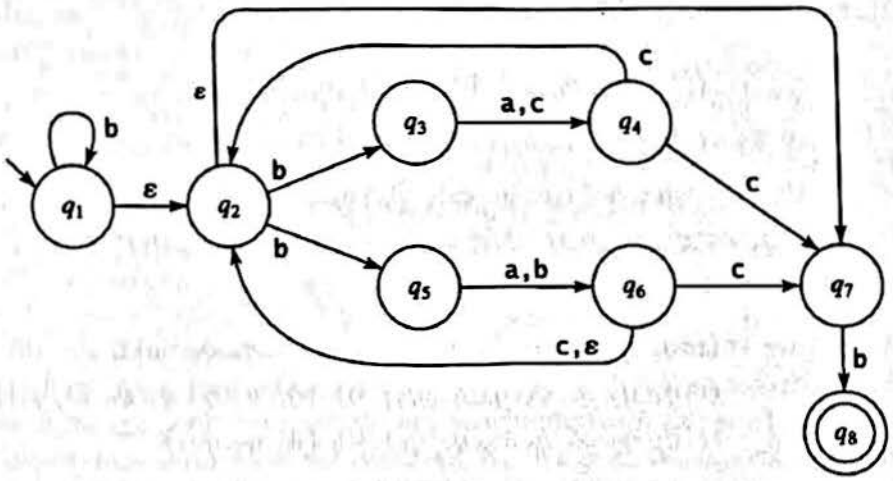
- It proves the theorem that, for every NDFSM there exists an equivalent DFSM.
- It lets us use nondeterminism as a design tool, even though we may ultimately need a deterministic machine. If we have an implementation of *ndfsmtodfsm*, then, if we can build an NDFSM to solve our problem, *ndfsmtodfsm* can easily construct an equivalent DFSM.

EXAMPLE 5.20 Using *ndfsmtodfsm* to Build a Deterministic FSM

Consider the NDFSM M shown on the next page. To get a feel for M, simulate it on the input string bbbacb, using coins to keep track of the states it enters.

We can apply *ndfsmtodfsm* to M as follows:

1. Compute *eps*(q) for each state q in K_M:



$eps\,(q_1) = \{q_1, q_2, q_7\}$, $eps\,(q_2) = \{q_2, q_7\}$, $eps\,(q_3) = \{q_3\}$, $eps\,(q_4) = \{q_4\}$, $eps\,(q_5) = \{q_5\}$, $eps\,(q_6) = \{q_2, q_6, q_7\}$, $eps\,(q_7) = \{q_7\}$, $eps\,(q_8) = \{q_8\}$.

2. $s' = eps\,(s) = \{q_1, q_2, q_7\}$.

3. Compute δ':

active-states = $\{\{q_1, q_2, q_7\}\}$. Consider $\{q_1, q_2, q_7\}$:

$((\{q_1, q_2, q_7\}, \mathrm{a}), \varnothing)$.
$((\{q_1, q_2, q_7\}, \mathrm{b}), \{q_1, q_2, q_3, q_5, q_7, q_8\})$.
$((\{q_1, q_2, q_7\}, \mathrm{c}), \varnothing)$.

active-states = $\{\{q_1, q_2, q_7\}, \varnothing, \{q_1, q_2, q_3, q_5, q_7, q_8\}\}$. Consider $\varnothing$:

$((\varnothing, \mathrm{a}), \varnothing)$. /* $\varnothing$ is a dead state and we will generally omit it.
$((\varnothing, \mathrm{b}), \varnothing)$.
$((\varnothing, \mathrm{c}), \varnothing)$.

active-states = $\{\{q_1, q_2, q_7\}, \varnothing, \{q_1, q_2, q_3, q_5, q_7, q_8\}\}$. Consider $\{q_1, q_2, q_3, q_5, q_7, q_8\}$:

$((\{q_1, q_2, q_3, q_5, q_7, q_8\}, \mathrm{a}), \{q_2, q_4, q_6, q_7\})$.
$((\{q_1, q_2, q_3, q_5, q_7, q_8\}, \mathrm{b}), \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\})$.
$((\{q_1, q_2, q_3, q_5, q_7, q_8\}, \mathrm{c}), \{q_4\})$.

active-states = $\{\{q_1, q_2, q_7\}, \varnothing, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}\}$. Consider $\{q_2, q_4, q_6, q_7\}$:

$((\{q_2, q_4, q_6, q_7\}, \mathrm{a}), \varnothing)$.

EXAMPLE 5.20 (*Continued*)

$((\{q_2, q_4, q_6, q_7\}, \mathsf{b}), \{q_3, q_5, q_8\})$.

$((\{q_2, q_4, q_6, q_7\}, \mathsf{c}), \{q_2, q_7\})$.

active-states = $\{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}\}$.

Consider $\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}$:

$((\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \mathsf{a}), \{q_2, q_4, q_6, q_7\})$.

$((\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \mathsf{b}), \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\})$.

$((\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \mathsf{c}), \{q_2, q_4, q_7\})$.

active-states = $\{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}, \{q_2, q_4, q_7\}\}$. Consider $\{q_4\}$:

$((\{q_4\}, \mathsf{a}), \emptyset)$.

$((\{q_4\}, \mathsf{b}), \emptyset)$.

$((\{q_4\}, \mathsf{c}), \{q_2, q_7\})$.

active-states did not change. Consider $\{q_3, q_5, q_8\}$:

$((\{q_3, q_5, q_8\}, \mathsf{a}), \{q_2, q_4, q_6, q_7\})$.

$((\{q_3, q_5, q_8\}, \mathsf{b}), \{q_2, q_6, q_7\})$.

$((\{q_3, q_5, q_8\}, \mathsf{c}), \{q_4\})$.

active-states = $\{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}, \{q_2, q_4, q_7\}, \{q_2, q_6, q_7\}\}$.

Consider $\{q_2, q_7\}$:

$((\{q_2, q_7\}, \mathsf{a}), \emptyset)$.

$((\{q_2, q_7\}, \mathsf{b}), \{q_3, q_5, q_8\})$.

$((\{q_2, q_7\}, \mathsf{c}), \emptyset)$.

active-states did not change. Consider $\{q_2, q_4, q_7\}$:

$((\{q_2, q_4, q_7\}, \mathsf{a}), \emptyset)$.

$((\{q_2, q_4, q_7\}, \mathsf{b}), \{q_3, q_5, q_8\})$.

$((\{q_2, q_4, q_7\}, \mathsf{c}), \{q_2, q_7\})$.

active-states did not change. Consider $\{q_2, q_6, q_7\}$:

$((\{q_2, q_6, q_7\}, \mathsf{a}), \emptyset)$.

$((\{q_2, q_6, q_7\}, \mathsf{b}), \{q_3, q_5, q_8\})$.

$((\{q_2, q_6, q_7\}, \mathsf{c}), \{q_2, q_7\})$.

active-states did not change. δ has been computed for each element of *active-states*.

4. $K' = \{\{q_1, q_2, q_7\}, \varnothing, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}, \{q_2, q_4, q_7\}, \{q_2, q_6, q_7\}\}$.
5. $A' = \{\{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_3, q_5, q_8\}\}$.

Notice that, in Example 5.20, the original NDFSM had 8 states. So $|\mathcal{P}(K)| = 256$. There could have been that many states in the DFSM that was constructed from the original machine. But only 10 of those are reachable from the start state and so can play any role in the operation of the machine. We designed the algorithm *ndfsmtodfsm* so that only those 10 would have to be built.

Sometimes, however, all or almost all of the possible subsets of states are reachable. Consider again the NDFSM of Example 5.15, the missing letter machine. Let's imagine a slight variant that considers all 26 letters of the alphabet. That machine M has 27 states. So, in principle, the corresponding DFSM could have 2^{27} states. And, this time, all subsets are possible except that M can not be in the start state, q_0, at any time except before the first character is read. So the DFSM that we would build if we applied *ndfsmtodfsm* to M would have $2^{26} + 1$ states. In Section 5.6, we will describe a technique for interpreting NDFSMs without converting them to DFSMs first. Using that technique, highly nondeterministic machines, like the missing letter one, are still practical.

What happens if we apply *ndfsmtodfsm* to a machine that is already deterministic? It must work, since every DFSM is also a legal NDFSM. You may want to try it on one of the machines in Section 5.3. What you will see is that the machine that *ndfsmtodfsm* builds, given an input DFSM M, is identical to M except for the names of the states.

5.5 From FSMs to Operational Systems

An FSM is an abstraction. We can describe an FSM that solves a problem without worrying about many kinds of implementation details. In fact, we don't even need to know whether it will be etched into silicon or implemented in software.

> Statecharts, which are based on the idea of hierarchically structured transition networks, are widely used in software engineering precisely because they enable system designers to work at varying levels of abstraction. (H.2)

FSMs for real problems can be turned into operational systems in any of a number of ways:

- An FSM can be translated into a circuit design and implemented directly in hardware. For example, it makes sense to implement the parity checking FSM of Example 5.4 in hardware.

- An FSM can be simulated by a general purpose interpreter. We will describe designs for such interpreters in the next section. Sometimes all that is required is a simulation. In other cases, a simulation can be used to check a design before it is translated into hardware.
- An FSM can be used as a specification for some critical aspect of the behavior of a complex system. The specification can then be implemented in software just as any specification might be. And the correctness of the implementation can be shown by verifying that the implementation satisfies the specification (i.e., that it matches the FSM).

> Many network communication protocols, including the Alternating Bit protocol and TCP, are described as FSMs. (1.1)

5.6 Simulators for FSMs ●

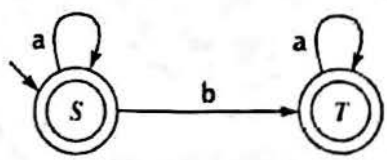
Once we have created an FSM to solve a problem, we may want to simulate its execution. In this section, we consider techniques for doing that, starting with DFSMs, and then extending our ideas to handle nondeterminism.

5.6.1 Simulating Deterministic FSMs

We begin by considering only deterministic FSMs. One approach is to think of an FSM as the specification for a simple, table-driven program and then proceed to write the code.

EXAMPLE 5.21 Hardcoding a Deterministic FSM

Consider the following deterministic FSM M that accepts the language $L = \{w \in \{a, b\}^* : w \text{ contains no more than one } b\}$.



We could view M as a specification for the following program:

```
Until accept or reject do:
    S:   s = get-next-symbol.
         If s = end-of-file then accept.
         Else if s = a then go to S.
         Else if s = b then go to T.
```