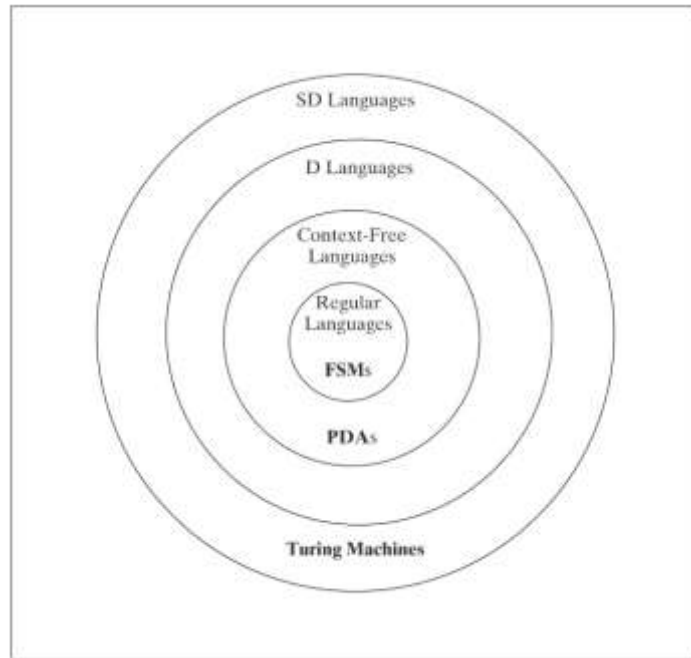


Part II: Finite State Machines and Regular Languages

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.



5 Finite State Machines

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

The history of finite state machines substantially predates modern computers. © 795.

5.1 Deterministic Finite State Machines

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.

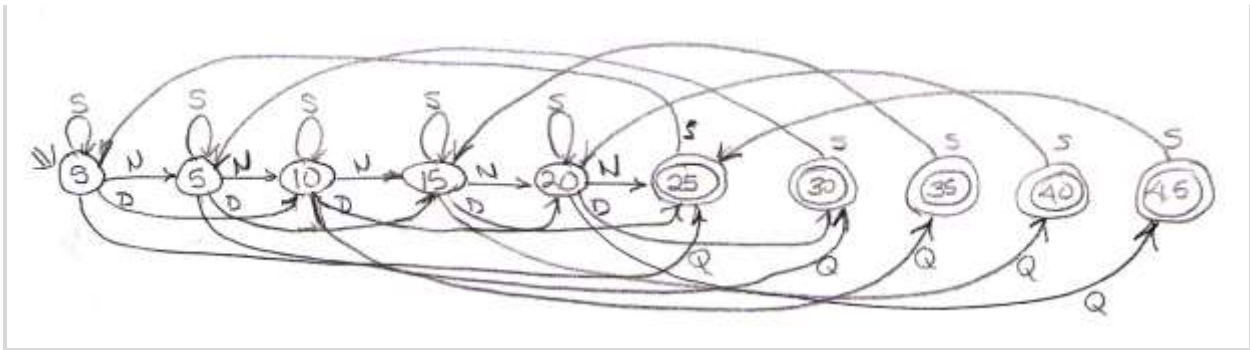
After any finite sequence of inputs, the controller will be in either:

- a dispensing state, in which it is willing to dispense a drink if a drink button is pushed, or
- 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.
- 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 accept 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.

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, the 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:

$$\begin{array}{ccccc} K & \times & \Sigma & \text{to} & K. \\ \text{state} & & \text{input symbol} & & \text{state} \end{array}$$

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, and
- 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 \vdash_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:

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

We can now define the relation *yields*, written \vdash_M^* to be the reflexive, transitive closure of \vdash_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:

$$C_1 \vdash_M^* C_2.$$

A **computation** by M is a finite sequence of configurations C_0, C_1, \dots, 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
- $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \dots \vdash_M C_n$.

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

- M **accepts** w iff $(s, w) \vdash_M^* (q, \varepsilon)$, 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 $(s, w) \vdash_M^* (q, \varepsilon)$, 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

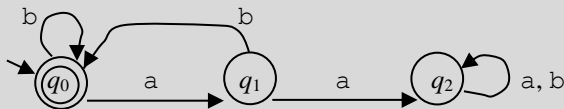
Let $L = \{w \in \{a, b\}^* : \text{every } a \text{ is immediately followed by a } b\}$. L can be accepted by the DFSM $M = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_0\})$, where:

$$\delta = \{ ((q_0, a), q_1), ((q_0, b), q_0), ((q_1, a), q_2), ((q_1, b), q_0), ((q_2, a), q_2), ((q_2, b), q_2) \}.$$

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:

- The start state will be indicated with an unlabeled arrow pointing into it.
- 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 .

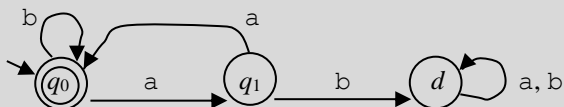
As an example of M 's operation, consider the input string $abbabab$. M 's computation is the sequence of configurations: $(q_0, abbabab)$, $(q_1, bbabab)$, $(q_0, babab)$, $(q_0, abab)$, (q_1, bab) , (q_0, ab) , (q_1, b) , (q_0, ϵ) . Since q_0 is an accepting state, M accepts.

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

- State q_0 is an accepting state. Every string that drives M to state q_0 is in L .
- 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 .
- 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 $L = \{w \in \{a, b\}^* : \text{every } a \text{ region in } w \text{ is of even length}\}$. L can be accepted by the DFSM M :



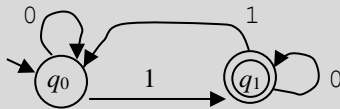
If M sees a b in state q_1 , then there has been an a region whose length is odd. So, no matter what happens next, M must reject. So it goes to the dead state d .

A useful way to prototype a complex system is as a finite state machine. See $\text{C } 801$ 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 $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \dots \vdash_M C_n$, 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:

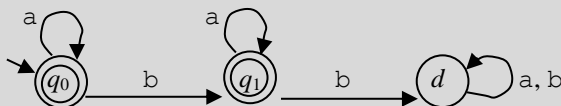
- “enough money to buy a drink”.
- $\{w \in \{a,b\}^* : \text{every } a \text{ is immediately followed by a } b\}$.
- $\{w \in \{a,b\}^* : \text{every } a \text{ region in } w \text{ is of even length}\}$.
- binary strings with odd parity.

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

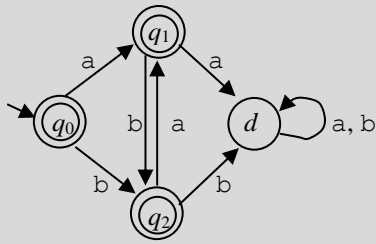
Let $L = \{w \in \{a,b\}^* : w \text{ contains no more than one } b\}$. L is regular because it can be accepted by the DFSM M :



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 $L = \{w \in \{a, b\}^* : \text{no two consecutive characters are the same}\}$. 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. $\text{C } 704$.

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

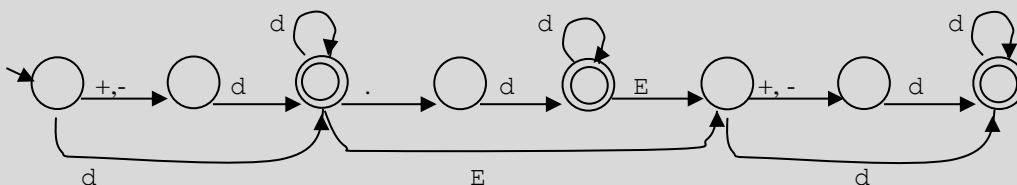
Let $\text{FLOAT} = \{w : w \text{ is the string representation of a floating point number}\}$. Assume the following syntax for floating point numbers:

- 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.
- 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:



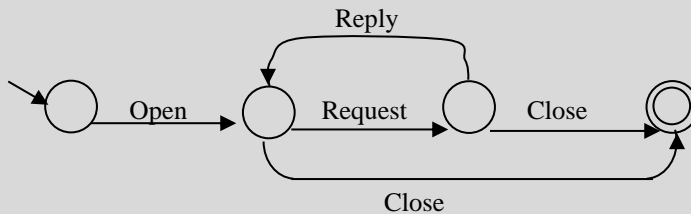
In this diagram, we have used the shorthand d to stand for any one of the decimal digits (0 - 9). And we have omitted the dead state to avoid arrows crossing over each other.

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 $\Sigma_L = \{\text{Open, Request, Reply, Close}\}$. 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. © 693.

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:

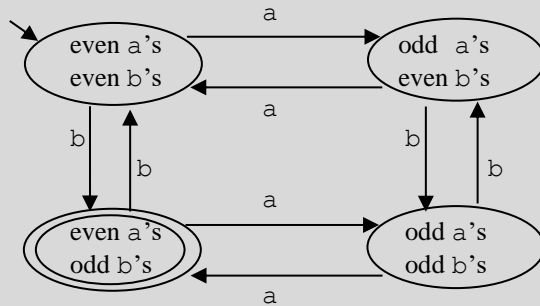
- Imagine any DFSM M that accepts L . As a string w is being read by M , what properties of the part of w that has been seen so far are going to have any bearing on the ultimate answer that M needs to produce? Those are the properties that M needs to record. So, for example, in the “enough money” machine, 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. © 717.

Example 5.9 Even a's, Odd b's

Let $L = \{w \in \{a, b\}^* : w \text{ contains an even number of } a\text{'s and an odd number of } b\text{'s}\}$. 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

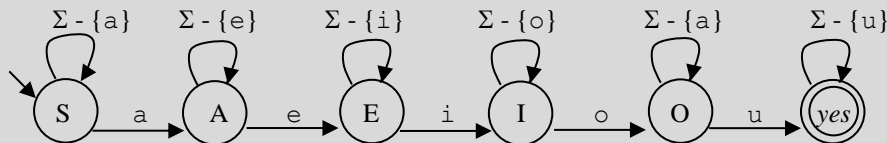
need a four-state DFSM. Often it helps to name the states with a description of the clusters to which they correspond. The following DFSM M accepts L :



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 $L = \{w \in \{a-z\}^* : \text{all five vowels, } a, e, i, o, \text{ and } u, \text{ occur in } w \text{ in alphabetical order}\}$. So L contains words like *abstemious*, *facetious*, and *sacriligious*. 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 ":

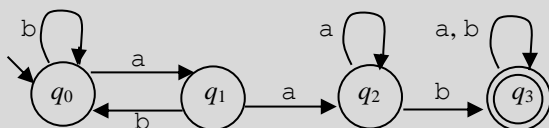


Notice that the state that we have labeled *yes* functions exactly opposite to the way in which the dead state works. If M ever reaches *yes*, it has decided to accept no matter what comes next.

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 $L = \{w \in \{a, b\}^* : w \text{ does not contain the substring } aab\}$. It is straightforward to design a DFSM 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 $L_{Missing} = \{w : \text{there is a symbol } a_i \in \Sigma \text{ not appearing in } w\}$. $L_{Missing}$ 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.

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,
- $A \subseteq K$ is the set of final states, and
- Δ is the transition relation. It is a finite subset of:

$$(K \times (\Sigma \cup \{\epsilon\})) \times K.$$

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.

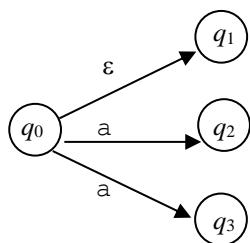


Figure 5.1 An NDFSM with two kinds of nondeterminism

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 .

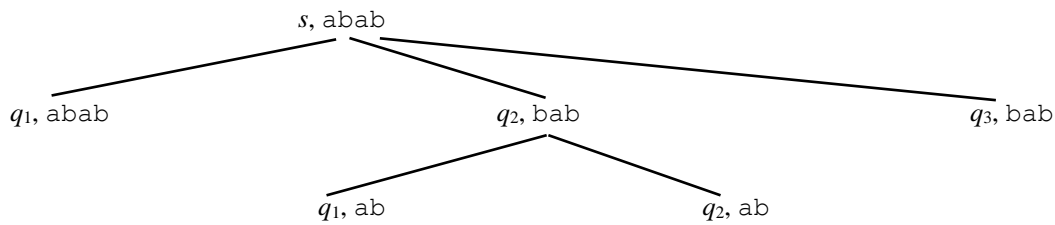
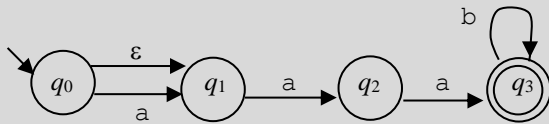


Figure 5.2 Viewing nondeterminism as search through a space of computation paths

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.

Example 5.13 An Optional Initial a

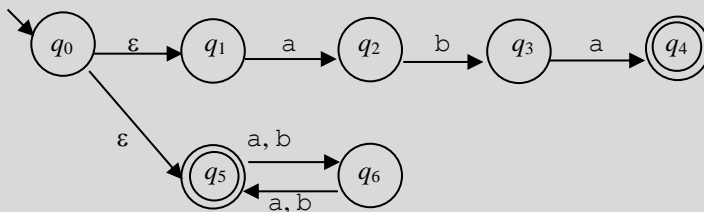
Let $L = \{w \in \{a, b\}^* : w \text{ is made up of an optional } a \text{ followed by } aa \text{ followed by zero or more } b\text{'s}\}$. The following NDFSM M accepts L :



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 $L = \{w \in \{a, b\}^* : w = aba \text{ or } |w| \text{ is even}\}$. 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:

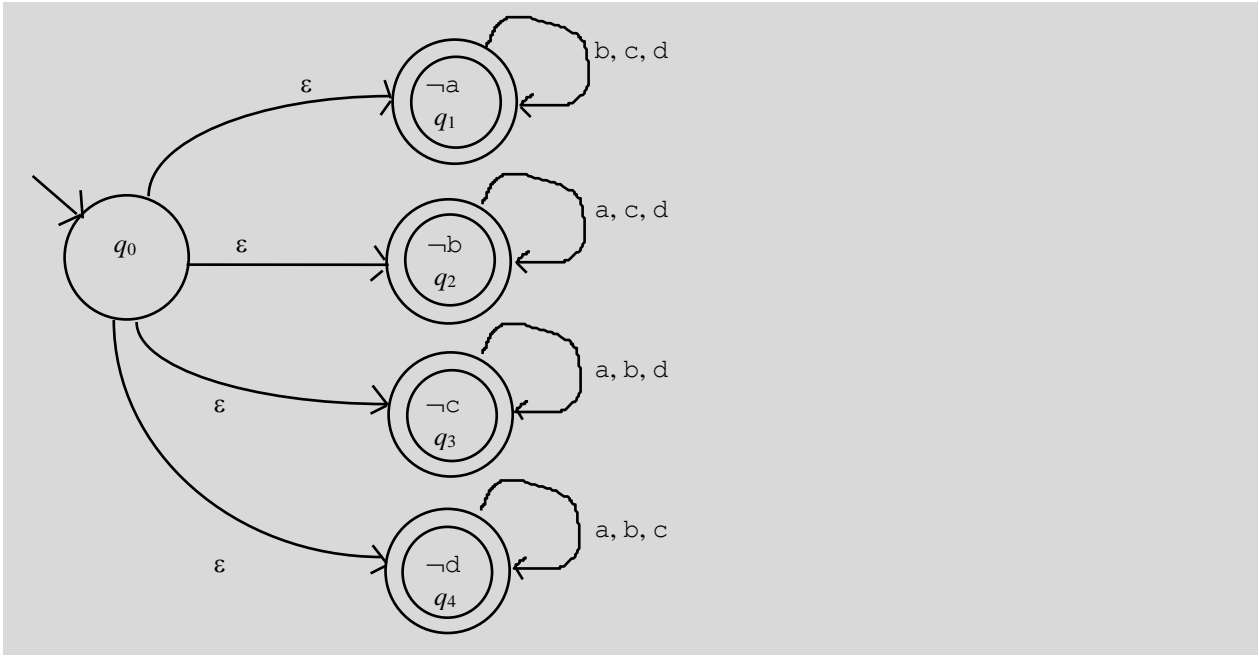


The upper machine accepts $\{w \in \{a, b\}^* : w = aba\}$. The lower one accepts $\{w \in \{a, b\}^* : |w| \text{ is even}\}$.

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



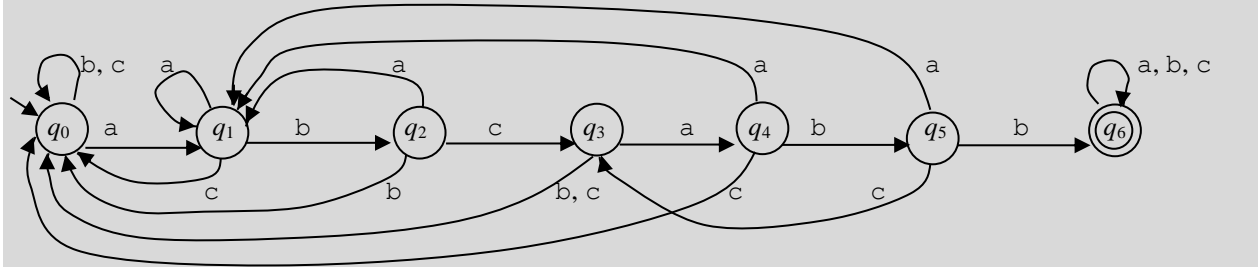
M works by guessing which letter is going to be the missing one. If any of its guesses is right, it will accept. If all of them are wrong, then all paths will fail and M will reject.

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.

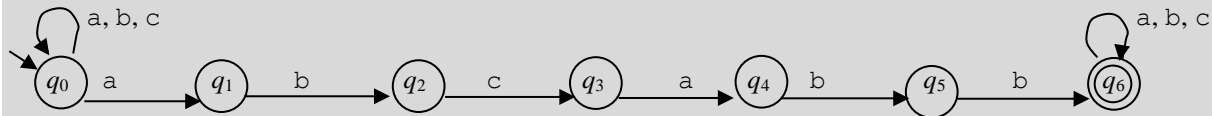
Example 5.16 Exploiting Nondeterminism for Keyword Matching

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 :



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.

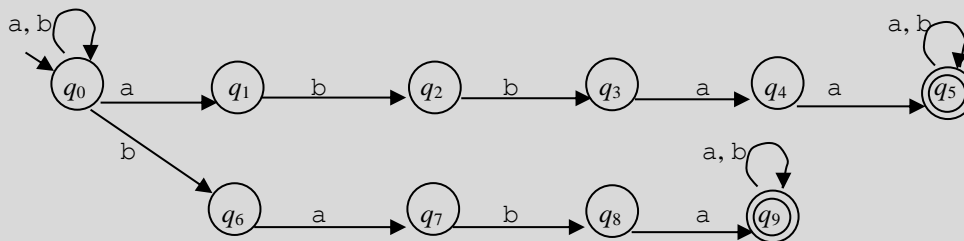
Of course, practical string search engines need to be small and deterministic. But NDFSMs like the one we just built can be used as the basis for constructing such efficient search machines. In Section 5.4.4, we will describe an algorithm that converts an arbitrary NDFSM into an equivalent DFSM. It is 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 $L = \{w \in \{a, b\}^* : \exists x, y \in \{a, b\}^* ((w = x abbaa y) \vee (w = x baba y))\}$. In other words, w contains at least one occurrence of the substring $abbaa$ or the substring $baba$. The following NDFSM M accepts L :

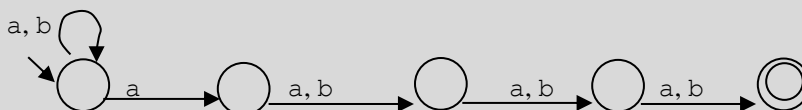


The idea here is that, whenever M sees an a , it may guess that it is at the beginning of the substring $abbaa$. Whenever it sees a b , it may guess that it is at the beginning of the substring $baba$. Alternatively, on either a or b , it may guess that it is not yet at the beginning of either substring (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 $L = \{w \in \{a, b\}^* : \text{the fourth from the last character is } a\}$. 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 DFMSs 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 $\{(p, r) : \text{there is a transition } (p, \epsilon, r) \in \Delta\}$. The following algorithm computes eps :

$eps(q: \text{state}) =$

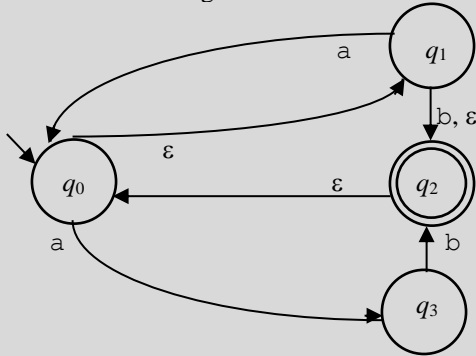
1. $result = \{q\}$.
2. While there exists some $p \in result$ and some $r \notin result$ and some transition $(p, \epsilon, r) \in \Delta$ do:
 Insert r into $result$.
3. Return $result$.

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 are 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 $e(q_0)$ halts.

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

$$\begin{aligned} eps(q_0) &= \{q_0, q_1, q_2\}. \\ eps(q_1) &= \{q_0, q_1, q_2\}. \\ eps(q_2) &= \{q_0, q_1, q_2\}. \\ eps(q_3) &= \{q_3\}. \end{aligned}$$

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 :

$ndfmssimulate(M: \text{NDFSM}, w: \text{string}) =$

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. $c = \text{get-next-symbol}(w)$.
 - 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 $ndfmssimulate$ 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 NDFSMS models that we have just defined.

Theorem 5.2 If There is a DFSM for L , There is an NDFSMS for L

Theorem: For every DFSM there is an equivalent NDFSMS.

Proof: Let M be a DFSM that accepts some language L . M is also an NDFSMS that happens to contain no ϵ -transitions and whose transition relation happens to be a function. So the NDFSMS 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 NDFSMS 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 NDFSMS for L , There is a DFSM for L

Theorem: Given an NDFSMS $M = (K, \Sigma, \delta, s, A)$ 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 NDFSMS, 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( $M$ : NDFSMS) =
  1. For each state  $q$  in  $K$  do:
      Compute  $eps(q)$ .          /* These values will be used below.
  2.  $s' = eps(s)$ 
  3. Compute  $\delta'$ :
      3.1.  $active-states = \{s'\}$ .    /* 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  $K$ .
```

- 3.2. $\delta' = \emptyset$.
- 3.3. 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 . ■

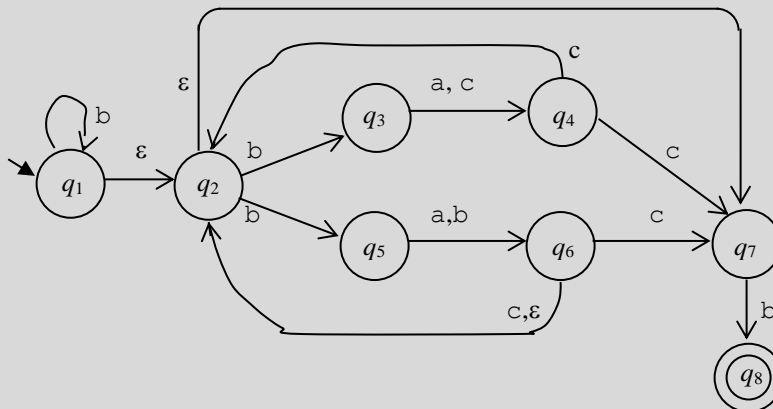
A rigorous construction proof requires a proof that the construction algorithm is correct. We will generally omit the details of such proofs. But we show them for this case as an example of what these proofs look like. [§ 627](#).

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 following NDFSM M :



First, 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 :

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

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

3. Compute δ' :

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

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

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

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

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

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

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

Consider $\{q_2, q_4, q_6, q_7\}$:

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

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

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

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 $\{q_4\}$:

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

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

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

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 $\{q_2, q_7\}$:

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

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

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

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

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

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

4. $K' = \{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}, \{q_2, q_4, q_7\}, \{q_2, q_6, q_7\}\}$.
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 5.2. 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. © 663.

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. © 693.

5.6 Simulators for FSMs ♦

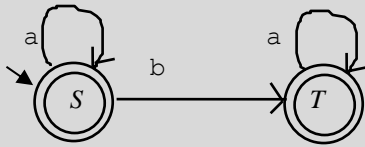
Once we have created an FSM to solve a problem, we may want to simulate its execution. In this section, we consider techniques for doing that, starting with DFSMs, and then extending our ideas to handle nondeterminism.

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

Given an FSM M with states K , this approach will create a program of length $= 2 + (|K| \cdot (|\Sigma| + 2))$. The time required to analyze an input string w is $\mathcal{O}(|w| \cdot |\Sigma|)$. The biggest problem with this approach is that we must generate new code for every FSM that we wish to run. Of course, we could write an FSM compiler that did that for us. But we don't need to. We can, instead, build an interpreter that executes the FSM directly.

Here's a simple interpreter for a deterministic FSM $M = (K, \Sigma, \delta, s, A)$:

dfsmsimulate(M : DFMSM, w : string) =

1. $st = s$.
2. Repeat:
 - 2.1. $c = \text{get-next-symbol}(w)$.
 - 2.2. If $c \neq \text{end-of-file}$ then:
 $st = \delta(st, c)$.
3. If $st \in A$ then accept else reject.

The algorithm *dfsmsimulate* runs in time approximately $\mathcal{O}(|w|)$, if we assume that the lookup in step 2.2.1 can be implemented in constant time.

5.6.2 **Simulating Nondeterministic FSMs**

Now suppose that we want to execute an NDFSM M . One solution is:

ndfsmconvertandsimulate(M : NDFSM) =
dfsmsimulate(*ndfsmto**dfsm*(M)).

But, as we saw in Section 5.4, converting an NDFSM to a DFMSM can be very inefficient in terms of both time and space. If M has k states, it could take time and space equal to $\mathcal{O}(2^k)$ just to do the conversion, although the simulation, after the conversion would take time equal to $\mathcal{O}(|w|)$. So we would like a better way. We would like an algorithm that directly simulates an NDFSM M without converting it to a DFMSM first.

We sketched such an algorithm *ndfsmsimulate* in our discussion leading up to the definition of the conversion algorithm *ndfsmto**dfsm*. The idea is to simulate being in sets of states at once. But, instead of generating all of the

reachable sets of states right away, as *ndfsmtdfsm* does, it generates them on the fly, as they are needed, being careful not to get stuck chasing ϵ -loops.

We give here a more detailed description of *ndfsmsimulate*, which simulates an NDFSM $M = (K, \Sigma, \Delta, s, A)$ running on an input string w :

```

ndfsmsimulate( $M$ : NDFSM,  $w$ : string) =
1.  Declare the set  $st$ .                                /*  $st$  will hold the current state (a set of states from  $K$ ).
2.  Declare the set  $st1$ .                                /*  $st1$  will be built to contain the next state.
3.   $st = \text{eps}(s)$ .                                    /* Start in all states reachable from  $s$  via only  $\epsilon$ -transitions.
4.  Repeat:
     $c = \text{get-next-symbol}(w)$ .
    If  $c \neq \text{end-of-file}$  then do:
         $st1 = \emptyset$ .
        For all  $q \in st$  do:                               /* Follow paths from all states  $M$  is currently in.
            For all  $r : (q, c, r) \in \Delta$  do:           /* Find all states reachable from  $q$  via a transition labeled  $c$ .
                 $st1 = st1 \cup \text{eps}(r)$ .               /* Follow all  $\epsilon$ -transitions from there.
             $st = st1$ .                                    /* Done following all paths. So  $st$  becomes  $M$ 's new state.
            If  $st = \emptyset$  then exit.                  /* If all paths have died, quit.
        until  $c = \text{end-of-file}$ .
5.  If  $st \cap A \neq \emptyset$  then accept else reject.

```

Now there is no conversion cost. To analyze a string w requires $|w|$ passes through the main loop in step 4. In the worst case, M is in all states all the time and each of them has a transition to every other one. So one pass could take as many as $\mathcal{O}(|K|^2)$ steps, for a total cost of $\mathcal{O}(|w| \cdot |K|^2)$.

There is also a third way we could build a simulator for an NDFSM. We could build a depth-first search program that examines the paths through M and stops whenever either it finds a path that accepts or it has tried all the paths there are.

5.7 Minimizing FSMs

If we are going to solve a real problem with an FSM, we may want to find the smallest one that does the job. We will say that a DFSM M is *minimal* iff there is no other DFSM M' such that $L(M) = L(M')$ and M' has fewer states than M does.

We might want to be able to ask:

1. Given a language, L , is there a minimal DFSM that accepts L ?
2. If there is a minimal machine, is it unique?
3. Given a DFSM M that accepts some language L , can we tell whether M is minimal?
4. Given a DFSM M , can we construct a minimal equivalent DFSM M' ?

The answer to all four questions is yes. We'll consider questions 1 and 2 first, and then consider questions 3 and 4.

5.7.1 Building a Minimal DFSM for a Language

Recall that in Section 5.3 we suggested that an effective way to think about the design of a DFSM M to accept some language L over an alphabet Σ was to cluster the strings in Σ^* in such a way that strings that share a future will drive M to the same state. We will now formalize that idea and use it as the basis for constructing a minimal DFSM to accept L .

We will say that x and y are *indistinguishable* with respect to L , which we will write as $x \approx_L y$ iff:

$$\forall z \in \Sigma^* \text{ (either both } xz \text{ and } yz \in L \text{ or neither is).}$$

In other words, \approx_L is a relation that is defined so that $x \approx_L y$ precisely in case, if x and y are viewed as prefixes of some longer string, no matter what continuation string z comes next, either both xz and yz are in L or both are not.

Example 5.22 How \approx_L Depends on L

If $L = \{a\}^*$, then $a \approx_L aa \approx_L aaa$. But if $L = \{w \in \{a, b\}^* : |w| \text{ is even}\}$, then $a \approx_L aaa$, but it is not the case that $a \approx_L aa$ because, if $z = a$, we have $aa \in L$ but $aaa \notin L$.

We will say that x and y are *distinguishable* with respect to L , iff they are not indistinguishable. So, if x and y are distinguishable, then there exists at least one string z such that one but not both of xz and yz is in L .

Note that \approx_L is an equivalence relation because it is:

- reflexive: $\forall x \in \Sigma^* (x \approx_L x)$, because $\forall x, z \in \Sigma^* (xz \in L \leftrightarrow xz \in L)$.
- symmetric: $\forall x, y \in \Sigma^* (x \approx_L y \rightarrow y \approx_L x)$, because $\forall x, y, z \in \Sigma^* ((xz \in L \leftrightarrow yz \in L) \leftrightarrow (yz \in L \leftrightarrow xz \in L))$.
- transitive: $\forall x, y, z \in \Sigma^* (((x \approx_L y) \wedge (y \approx_L w)) \rightarrow (x \approx_L w))$, because:

$$\forall x, y, z \in \Sigma^* (((xz \in L \leftrightarrow yz \in L) \wedge (yz \in L \leftrightarrow wz \in L)) \rightarrow (xz \in L \leftrightarrow wz \in L)).$$

We will use three notations to describe the equivalence classes of \approx_L :

- $[1], [2]$, etc. will refer to explicitly numbered classes.
- $[x]$ describes the equivalence class that contains the string x .
- $[\text{some logical expression } P]$ describes the equivalence class of strings that satisfy P .

Since \approx_L is an equivalence relation, its equivalence classes constitute a partition of the set Σ^* . So:

- No equivalence class of \approx_L is empty, and
- Every string in Σ^* is in exactly one equivalence class of \approx_L .

What we will see soon is that the equivalence classes of \approx_L correspond exactly to the states of the minimum DFSM that accepts L . So every string in Σ^* will drive that DFSM to exactly one state.

Given some language L , how can we determine \approx_L ? Any pair of strings x and y are related via \approx_L unless there exists some z that could follow them and cause one to be in L and the other not to be. So it helps to begin the analysis by considering simple strings and seeing whether they are distinguishable or not. One way to start this process is to begin lexicographically enumerating the strings in Σ^* and continue until a pattern has emerged.

Example 5.23 Determining \approx_L

Let $\Sigma = \{a, b\}$. Let $L = \{w \in \Sigma^* : \text{every } a \text{ is immediately followed by a } b\}$.

To determine the equivalence classes of \approx_L , we begin by creating a first class $[1]$ and arbitrarily assigning ε to it. Now consider a . It is distinguishable from ε since $\varepsilon ab \in L$ but $aab \notin L$. So we create a new equivalence class $[2]$ and put a in it. Now consider b . $b \approx_L \varepsilon$ since every string is in L unless it has an a that is not followed by a b . Neither of these has an a that could have that problem. So they are both in L as long as their continuation doesn't violate the rule. If their continuation does violate the rule, they are both out. So b goes into $[1]$.

Next we try aa . It is distinguishable from the strings in $[1]$ because the strings in $[1]$ are in L but aa is not. So, consider ε as a continuation string. Take any string in $[1]$ and concatenate ε . The result is still in L . But $aa\varepsilon$ is not in L . We also notice that aa is distinguishable from a , and so cannot be in $[2]$, because a still has a chance to become in L if it is followed by a string that starts with a b . But aa is out, no matter what comes next. We create a new equivalence class $[3]$ and put aa in it. We continue in this fashion until we discover the property that holds of each equivalence class.

The equivalence classes of \approx_L are:

- | | | |
|-----|-----------------------------|-------------------------------------------------------------------------------------|
| [1] | $[\epsilon, b, abb, \dots]$ | [all strings in L]. |
| [2] | $[a, abbbba, \dots]$ | [all strings that end in a and have no prior a that is not followed by a b]. |
| [3] | $[aa, abaa, \dots]$ | [all strings that contain at least one instance of aa]. |

Even this simple example illustrates three key points about \approx_L :

- No equivalence class can contain both strings that are in L and strings that are not. This is clear if we consider the continuation string ϵ . If $x \in L$ then $x\epsilon \in L$. If $y \notin L$ then $y\epsilon \notin L$. So x and y are distinguishable by ϵ .
- If there are strings that would take a DFSM for L to the dead state (in other words, strings that are out of L no matter what comes next), then there will be one equivalence class of \approx_L that corresponds to the dead state.
- Some equivalence class contains ϵ . It will correspond to the start state of the minimal machine that accepts L .

Example 5.24 When More Than One Class Contains Strings in L

Let $\Sigma = \{a, b\}$. Let $L = \{w \in \{a, b\}^* : \text{no two adjacent characters are the same}\}$.

The equivalence classes of \approx_L are:

- | | | |
|-----|-----------------------------|-----------------------------------------------------------------------------------|
| [1] | $[\epsilon]$ | $[\epsilon]$. |
| [2] | $[a, aba, ababa, \dots]$ | [all nonempty strings that end in a and have no identical adjacent characters]. |
| [3] | $[b, ab, bab, abab, \dots]$ | [all nonempty strings that end in b and have no identical adjacent characters]. |
| [4] | $[aa, abaa, ababb\dots]$ | [all strings that contain at least one pair of identical adjacent characters]. |

From this example, we make one new observation about \approx_L :

- While no equivalence class may contain both strings that are in L and strings that are not, there may be more than one equivalence class that contains strings that are in L . For example, in this last case, all the strings in classes [1], [2], and [3] are in L . Only those that are in [4], which corresponds to the dead state, are not in L . That is because of the structure of L : any string is in L until it violates the rule, and then it is hopelessly out.

Does \approx_L always have a finite number of equivalence classes? It has in the two examples we have considered so far. But let's consider another one.

Example 5.25 \approx_L for A^nB^n

Let $\Sigma = \{a, b\}$. Let $L = A^nB^n = \{a^n b^n : n \geq 0\}$.

We can begin constructing the equivalence classes of \approx_L :

- | | |
|-----|----------------|
| [1] | $[\epsilon]$. |
| [2] | $[a]$. |
| [3] | $[aa]$. |
| [4] | $[aaa]$. |

But we seem to be in trouble. Each new string of a 's has to go in an equivalence class distinct from the shorter strings because each string requires a different continuation string in order to become in L . So the set of equivalence classes of \approx_L must include at least all of the following classes:

$$\{[n] : n \text{ is a positive integer and } [n] \text{ contains the single string } a^{n-1}\}$$

Of course, classes that include strings that contain b 's are also required.

So, if $L = A^n B^n$, then \approx_L has an infinite number of equivalence classes. This should come as no surprise. $A^n B^n$ is not regular, as we will prove in Chapter 8. If the equivalence classes of \approx_L are going to correspond to the states of a machine to accept L , then there will be a finite number of equivalence classes precisely in case L is regular.

We are now ready to talk about DFSMs and to examine the relationship between \approx_L and any DFSM that accepts L . To help do that we will say that a state q of a DFSM M *contains* the set of strings s such that M , when started in its start state, lands in q after reading s .

Theorem 5.4 \approx_L Imposes a Lower Bound on the Minimum Number of States of a DFSM for L

Theorem: Let L be a regular language and let $M = (K, \Sigma, \delta, s, A)$ be a DFSM that accepts L . The number of states in M is greater than or equal to the number of equivalence classes of \approx_L .

Proof: Suppose that the number of states in M were less than the number of equivalence classes of \approx_L . Then, by the pigeonhole principle, there must be at least one state q that contains strings from at least two equivalence classes of \approx_L . But then M 's future behavior on those strings will be identical, which is not consistent with the fact that they are in different equivalence classes of \approx_L . ■

So now we know a lower bound on the number of states that are required to build an FSM to accept a language L . But is it always possible to find a DFSM M such that $|K_M|$ is exactly equal to the number of equivalence classes of \approx_L ? The answer is yes.

Theorem 5.5 There Exists a Unique Minimal DFSM for Every Regular Language

Theorem: Let L be a regular language over some alphabet Σ . Then there is a DFSM M that accepts L and that has precisely n states where n is the number of equivalence classes of \approx_L . Any other DFSM that accepts L must either have more states than M or it must be equivalent to M except for state names.

Proof: The proof is by construction of $M = (K, \Sigma, \delta, s, A)$, where:

- K contains n states, one for each equivalence class of \approx_L .
- $s = [\varepsilon]$, the equivalence class of ε under \approx_L .
- $A = \{[x] : x \in L\}$.
- $\delta([x], a) = [xa]$. In other words, if M is in the state that contains some string x , then, after reading the next symbol a , it will be in the state that contains xa .

For this construction to prove the theorem, we must show:

- K is finite. Since L is regular, it is accepted by some DFSM M' . M' has some finite number of states m . By Theorem 5.4, $n \leq m$. So K is finite.
- δ is a function. In other words, it is defined for all (state, input) pairs and it produces, for each of them, a unique value. The construction defines a value of δ for all (state, input) pairs. The fact that the construction guarantees a unique such value follows from the definition of \approx_L .
- $L = L(M)$. In other words, M does in fact accept the language L . To prove this, we must first show that $\forall s, t (([\varepsilon], st) \vdash_{M^*} ([s], t))$. In other words, when M starts in its start state and has a string that we are describing as having two parts, s and t , to read, it correctly reads the first part s and lands in the state $[s]$, with t left to read. We do this by induction on $|s|$. If $|s| = 0$ then we have $([\varepsilon], \varepsilon) \vdash_{M^*} ([\varepsilon], t)$, which is true since M simply makes zero moves. Assume that the claim is true if $|s| = k$. Then we consider what happens when $|s| = k+1$. $|s| \geq 1$, so we can let $s = yc$ where $y \in \Sigma^*$ and $c \in \Sigma$. We have:

/* M reads the first k characters:
 $([\varepsilon], yct) \vdash_{M^*} ([y], ct)$ (induction hypothesis, since $|y| = k$).
 /* M reads one more character:

$$\begin{array}{ll}
([y], ct) \mid_{-M^*} ([yc], t) & \text{(definition of } \delta_M \text{).} \\
/* \text{ Combining those two, after } M \text{ has read } k+1 \text{ characters:} & \\
([\varepsilon], yct) \mid_{-M^*} ([yc], t) & \text{(transitivity of } \mid_{-M^*} \text{).} \\
([\varepsilon], st) \mid_{-M^*} ([s], t) & \text{(definition of } s \text{ as } yc \text{).}
\end{array}$$

Now let t be ε . (In other words, we are examining M 's behavior after it reads its entire input string.) Let s be any string in Σ^* . By the claim we just proved, $([\varepsilon], s) \mid_{-M^*} ([s], \varepsilon)$. M will accept s iff $[s] \in A$, which, by the way in which A was constructed, it will be if the strings in $[s]$ are in L . So M accepts precisely those strings that are in M .

- There exists no smaller machine $M\#$ that also accepts L . This follows directly from Theorem 5.4, which says that the number of equivalence classes of \approx_L imposes a lower bound on the number of states in any DFSM that accepts L .
- There is no different machine $M\#$ that also has n states and that accepts L . Consider any DFSM $M\#$ with n states. We show that either $M\#$ is identical to M (up to state names) or $L(M\#) \neq L(M)$.

Since we do not care about state names, we can standardize them. Call the start state of both M and $M\#$ state 1. Define a lexicographic ordering on the elements of Σ . Number the rest of the states in both M and $M\#$ as follows:

Until all states have been numbered do:

- Let q be the lowest numbered state from which there are transitions that lead to an as yet unnumbered state.
- List the transitions that lead out from q to any unnumbered state. Sort those transitions lexicographically by the symbol on them.
- Go through the sorted transitions (q, a, p) , in order, and, for each, assign the next unassigned number to state p .

Note that $M\#$ has n states and there are n equivalence classes of \approx_L . Since none of those equivalence classes is empty (by the definition of equivalence classes), $M\#$ either wastes no states (i.e., every state contains at least one string) or, if it does waste any states, it has at least one state that contains strings in different equivalence classes of \approx_L . If the latter, then $L(M\#) \neq L$. So we assume the former. Now suppose that $M\#$ is different from M . Then there would have to be at least one state q and one input symbol c such that M has a transition (q, c, r) and $M\#$ has a transition (q, c, t) and $r \neq t$. Call the set of strings that r contains $[r]$. Since $M\#$ has no unused states (i.e., states that contain no strings), by the pigeonhole principle, $M\#$'s transition (q, c, t) must send some string s in $[r]$ to a state, t , that also contains strings that are not in $[r]$. All strings in $[t]$ will then share all futures with s . But s is distinguishable from the strings in $[t]$. If two strings that are distinguishable with respect to L share all futures in $M\#$, then $L(M\#) \neq L$. Contradiction. ■

The construction that we used to prove Theorem 5.5 is useful in its own right: We can use it, if we know \approx_L , to construct a minimal DFSM for L .

Example 5.26 Building a Minimal DFSM from \approx_L

We consider again the language of Example 5.24: Let $\Sigma = \{a, b\}$. Let $L = \{w \in \{a, b\}^* : \text{no two adjacent characters are the same}\}$.

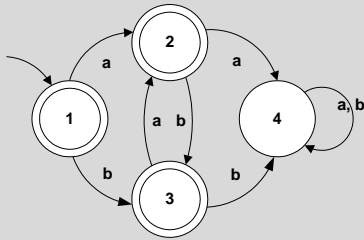
The equivalence classes of \approx_L are:

[1]	$[\varepsilon]$	$\{\varepsilon\}$.
[2]	$[a, aba, ababa, \dots]$	{all nonempty strings that end in a and have no identical adjacent characters}.
[3]	$[b, ab, bab, abab, \dots]$	{all nonempty strings that end in b and have no identical adjacent characters}.
[4]	$[aa, abaa, ababb\dots]$	{all strings that contain at least one pair of identical adjacent characters; these strings are not in L , no matter what comes next}.

We build a minimal DFSM M to accept L as follows:

- The equivalence classes of \approx_L become the states of M .

- The start state is $[\varepsilon] = [1]$.
- The accepting states are all equivalence classes that contain strings in L , namely $[1]$, $[2]$, and $[3]$.
- $\delta([x], a) = [xa]$. So, for example, equivalence class $[1]$ contains the string ε . If the character a follows ε , the resulting string, a , is in equivalence class $[2]$. So we create a transition from $[1]$ to $[2]$ labeled a . Equivalence class $[2]$ contains the string a . If the character b follows a , the resulting string, ab , is in equivalence class $[3]$. So we create a transition from $[2]$ to $[3]$ labeled b . And so forth.



The fact that it is always possible to construct a minimum DFSM M to accept any language L is good news. As we will see later, the fact that that minimal DFSM is unique up to state names is also useful. In particular, we will use it as a basis for an algorithm that checks two DFSMs to see if they accept the same languages. The theorem that we have just proven is also useful because it gives us an easy way to prove the following result, which goes by two names, Nerode's Theorem and the Myhill-Nerode Theorem:

Theorem 5.6 Myhill-Nerode Theorem

Theorem: A language is regular iff the number of equivalence classes of \approx_L is finite.

Proof: We do two proofs to show the two directions of the implication:

L regular \rightarrow the number of equivalence classes of \approx_L is finite: If L is regular, then there exists some DFSM M that accepts L . M has some finite number of states m . By Theorem 5.4, the cardinality of $\approx_L \leq m$. So the number of equivalence classes of \approx_L is finite.

The number of equivalence classes of \approx_L is finite $\rightarrow L$ regular: If the number of equivalence classes of \approx_L is finite, then the construction that was described in the proof of Theorem 5.5 will build a DFSM that accepts L . So L must be regular. ■

The Myhill-Nerode Theorem gives us our first technique for proving that a language L , such as A^nB^n , is not regular. It suffices to show that \approx_L has an infinite number of equivalence classes. But using the Myhill-Nerode Theorem rigorously is difficult. In Chapter 8, we will introduce other methods that are harder to use incorrectly.

5.7.2 Minimizing an Existing DFSM

Now suppose that we already have a DFSM M that accepts L . In fact, possibly M is the only definition we have of L . In this case, it makes sense to construct a minimal DFSM to accept L by starting with M rather than with \approx_L . There are two approaches that we could take to constructing a minimization algorithm:

1. Begin with M and collapse redundant states, getting rid of one at a time until the resulting machine is minimal.
2. Begin by overclustering the states of L into just two groups, accepting and nonaccepting. Then iteratively split those groups apart until all the distinctions that L requires have been made.

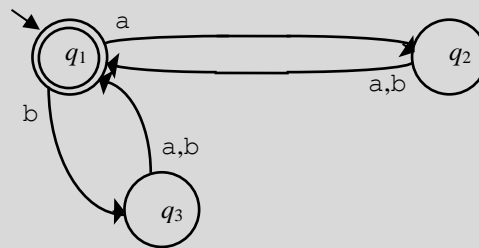
Both approaches work. We will present an algorithm that takes the second one.

Our goal is to end up with a minimal machine in which all equivalent states of M have been collapsed. In order to do that, we need a precise definition of what it means for two states to be equivalent (and thus collapsible). We will use the following:

We will say that two states q and p in M are **equivalent**, which we will write $q \equiv p$, iff for all strings $w \in \Sigma^*$, either w drives M to an accepting state from both q and p or it drives M to a rejecting state from both q and p . In other words, no matter what continuation string comes next, M behaves identically from both states. Note that \equiv is an equivalence relation over states, so it will partition the states of M into a set of equivalence classes.

Example 5.27 A Nonminimal DFSM with Two Equivalent States

Let $\Sigma = \{a, b\}$. Let $L = \{w \in \Sigma^* : |w| \text{ is even}\}$. Consider the following FSM that accepts L :



In this machine state $q_2 \equiv$ state q_3 .

For two states q and p to be equivalent, they must yield the same outcome for all possible continuation strings. We can't claim an algorithm for finding equivalent states that works by trying all possible continuation strings since there is an infinite number of them (assuming that Σ is not empty). Fortunately, we can show that it is necessary to consider only a finite subset of them. In particular, we will consider them one character at a time, and quit when considering another character has no effect on the machine we are building.

We define a series of equivalence relations \equiv^n , for values of $n \geq 0$. For any two states p and q , $p \equiv^n q$ iff p and q yield the same outcome for all strings of length n . So:

- $p \equiv^0 q$ iff they behave equivalently when they read ϵ . In other words, if they are both accepting or both rejecting states.
- $p \equiv^1 q$ iff they behave equivalently when they read any string of length 1. In other words, if any single character sends both of them to an accepting state or both of them to a rejecting state. Note that this is equivalent to saying that any single character sends them to states that are \equiv^0 to each other.
- $p \equiv^2 q$ iff they behave equivalently when they read any string of length 2, which they will do if, when they read the first character they land in states that are \equiv^1 to each other. By the definition of \equiv^1 , they will then yield the same outcome when they read the single remaining character.
- And so forth.

We can state this definition concisely as follows. For all $p, q \in K$:

- $p \equiv^0 q$ iff they are both accepting or both rejecting states.
- For all $n \geq 1$, $q \equiv^n p$ iff:
 - $q \equiv^{n-1} p$, and
 - $\forall a \in \Sigma (\delta(p, a) \equiv^{n-1} \delta(q, a))$.

We will define *minDFSM*, a minimization algorithm that takes as its input a DFSM $M = (K, \Sigma, \delta, s, A)$. *minDFSM* will construct a minimal DFSM M' that is equivalent to M . It begins by constructing \equiv^0 , which divides the states of M into at most two equivalence classes, corresponding to A and $K - A$. If M has no accepting states or if all its states are accepting, then there will be only one nonempty equivalence class and we can quit since there is a one-state machine that is equivalent to M . We consider therefore only those cases where both A and $K - A$ are nonempty.

MinDFSM executes a sequence of steps, during which it constructs the sequence of equivalence relations $\equiv^1, \equiv^2, \dots$. To construct \equiv^{k+1} , *minDFSM* begins with \equiv^k . But then it splits equivalence classes of \equiv^k whenever it discovers some pair of states that do not behave equivalently. *MinDFSM* halts when it discovers that \equiv^n is the same as \equiv^{n+1} . Any further steps would operate on the same set of equivalence classes and so would also fail to find any states that need to be split.

We can now state the algorithm:

```

minDFSM(M: DFMSM) =
1.  classes = {A, K-A}.          /* Initially, just two classes of states, accepting and rejecting.

2.  Repeat until a pass at which no change to classes has been made:
    2.1. newclasses =  $\emptyset$ .      /* At each pass, we build a new set of classes, splitting the old
                                   ones as necessary. Then this new set becomes the old set, and
                                   the process is repeated.
    2.2. For each equivalence class e in classes, if e contains more than one state, see if it needs to be split:
        For each state q in e do:  /* Look at each state and build a table of what it does.
                                   Then the tables for all states in the class can be
                                   compared to see if there are any differences that
                                   force splitting.

                                   For each character c in  $\Sigma$  do:
                                   Determine which element of classes q goes to if c is read.
                                   If there are any two states p and q such that there is any character c such that, when c is read, p goes
                                   to one element of classes and q goes to another, then p and q must be split. Create as many
                                   new equivalence classes as are necessary so that no state remains in the same class with a state
                                   whose behavior differs from its. Insert those classes into newclasses.
                                   If there are no states whose behavior differs, no splitting is necessary. Insert e into newclasses.
    2.3. classes = newclasses.

    /* The states of the minimal machine will correspond exactly to the elements of classes at this point.
    We use the notation [q] for the element of classes that contains the original state q.

3.  Return  $M' = (\textit{classes}, \Sigma, \delta, [s_M], \{[q] : \textit{the elements of } q \textit{ are in } A_M\})$ , where  $\delta_{M'}$  is constructed as follows:
        if  $\delta_M(q, c) = p$ , then  $\delta_{M'}([q], c) = [p]$ .

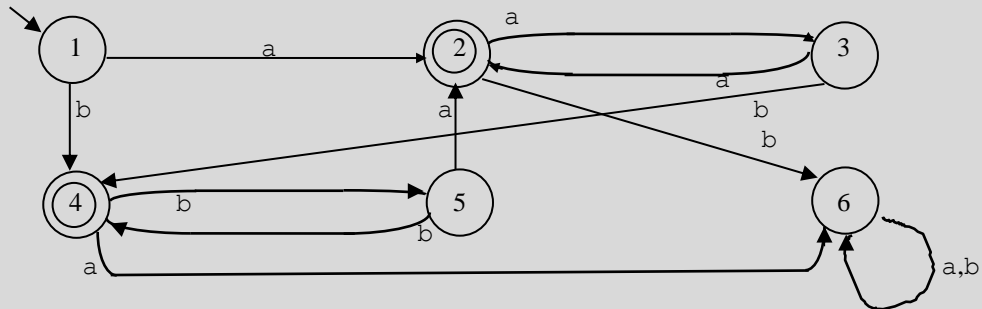
```

Clearly, no class that contains a single state can be split. So, if $|K|$ is k , then the maximum number of times that *minDFSM* can split classes is $k-1$. Since *minDFSM* halts when no more splitting can occur, the maximum number of times it can go through the loop is $k-1$. Thus *minDFSM* must halt in a finite number of steps. M' is the minimal DFMSM that is equivalent to M since:

- M' is minimal: It splits classes and thus creates new states only when necessary to simulate M . and
- $L(M') = L(M)$: The proof of this is straightforward by induction on the length of the input string.

Example 5.28 Using *minDFSM* to Find a Minimal Machine

Let $\Sigma = \{a, b\}$. Let $M =$



We will show the operation of *minDFSM* at each step:

Initially, $classes = \{[2, 4], [1, 3, 5, 6]\}$.

At step 1:

$((2, a), [1, 3, 5, 6])$	$((4, a), [1, 3, 5, 6])$	No splitting required here.	
$((2, b), [1, 3, 5, 6])$	$((4, b), [1, 3, 5, 6])$		
$((1, a), [2, 4])$	$((3, a), [2, 4])$	$((5, a), [2, 4])$	$((6, a), [1, 3, 5, 6])$
$((1, b), [2, 4])$	$((3, b), [2, 4])$	$((5, b), [2, 4])$	$((6, b), [1, 3, 5, 6])$

There are two different patterns, so we must split into two classes, $[1, 3, 5]$ and $[6]$. Note that, although $[6]$ has the same behavior as $[2, 4]$ after reading a single character, it cannot be combined with $[2, 4]$ because they do not share behavior after reading no characters.

$Classes = \{[2, 4], [1, 3, 5], [6]\}$.

At step 2:

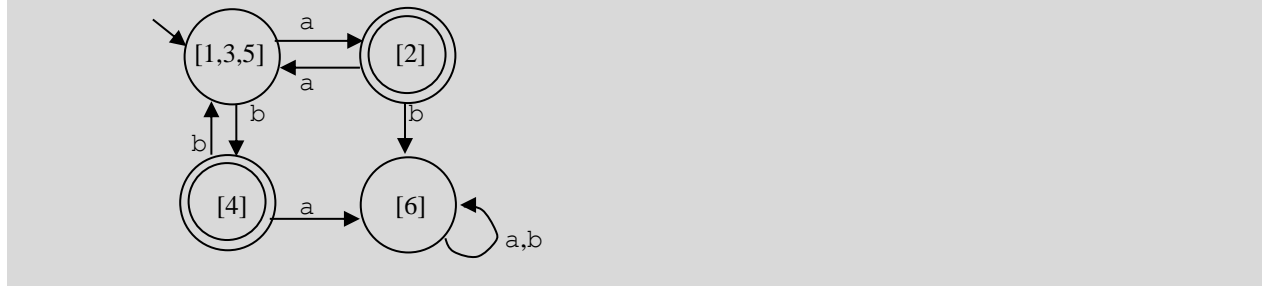
$((2, a), [1, 3, 5])$	$((4, a), [6])$	These two must be split.	
$((2, b), [6])$	$((4, b), [1, 3, 5])$		
$((1, a), [2, 4])$	$((3, a), [2, 4])$	$((5, a), [2, 4])$	No splitting required here.
$((1, b), [2, 4])$	$((3, b), [2, 4])$	$((5, b), [2, 4])$	

$Classes = \{[2], [4], [1, 3, 5], [6]\}$.

At step 3:

$((1, a), [2])$	$((3, a), [2])$	$((5, a), [2])$	No splitting required here.
$((1, b), [4])$	$((3, b), [4])$	$((5, b), [4])$	

So *minDFSM* returns $M' =$



5.8 A Canonical Form for Regular Languages

A *canonical form* for some set of objects C assigns exactly one representation to each class of “equivalent” objects in C . Further, each such representation is distinct, so two objects in C share the same representation iff they are “equivalent” in the sense for which we define the form.

The ordered binary decision diagram (OBDD) is a canonical form for Boolean expressions that makes it possible for model checkers to verify the correctness of very large concurrent systems and hardware circuits. \mathfrak{R} 612.

Suppose that we had a canonical form for FSMs with the property that two FSMs share a canonical form iff they accept the same language. Further suppose that we had an algorithm that on input M , constructed M ’s canonical form. Then some questions about FSMs would become easy to answer. For example, we could test whether two FSMs are equivalent (i.e., they accept the same language). It would suffice to construct the canonical form for each of them and test whether the two forms are identical.

The algorithm *minDFSM* constructs, from any DFSM M , a minimal machine that accepts $L(M)$. By Theorem 5.5, all minimal machines for $L(M)$ are identical except possibly for state names. So, if we could define a standard way to name states, we could define a canonical machine to accept $L(M)$ (and thus any regular language). The following algorithm does this by using the state-naming convention that we described in the proof of Theorem 5.5:

buildFSMcanonicalform(M : FSM) =

1. $M' = \text{ndfsmtodfsm}(M)$.
2. $M\# = \text{minDFSM}(M')$.
3. Create a unique assignment of names to the states of $M\#$ as follows:
 - 3.1. Call the start state q_0 .
 - 3.2. Define an order on the elements of Σ .
 - 3.3. Until all states have been named do:

Select the lowest numbered named state that has not yet been selected. Call it q .
 Create an ordered list of the transitions out of q by the order imposed on their labels.
 Create an ordered list of the as yet unnamed states that those transitions enter by doing the following: if the first transition is (q, c_1, p_1) , then put p_1 first. If the second transition is (q, c_2, p_2) and p_2 is not already on the list, put it next. If it is already on the list, skip it. Continue until all transitions have been considered. Remove from the list any states that have already been named.

Name the states on the list that was just created: Assign to the first one the name q_k , where k is the smallest index that hasn’t yet been used. Assign the next name to the next state and so forth until all have been named.
4. Return $M\#$.

Given two FSMs M_1 and M_2 , *buildFSMcanonicalform*(M_1) = *buildFSMcanonicalform*(M_2) iff $L(M_1) = L(M_2)$. We’ll see, in Section 9.1.4 one important use for this canonical form: it provides the basis for a simple way to test whether an FSM accepts any strings or whether two FSMs are equivalent.

5.9 Finite State Transducers ✦

So far, we have used finite state machines as language recognizers. All we have cared about, in analyzing a machine M , is whether or not M ends in an accepting state. But it is a simple matter to augment our finite state model to allow for output at each step of a machine's operation. Often, once we do that, we may cease to care about whether M actually accepts any strings. Many finite state transducers are loops that simply run forever, processing inputs.

One simple kind of finite state transducer associates an output with each state of a machine M . That output is generated whenever M enters the associated state. Deterministic finite state transducers of this sort are called Moore machines, after their inventor Edward Moore. A **Moore machine** M is a seven-tuple $(K, \Sigma, O, \delta, D, s, A)$, where:

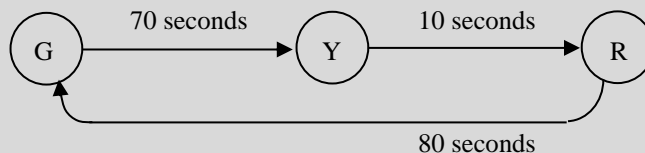
- K is a finite set of states,
- Σ is an input alphabet,
- O is an output alphabet,
- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states (although for some applications this designation is not important),
- δ is the transition function. It is function from $(K \times \Sigma)$ to (K) , and
- D is the display or output function. It is a function from (K) to (O^*) .

A Moore machine M computes a function $f(w)$ iff, when it reads the input string w , its output sequence is $f(w)$.

Example 5.29 A Typical United States Traffic Light

Consider the following controller for a single direction of a very simple US traffic light (which ignores time of day, traffic, the need to let emergency vehicles through, etc.). We will also ignore the fact that practical controller has to manage all directions for a particular intersection. In Exercise 5.16), we will explore removing some of these limitations.

The states in this simple controller correspond to the light's colors: green, yellow and red. Note that the definition of the start state is arbitrary. There are three inputs, all of which are elapsed time.



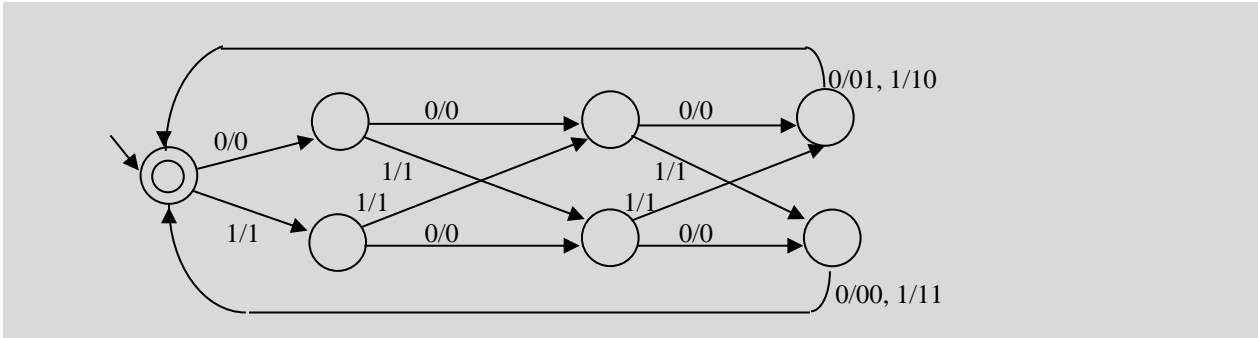
A different definition for a deterministic finite state transducer permits each machine to output any finite sequence of symbols as it makes each transition (in other words, as it reads each symbol of its input). FSMs that associate outputs with transitions are called Mealy machines, after their inventor George Mealy. A **Mealy machine** M is a six-tuple $(K, \Sigma, O, \delta, s, A)$, where:

- K is a finite set of states,
- Σ is an input alphabet,
- O is an output alphabet,
- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states, and
- δ is the transition function. It is function from $(K \times \Sigma)$ to $(K \times O^*)$.

A Mealy machine M computes a function $f(w)$ iff, when it reads the input string w , its output sequence is $f(w)$.

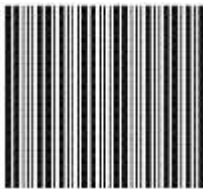
Example 5.30 Generating Parity Bits

The following Mealy machine adds an odd parity bit after every four binary digits that it reads. We will use the notation a/b on an arc to mean that the transition may be followed if the input character is a . If it is followed, then the string b will be generated.



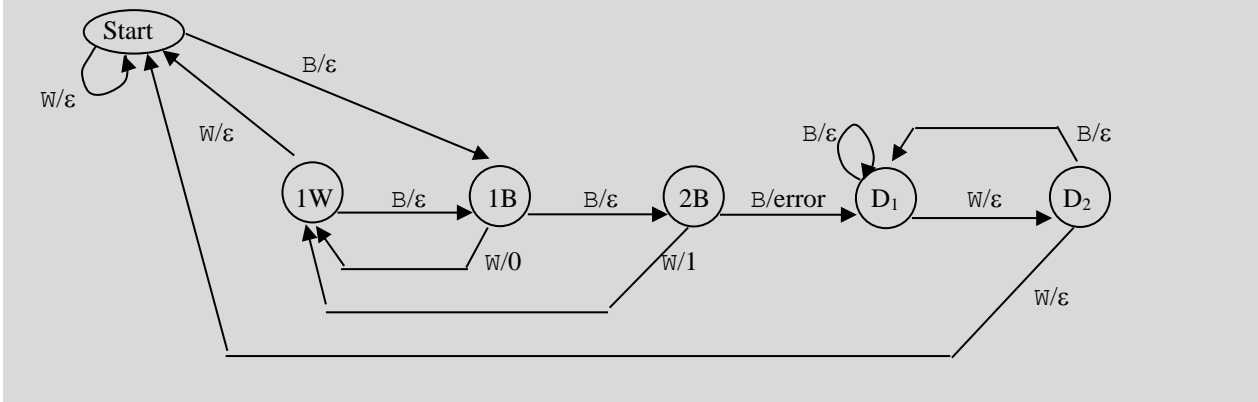
Digital circuits can be modeled as transducers using either Moore or Mealy machines. © 800.

Example 5.31 A Bar Code Reader



Bar codes are ubiquitous. We consider here a simplification: a bar code system that encodes just binary numbers. Imagine an individual bar code as being composed of columns, each of the same width. A column can be either white or black. If two black columns occur next to each other, it will look to us like a single, wide, black column, but the reader will see two adjacent black columns of the standard width. The job of the white columns is to delimit the black ones. A single black column encodes 0. A double black column encodes 1.

We can build a finite state transducer to read such a bar code and output a string of binary digits. We'll represent a black bar with the symbol B and a white bar with the symbol w . The input to the transducer will be a sequence of those symbols, corresponding to reading the bar code left to right. We'll assume that every correct bar code starts with a black column, so white space ahead of the first black column is ignored. We'll also assume that after every complete bar code there are at least two white columns. So the reader should, at that point, reset to be ready to read the next code. If the reader sees three or more black columns in a row, it must indicate an error and stay in its error state until it is reset by seeing two white columns.



Interpreters for finite state transducers can be built using techniques similar to the ones that we used in Section 5.6 to build interpreters for finite state machines.

5.10 Bidirectional Transducers

A process that reads an input string and constructs a corresponding output string can be described in a variety of different ways. Why should we choose the finite state transducer model? One reason is that it provides a declarative, rather than a procedural, way to describe the relationship between inputs and outputs. Such a declarative model can then be run in two directions. For example:

- To read an English text requires transforming a word like “liberties” into the root word “liberty” and the affix PLURAL. To generate an English text requires transforming a root word like “liberty” and the semantic marker “PLURAL” into the surface word “liberties”. If we could specify, in a single declarative model, the relationship between surface words (the ones we see in text) and underlying root words and affixes, we could use it for either application.

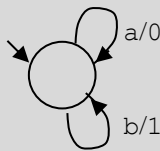
The facts about English spelling rules and morphological analysis can be described with a bidirectional finite state transducer. © 739.

- The Soundex system, described below in Example 5.33, groups names that sound alike. To create the Soundex representation of a name requires a set of rules for mapping the spelling of the name to a unique four character code. To find other names that sound like the one that generated a particular code requires running those same rules backwards.
- Many things we call translators need to run in both directions. For example, consider translating between Roman numerals Ⅲ and Arabic ones.

If we expand the definition of a Mealy machine to allow nondeterminism, than any of these bidirectional processes can be represented. A nondeterministic Mealy machine can be thought of as defining a relation between one set of strings (for example, English surface words) and a second set of strings (for example, English underlying root words, along with affixes). It is possible that we will need a machine that is nondeterministic in one or both directions because the relationship between the two sets may not be able to be described as a function.

Example 5.32 Letter Substitution

When we define a regular language, it doesn’t matter what alphabet we use. Anything that is true of a language L defined over the alphabet $\{a, b\}$ will also be true of the language L' that contains exactly the strings in L except that every a has been replaced by a 0 and every b has been replaced by a 1 . We can build a simple bidirectional transducer that can convert strings in L to strings in L' and vice versa.



Of course, the real power of bidirectional finite state transducers comes from their ability to model more complex processes.

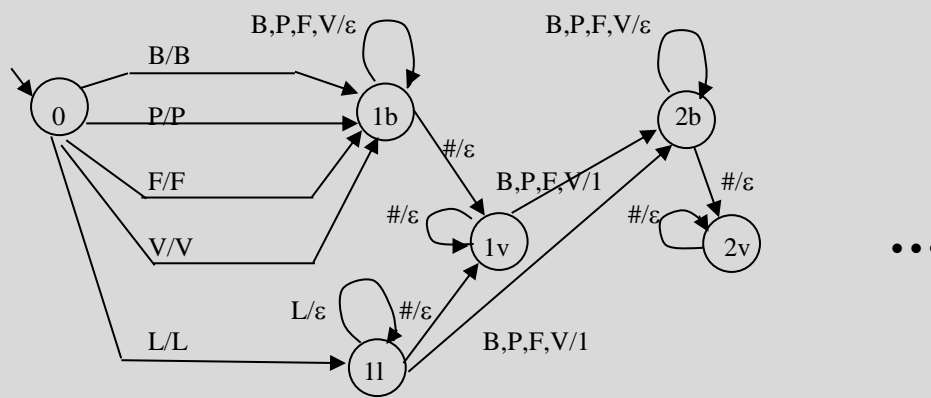
Example 5.33 Soundex: A Way to Find Similar Sounding Names

People change the spelling of their names. Sometimes the spelling was changed for them when they immigrated to a country with a different language, a different set of sounds, and maybe a different writing system. For various reasons, one might want to identify other people to whom one is related. But because of spelling changes, it isn’t sufficient simply to look for people with exactly the same last name. The Soundex Ⅲ system was patented by Margaret O’Dell and Robert C. Russell in 1918 as a solution to this problem. The system maps any name to a four character code that is derived from the original name but that throws away details of the sort that often get perturbed as names evolve. So, to find related names, one can run the Soundex transducer in one direction, from a starting name to its Soundex code and then, in the other direction, from the code to the other names that share that code. For example, if we start with the name Kaylor, we will produce the Soundex code K460. If we then use that code and run the transducer backwards, we can generate the names Kahler, Kaler, Kaylor, Keeler, Kellar, Kelleher, Keller, Kelliher, Kilroe, Kilroy, Koehler, Kohler, Koller, and Kyler.


The Soundex system is described by the following set of rules for mapping from a name to a Soundex code:

1. If two or more adjacent letters (including the first in the name) would map to the same number if rule 3.1 were applied to them, remove all but the first in the sequence.
2. The first character of the Soundex code will be the first letter of the name.
3. For all other letters of the name do:
 - 3.1. Convert the letters B, P, F, V, C, S, G, J, K, Q, X, Z, D, T, L, M, N, and R to numbers using the following correspondences:
 - B, P, F, V = 1.
 - C, S, G, J, K, Q, X, Z = 2.
 - D, T = 3.
 - L = 4.
 - M, N = 5.
 - R = 6.
 - 3.2 Delete all instances of the letters A, E, I, O, U, Y, H, and W.
4. If the string contains more than three numbers, delete all but the leftmost three.
5. If the string contains fewer than three numbers, pad with 0's on the right to get three.

Here's an initial fragment of a finite-state transducer that implements the relationship between names and Soundex codes. The complete version of this machine can input a name and output a code by interpreting each transition labeled x/y as saying that the transition can be taken on input x and it will output y . Going the other direction, it can input a code and output a name if it interprets each transition the other way: on input y , take the transition and output x . To simplify the diagram, we've used two conventions: The symbol # stands for any one of the letters A, E, I, O, U, Y, H, or W. And a label of the form $x, y, z/a$ is a shorthand for three transitions labeled x/a , y/a , and z/a . Also, the states are named to indicate how many code symbols have been generated/read.



Notice that in one direction (from names to codes), this machine operates deterministically. But, because information is lost in that direction, if we run the machine in the direction that maps from code to name, it becomes nondeterministic. For example, the ϵ -transitions can be traversed any number of times to generate vowels that are not represented in the code. Because the goal, in running the machine in the direction from code to names is to generate actual names, the system that does this is augmented with a list of names found in US census reports. It can then follow paths that match those names.

The Soundex system was designed for the specific purpose of matching names in United States census data from the early part of the twentieth century and before. Newer systems, such as Phonix and Metaphone , are attempts to solve the more general problem of identifying words that sound similar to each other. Such systems are used in a variety of applications, including ones that require matching a broader range of proper names (e.g., genealogy and white pages look up) as well as more general word matching tasks (e.g., spell checking).

5.11 Stochastic Finite Automata: Markov Models and HMMs

Most of the finite state transducers that we have considered so far are deterministic. But that is simply a property of the kinds of applications to which they are put. We do not want to live in a world of nondeterministic traffic lights or phone switching circuits. So we typically design controllers (i.e., machines that run things) to be deterministic. For some applications though, nondeterminism can be useful. For example, it can add entertainment value.

Nondeterministic (possibly stochastic) FSMs can form the basis of video games. © 789.

But now consider problems like the name-evolution one we just discussed. Now we are not attempting to build a controller that drives the world. Instead we are trying to build a model that describes and predicts a world that we are not in control of. Nondeterministic finite state models are often very useful tools in solving such problems. And typically, although we do not know enough to predict with certainty how the behavior of the model will change from one step to the next (thus the need for nondeterminism), we do have some data that enable us to estimate the probability that the system will move from one state to the next. In this section, we explore the use of nondeterministic finite state machines and transducers that have been augmented with probabilistic information.

5.11.1 Markov Models

A Markov model is an NDFSM in which the state at each step can be predicted by a probability distribution associated with the current state. Steps usually correspond to time intervals, but they may correspond to any ordered discrete sequence. In essence we replace transitions labeled with input symbols by transitions labeled with probabilities. The usual definition of a Markov model is that its behavior at time t depends only on its state at time $t-1$ (although higher-order models may allow any finite number of past states to play a role). Of course, if we eliminate an input sequence, that is exactly the property that characterizes an FSM.

Markov models have been used in music composition. © 776. They have also been used to model the generation of many other sorts of content, including Web pages.

Formally a *Markov model* is a triple $M = (K, \pi, A)$, where:

- K is a finite set of states,
- π is a vector that contains the initial probabilities of each of the states, and
- A is a matrix that represents the transition probabilities. $A[p, q] = \Pr(\text{state } q \text{ at time } t \mid \text{state } p \text{ at time } t-1)$. In other words $A[p, q]$ is the probability that, if M is in state p , it will go to state q next.

Some definitions specify a unique start state, but this definition is more general. If there is a unique start state, then its initial probability is 1 and the initial probabilities of all other states are 0.

Notice that we have not mentioned any output alphabet. We will assume that the output at each step is simply the name of the state of the machine at that step. The sequence of outputs produced by a Markov model is often called a *Markov chain*.

The link structure of the World Wide Web can be modeled as a Markov chain, where the states correspond to Web pages and the probabilities describe the likelihood, in a random walk, of going from one page to the next. Google's PageRank is based on the limits of those probabilities.

Given a Markov model that describes some random process, we can answer either of the following questions:

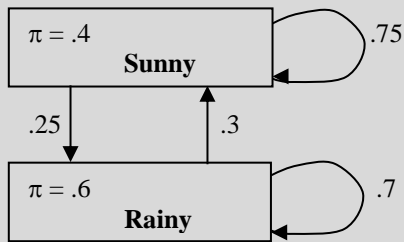
- What is the probability that we will observe a particular sequence $s_1 s_2 \dots s_n$ of states? We can compute this as follows, using the probability that s_1 is the start state and then multiplying by the probabilities of each of the transitions:

$$\Pr(s_1 s_2 \dots s_n) = \pi[s_1] \cdot \prod_{i=2}^n A[s_{i-1}, s_i].$$

- If the process runs for an arbitrarily long sequence of steps, what is likely to be the result? More specifically, for each state in the system, what is the probability that the system will land in that state?

Example 5.34 A Simple Markov Model of the Weather

Suppose that we have the following model for the weather where we live. This model assumes that the weather on day t is influenced only by the weather on day $t-1$.



We are considering a five day camping trip and want to know the probability of five sunny days in a row. So we want to know the probability of the sequence Sunny Sunny Sunny Sunny Sunny. The model tells us that it is:

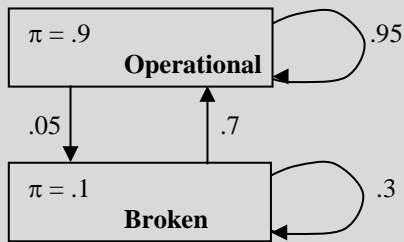
$$.4 \cdot (.75)^4 = .1266$$

Or we could ask, given that it's sunny today, what is the probability that, if we leave now, it will stay sunny for four more days. Now we assume that the model starts in state Sunny, so we compute:

$$(.75)^4 = .316$$

Example 5.35 A Simple Markov Model of System Performance

Markov models are used extensively to model the performance of complex systems of all kinds, including computers, electrical grids, and manufacturing plants. While real models are substantially more complex, we can see how these models work by taking Example 5.34 and renaming the states:



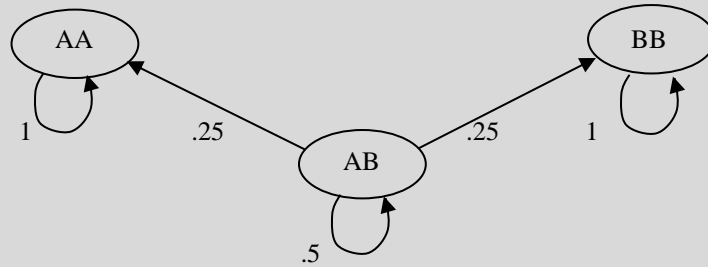
To make it a bit more realistic, we've changed the probabilities so that they describe a system that actually works most of the time. We'll also use smaller time intervals, say seconds. Now we might ask, "Given that the system is now up, what is the probability that the system will stay up for an hour (i.e., for 3600 time steps). The (possibly surprising) answer is:

$$.95^{3600} = 6.3823 \cdot 10^{-81}$$

Example 5.36 Population Genetics

In this example we consider a simple problem in population genetics. For a survey of the biological concepts behind this example, see § 727. Suppose that we are interested in the effect of inbreeding on the gene pool of a diploid organism (an organism, such as humans, in which each individual has two copies of each gene). Consider the following simple model of the inheritance of a single gene with two alleles (values): A and B. There are potentially three kinds of individuals in the population: the AA organisms, the BB organisms, and the AB organisms. Because we are studying inbreeding, we'll make the assumption that individuals always mate with others who are genetically similar to themselves and so possess the same gene pair.

To simplify our model, we will assume that one couple mates, has two children, and dies. So we can think of each individual as replacing itself and then dying. We can build the following Markov model of a chain of descendants. Each step now corresponds to a generation.



AA pairs can produce only AA offspring. BB pairs can produce only BB offspring. But what about AB pairs? What is their fate? We can answer this question by considering the probability that the model, if it starts in state AB and runs for some number of generations, will land in state AB. That probability is $.5^n$, where n is the number of generations. As n grows, that number approaches 0. We show how quickly it does so in the following table:

n	Pr(AB)
1	.5
5	.03125
10	.0009765625
100	$7.8886 \cdot 10^{-31}$

After only 10 generations, very few heterozygous individuals (i.e., possessing two different alleles) remain. After 100 generations, almost none do. If there is survival advantage in being heterozygous, this could be a disaster for the population. The disaster can be avoided, of course, if individuals mate with genetically different individuals.

Where do the probabilities in a Markov model come from? In some simple cases, they may be computed by hand and added to the system. In most cases, however, they are computed by examining real datasets and discovering the probabilities that best describe those data. So, for example, the probabilities we need for the system performance model of Example 5.35 could be extracted from a log of system behavior over some recent period of time. To see how this can be done, suppose that we have observed the output sequences: T P T Q P Q T and S S P T P Q Q P S T Q P T T P. The correct value for $A[P, Q]$ is the number of times the pair P Q appears in the sequence divided by the total number of times that P appears in any position except the last. Similarly, the correct value for $\pi[P]$ is the total number of times that P is the first symbol in a sequence divided by the total number of sequences. In realistic problem contexts, the models are huge and they evolve over time. There exist more computationally tractable algorithms for updating the probabilities (and, when necessary the states) of such models.

Substantial work has been done on efficient techniques for updating the huge Markov model of the World Wide Web that is used to compute Google's PageRanks . Note here that both the state set (corresponding to the set of pages on the Web) as well as the probabilities (which depend on the link structure of the Web) must be regularly revised.

All of the Markov models we have presented so far have the property that their behavior at step t is a function only of their state at step $t-1$. Such models are called first-order. To build a first-order model with k states requires that we specify k^2 transition probabilities. Now suppose that we wish to describe a situation in which what happens next depends on the previous two states. Or the previous three. Using the same techniques that we used to build a first-order model, we can build models that consider the previous n states for any fixed n . Such models are called n^{th} order Markov models. Notice that an n^{th} order model requires k^{n+1} transition probabilities. But if there are enough data available to train a higher-order model (i.e., to assign appropriate probabilities to all of the required transitions), it may be possible to build a system that quite accurately mimics the behavior of a very complex system.

A third-order Markov model, trained on about half of this book, used word frequencies to generate the text "The Pumping Theorem is a useful way to define a precedence hierarchy for the operators + and *." © 746. A clever application of a higher order Markov model of English is in producing spam that is hard to detect. © 747.

Early work on the use of Markov models for musical composition suggested that models of order four or less tended to create works that seemed random, while models of order seven or more tended to create works that felt just like copies of works on which the model was trained. © 776.

Whenever we build a Markov model to describe a naturally occurring process, there is a sense in which we are using probabilities to hide an underlying lack of understanding that would enable us to build a deterministic model of the phenomenon. So, for example, if we know that our computer system is more likely to crash in the morning than in the evening, that may show up as a pair of different probabilities in a Markov model, even if we have no clue why the time of day affects system performance. Some Markov models that do a pretty good job of mimicking nature may seem silly to us for exactly that reason. The one that generates random English text is a good example of that. But now suppose that we had a model that did a very good job of predicting earthquakes. Although we might rather have a good structural model that tells us why earthquakes happen, a purely statistical, predictive model would be a very useful tool. It is because of cases like this that Markov models can be extremely valuable tools for anyone studying complex systems (be they naturally occurring ones like plate tectonics or engineering artifacts like computer systems).

5.11.2 *Hidden Markov Models*

Now suppose that we are interested in analyzing a system that can be described with a Markov model with one important difference: the states of the system are not directly observable. Instead the model has a separate set of output symbols, which are emitted, with specified probabilities, whenever the system enters one of its now “hidden” states. Now we must base our analysis of the system on an observed sequence of output symbols, from which we can infer, with some probability, the actual sequence of states of the underlying model.

Examples of significant problems that can be described in this way include:

- DNA and protein evolution: A protein is a sequence of amino acids that is manufactured in living organisms according to a DNA blueprint. Mutations that change the blueprint can occur, with the result that one amino acid may be substituted for another, one or more amino acids may be deleted, or one or more additional amino acids may be inserted. When we examine a DNA fragment or a protein, we’d like to be able to reconstruct the evolutionary process so that we can find other proteins that are functionally related to the current one, even though its details may be different. But the process isn’t visible; only its result is.

HMMs are used for DNA and protein sequence alignment in the face of mutations and other kinds of evolutionary change. © 735.

- Speech understanding: When we talk, our mouths map from the sentences we want to say into sequences of sounds. The mapping is complex and nondeterministic since multiple words may map to the same sound, words are pronounced differently as a function of the words before and after them, we all form sounds slightly differently, and so forth. All a listener can hear is the sequence of sounds. (S)he would like to reconstruct the mapping (backwards) in order to determine what words we were attempting to say.

HMMs are used extensively in speech understanding systems. © 754.

- Optical character recognition (OCR) ☒: When we write, our hands map from an idealized symbol to some set of marks on a page. The marks are observable, but the process that generates them isn’t. Imagine that we could describe a probabilistic process corresponding to each symbol that we can write. Then, to interpret the marks, we must select the process that is most likely to have generated the marks we can see.

What is a Hidden Markov Model?

A powerful technique for solving problems such as this is the *hidden Markov model* or *HMM* ☒. An HMM is a nondeterministic finite state transducer that has been augmented with three kinds of probabilistic information:

- Each state is labeled with the probability that the machine will be in that state when it starts.

- Each transition from some state p to some (possibly identical) state q is labeled with the probability that, whenever the machine is in state p , it will go next to state q . We can specify M 's transition behavior completely by defining these probabilities. If it is not possible for M to go from some state p to some other state q , then we simply state the probability of going from p to q as 0.
- Each output symbol c at each state q is labeled with the probability that the machine, if it is in state q , will output c .

Formally, an HMM M is a quintuple (K, O, π, A, B) , where:

- K is a finite set of states,
- O is the output alphabet,
- π is a vector that contains the initial probabilities of each of the states,
- A is a matrix that represents the transition probabilities. $A[p, q] = \Pr(\text{state } q \text{ at time } t \mid \text{state } p \text{ at time } t - 1)$.
- B , sometimes called the confusion matrix, represents the output probabilities. $B[q, o] = \Pr(\text{output } o \mid \text{state } q)$. Note that outputs are associated with states (as in Moore machines).

The name “hidden Markov model” derives from the two key properties of such devices:

- They are Markov models. Their state at time t is a function solely of their state at time $t - 1$.
- The actual progression of the machine from one state to the next is hidden from all observers. Only the machine's output string can be observed.

To use an HMM as the basis for an application program, we typically have to solve some or all of the following problems:

- The **decoding problem**: Given an observation sequence O and an HMM M , discover the path through M that is most likely to have produced O . For example, O might be a string of words that form a sentence. We might have an HMM that describes the structure of naturally occurring English sentences. Each state in M corresponds to a part of speech, such as noun, verb, or adjective. It's not possible to tell, just by looking at O , what sequence of parts of speech generated it, since many words can have more than one part of speech. (Consider, for example, the simple English sentence, “Hit the fly ball.”) But we need to infer the parts of speech (a process called part of speech or POS tagging) before we can parse the sentence. We can do that if we can find the path through the HMM that is the most likely to have generated the observed sentence. This problem can be solved efficiently using a dynamic programming algorithm called the Viterbi algorithm, described below.

HMMs are often used for part of speech tagging. © 741.

Suppose that the sequences that we observe correspond to original sequences that have been altered in some way. The alteration may have been done intentionally (we'll call this “obfuscation”) or it may be the result of a natural phenomenon like evolution or a noisy transmission channel. In either case, if we want to know what the original sequence was, we have an instance of the decoding problem. We seek to find the original sequence that is most likely to have been the one that got transformed into the observed sequence.

In the Internet era, an important application of obfuscation is the generation of spam. If specific words are known to trigger spam filters, they can be altered, by changing vowels, introducing special characters, or whatever, so that they are still recognizable to people but unrecognizable, at least until the next patch, to the spam filters. HMMs can be used to perform “deobfuscation” in an attempt to foil the obfuscators. 📧.

- The **evaluation problem**: Given an observation sequence O and a set of HMMs that describe a collection of possible underlying models, choose the HMM that is most likely to have generated O . For example, O might be a sequence of sounds. We might have one HMM for each of the words that we know. We need to choose the word model that is most likely to have generated O . As another example, consider again the protein problem: Now we have one HMM for each family of related proteins. Given a new sample, we want to find the family to which it is most likely to be related. So we look for the HMM that is most likely to have generated it. This problem

can be solved efficiently using the forward algorithm, which is very similar to the Viterbi algorithm except that it considers all paths through a candidate HMM, rather than just the most likely one.

- The **training problem**: We typically assume, in crafting an HMM M , that the set K of states is built by hand. But where do all the probabilities in π , A , and B come from? Fortunately, there are algorithms that can learn them from a set of training data (i.e., a set of observed output sequences O). One of the most commonly used algorithms is the Baum-Welch algorithm [1], also called the forward-backward algorithm. Its goal is to tune π , A , and B so that the resulting HMM M has the property that, out of all the HMMs whose state set is equal to K , M is the one most likely to have produced the outputs that constitute the training set. Because the states cannot be directly observed (as they can be in a standard Markov model), the training technique that we described in Section 5.11.1 won't work here. Instead, the Baum-Welch algorithm employs a technique called **expectation maximization** or EM. It is an iterative method, so it begins with some initial set of values for π , A , and B . Then it runs the forward algorithm, along with a related backward algorithm, on the training data. The result of this step is a set of probabilities that describe the likelihood that the existing machine, with the current values of π , A , and B , would have output the training set. Using those probabilities, Baum-Welch updates π , A , and B to increase those probabilities. The process continues until no changes to the parameter values can be made.

The Viterbi Algorithm

Given an HMM M and an observed output sequence O , a solution to the decoding problem is the path through M that is most likely to have produced O . One way to find that most likely path is to explore all paths of length $|O|$, keeping track of the accumulated probabilities, and then report the path whose probability is the highest. This approach is straightforward, but may require searching a tree with $|K_M|^{|O|}$ nodes, so the time required may grow exponentially in the length of O .

A more efficient approach uses a dynamic programming technique in which the most likely path of some length, say t , is computed once and then extended by one more step to find the most likely path of length $t + 1$. The Viterbi algorithm uses this approach. It solves the decoding problem by computing, for each step t and for each state q in M :

- the most likely path to q of all the ones that would have generated $O_1 \dots O_t$, and
- the probability of that path.

Once it has done that for each step for which an output was observed, it traces the path backwards. It assumes that the last state is the one at the end of the overall most likely path. The next to the last state is the one that preceded that one on the most likely path, and so forth.


Assume, at each step t , that the algorithm has already considered all paths of length $t-1$ that could have generated $O_1 \dots O_{t-1}$. From those paths, it has selected, for each state p , the most likely path to p and it has recorded the probability of the model taking that path, reaching p , and producing $O_1 \dots O_{t-1}$. We assume further that the algorithm has also recorded, at each state p , the state that preceded p on that most likely path. Before the first output symbol is observed, the probability that the system has reached some state p is simply $\pi(p)$ and there is no preceding state.

Because the model is Markovian, the only thing that affects the probability of the next state is the previous state. In constructing the model, we assumed that prior history doesn't matter (although that may be only an approximation to reality for some problems). So, at step t , we compute, for each state q , the probability that the best path so far that is consistent with $O_1 \dots O_t$ ends in q and outputs the first t observed symbols. We do this by considering each state p that the model could have been in at step $t-1$. We already know the probability that the best path up to step $t-1$ landed in p and produced the observed output sequence. So, to add one more step, we multiply that probability by $A[p, q]$, the probability that the model, if it were in p , would go next to q . But we have one more piece of information: the next output symbol. So, to compute the probability that the model went through p , landed in q , and output the next symbol o , we multiply by $B[p, o]$. Once these numbers have been computed for all possible preceding states p , we choose the most likely one (i.e., the one with the highest score as described above). We record that score at q and we record at q that the most likely predecessor state is the one that produced that highest score.

Although we've described the output function as a function of the state the model is in, we don't actually consider it until we compute the next step, so it may be easier to think of the outputs as associated with the transitions rather than

with the states. In particular, the computation that we have just described will end by choosing the state in which the model is most likely to land just after it outputs the final observed symbol. That last state will not generate any output.

Once all steps have been considered, we can choose the overall most likely path as follows: Consider all states. The model is most likely to have ended in the one that, at the final time step, has the highest score as described above. Call that highest scoring state the last state in the path. Find the state that was marked as immediately preceding that one. Continue backwards to the start state.

We can summarize this process, known as the **Viterbi algorithm** , as follows: Given an observed output sequence O , we will consider each time step between 1 and the length of O . At each such step t , we will set $score(q, t)$ to the highest probability associated with any path of length t that lands M in q , having output the first t symbols in O . We will set $backptr(q, t)$ to the state that immediately preceded q along that best path. Once $score$ and $backptr$ have been computed for each state at each time step t , we can start at the most likely final state and trace backwards to find the sequence $states$ that describes the most likely path through M consistent with O . So the Viterbi algorithm is:

Viterbi(M : Markov model, O : output sequence) =


1. For $t = 0$, for each state q , set $score[q, t]$ to $\pi[q]$.
2. /* Trace forward recording the best path at each step:
For $t = 1$ to $|O|$ do:
 - 2.1. For each state q in K do:
 - For each state p in K that could have immediately preceded q :

$$candidatescore[p] = score[p, t-1] * A[p, q] * B[p, O_t].$$
 - /* Record score along most likely path:

$$score[q, t] = \max_{p \in K} candidatescore[p]$$
 - /* Set q 's $backptr$. The function $argmax$ returns the value of the argument p that produced the maximum value of $candidatescore[p]$:

$$backptr[q, t] = arg \max_{p \in K} candidatescore[p]$$
3. /* Retrieve the best path by going backwards from the most likely last state:
 $states[|O|] =$ the state q with the highest value of $score[q, |O|]$.
For $t = |O| - 1$ to 0 do:
 $states[t] = backptr[states[t+1], t+1]$.
4. Return $states[0 : |O|-1]$. /* Ignore the last state since its output was not observed.

The Forward Algorithm

Now suppose that we want to solve the evaluation problem: given a set of HMMs and an observed output sequence O , decide which HMM had the highest probability of producing O . This problem can be solved with the **forward algorithm** , which is very similar to the Viterbi algorithm except that, instead of finding the single best path through an HMM M , it computes the probability that M could have output O along *any* path. In step 2.1.2, the Viterbi algorithm selects the highest score associated with any one path to q . The forward algorithm, at that point, sums all the scores. The other big difference between the Viterbi algorithm and the forward algorithm is that the forward algorithm does not need to find a particular path. So it will not have to bother maintaining the $backptr$ array. We can state the algorithm as follows:

forward(M : Markov model, O : output sequence) =

1. For $t = 0$, for each state q , set $forward-score[q, t]$ to $\pi[q]$.
2. /* Trace forward recording, at each step, the total probability associated with all paths to each state:
For $t = 1$ to $|O|$ do:
 - 2.1. For each state q in K do:
 - 2.1.1. Consider each state p in K that could have immediately preceded q :

$$candidatescore[p] = forwardscore[p, t-1] * A[p, q] * B[p, O_t].$$

2.1.2. /* Sum scores over all paths:

$$forwardscore[q,t] = \sum_p candidatescore[p]$$

3. /* Find the total probability of going through M along any path, landing in any of M 's states, and emitting O . This is simply the sum of the probability of landing in state 1 having emitted O , plus the probability of landing in state 2 having emitted O , and so forth. So:

$$totalprob = \sum_{q \in K} forwardscore[q, |O|]$$

4. Return $totalprob$.

To solve the evaluation problem, we run the forward algorithm on all of the contending HMMs and return the one with the highest final score.

The Complexity of the Viterbi and the Forward Algorithms

Analyzing the complexity of the Viterbi and the forward algorithms is straightforward. In both cases, the outer loop of step 2 is executed once for each observed output, so $|O|$ times. Within that loop, the computation of $candidatescore$ is done once for each state pair. So if M has k states, it is done k^2 times. The computation of $score/forwardscore$ takes $\mathcal{O}(k)$ steps, as does the computation of $backptr$ in the Viterbi algorithm. The final operation of the Viterbi algorithm (computing the list of states to be returned) takes $\mathcal{O}(|O|)$ steps. The final operation of the forward algorithm (computing the total probability of producing the observed output) takes $\mathcal{O}(k)$ steps. So, in both cases, the total time complexity is $\mathcal{O}(k^2 \cdot |O|)$.

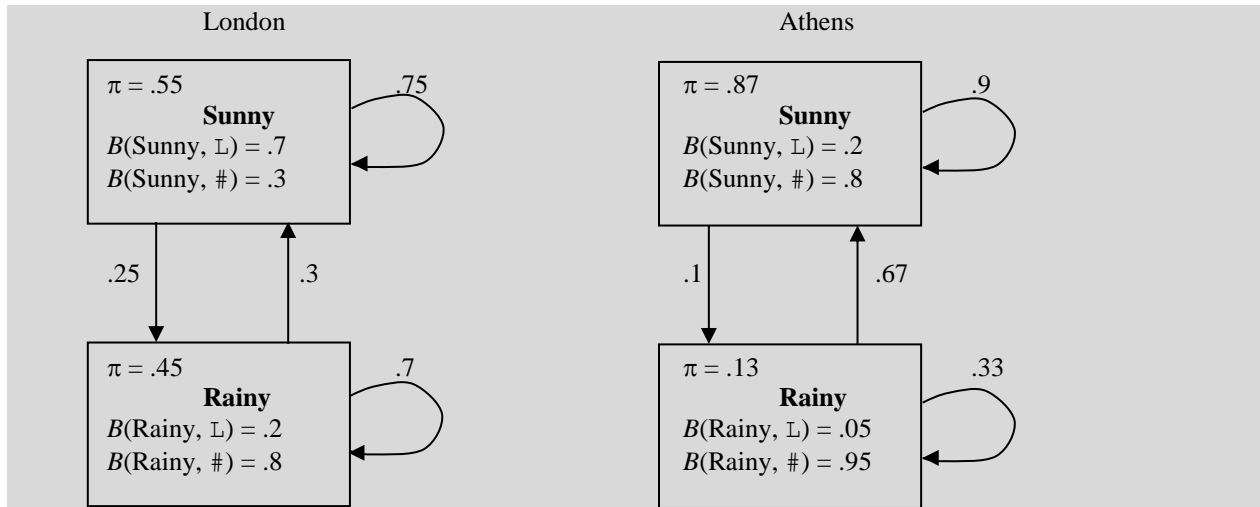
An Example of How These Algorithms Work

The real power of HMMs is in solving complex, real-world problems in which probability estimates can be derived from large datasets. So it is hard to illustrate the effectiveness of HMMs on small problems, but the idea should be clear from the following simple example of the use of the Viterbi algorithm.

Example 5.37 Using the Viterbi Algorithm to Guess the Weather

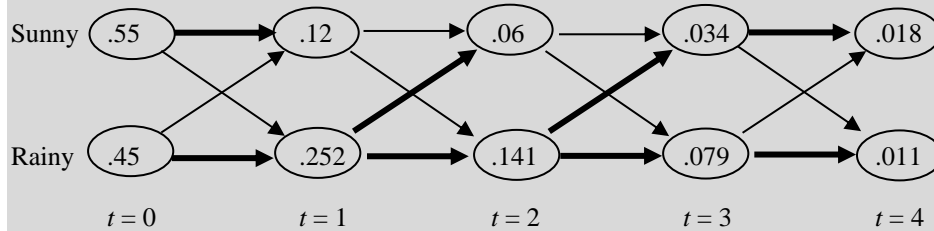
Suppose that you are a state department official in a small country. Each day, you receive a report from each of your consular offices telling you whether or not any of your passports were reported missing that day. You know that the probability of a passport getting lost or stolen is a function of the weather, since people tend to stay inside (and thus manage to keep track of their passports) when the weather is bad. But they tend to go out and thus risk getting their passport lost or stolen if the weather is good. So it amuses you to try to infer the weather in your favorite cities by watching the lost passport reports. We'll use the symbol \perp to mean that a passport was lost and the symbol $\#$ to mean that none was. So, for example, a report for a week might look like $\perp\perp\#\#\perp\#\#\#$.

We'll consider just two cities, London and Athens. We can build an HMM for each. Both HMMs have two states, Sunny and Rainy.



Now suppose that you receive the report ###L from London and you want to find out what the most likely sequence of weather reports was for those days. The Viterbi algorithm will solve the problem.

The easiest way to envision the way that *Viterbi* works is to imagine a lattice, in which each column corresponds to a step and each row corresponds to a state in M :



The number shown at each point (q, t) is the value that *Viterbi* computes for $score[q, t]$. So we can think of *Viterbi* as creating this lattice left to right, and filling in scores as it goes along. The arrows represent possible transitions in M . The heavy arrows indicate the path that is recorded in the matrix *backptr*.

At $t = 0$, the probabilities recorded in *score* are just the initial probabilities, as given in π . So the sum of the values in column 1 is 1. At later steps, the sum is less than 1 because we are considering only the probabilities of paths through M that result in the observed output sequence. Other paths could have produced other output sequences.

At all times $t > 0$, the values for *score* can be computed by considering the probabilities at the previous time (as recorded in *score*), the probabilities of moving from one state to another (as recorded in the matrix A), and the probabilities (recorded in the vector O) of observing the next output symbol. To see how the *Viterbi* algorithm computes those values, let's compute the value of $score[\text{Sunny}, 1]$:

$$\begin{aligned}
 \text{candidate-score}[\text{Sunny}] &= \text{score}[\text{Sunny}, 0] * A[\text{Sunny}, \text{Sunny}] * B[\text{Sunny}, \#] \\
 &= .55 * .75 * .3 \\
 &= .12 \\
 \text{candidate-score}[\text{Rainy}] &= \text{score}[\text{Rainy}, 0] * A[\text{Rainy}, \text{Sunny}] * B[\text{Rainy}, \#] \\
 &= .45 * .3 * .8 \\
 &= .11
 \end{aligned}$$

So $score[\text{Sunny}, 1] = \max(.12, .11) = .12$, and $\text{backptr}(\text{Sunny}, 1)$ is set to Sunny.

Once all the values of *score* have been computed, the final step is to observe that Sunny as the most likely state for M to have reached just prior to generating a fifth output symbol. The state that most likely preceded it is Sunny, so we

report Sunny as the last state to have produced output. Then we trace the backpointers and report that the most likely sequence of weather reports is Rainy, Rainy, Rainy, Sunny.

Now suppose that the fax machine was broken and the reports for last week came in with the city names chopped off the top. You have received the report ###L and you want to know whether it is more likely that it came from London or from Athens. To solve this problem, you use the forward algorithm. You run the output sequence ###L through the London model and through the Athens model, this time computing the total probability (as opposed to just the probability along the best path) of reaching each state from any path that is consistent with the output sequence. The most likely source of this report is the model with the highest final probability.

5.12 Finite Automata, Infinite Strings: Büchi Automata ✦

So far, we have considered, as input to our machines, only strings of finite length. Thus we have focused on problems for which we expect to write programs that read an input, compute a result, and halt. Many problems are of that sort, but some are not. For example, consider:

- an operating system.
- an air traffic control system.
- a factory process control system.

Ideally, such systems never halt. They should accept an infinite string of inputs and continue to function. Define Σ^ω to be the set of infinite length strings drawn from the alphabet Σ . For the rest of this discussion, define a language to be a set of such infinite-length strings.

To model the behavior of processes that do not halt, we can extend our notion of an NDFSM to define a machine whose inputs are elements of Σ^ω . Such machines are sometimes called ω -automata (or omega automata).

We'll define one particular kind of ω -automaton: A *Büchi automaton* is a quintuple $(K, \Sigma, \Delta, S, A)$, where:

- K is a finite set of states,
- Σ is the input alphabet,
- $S \subseteq K$ is a set of start states,
- $A \subseteq K$ is the set of accepting states, and
- Δ is the transition relation. It is a finite subset of:

$$(K \times \Sigma) \times K.$$

Note that, unlike NDFSMs, Büchi automata may have more than one start state. Note also that the definition of a Büchi automaton does not allow ϵ -transitions.

We define configuration, initial configuration, yields-in-one-step, and yields exactly as we did for NDFSMs. A *computation* of a Büchi automaton M is an infinite sequence of configurations C_0, C_1, \dots such that:

- C_0 is an initial configuration, and
- $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \dots$

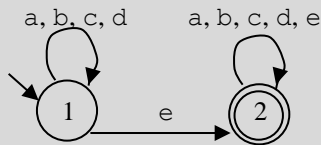
But now we must define what it means for a Büchi automaton M to accept a string. We can no longer define acceptance by the state of M when it runs out of input, since it won't. Instead, we'll say that M accepts a string $w \in \Sigma^\omega$ iff, in at least one of its computations, there is some accepting state q such that, when processing w , M enters q an infinite number of times. So note that it is not required that M enter an accepting state and stay there. But it is not sufficient for M to enter an accepting state just once (or any finite number of times). As before, the language accepted by M , denoted $L(M)$, is the set of all strings accepted by M . A language L is *Büchi-acceptable* iff it is accepted by some Büchi automaton.

Büchi automata can be used to model concurrent systems, hardware devices, and their specifications. Then programs called model checkers can verify that those systems correctly conform to a set of stated requirements. € 679.

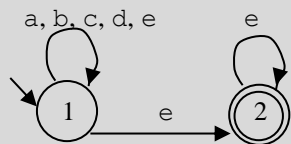
Example 5.38 Büchi Automata for Event Sequences

Suppose that there are five kinds of events that can occur in the system that we wish to model. We'll call them a , b , c , d , and e . So let $\Sigma = \{a, b, c, d, e\}$.

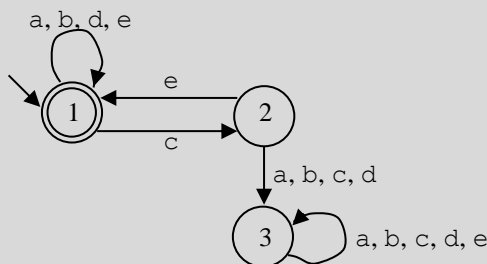
We first consider the case in which we require that event e occur at least once. The following (nondeterministic) Büchi automaton accepts all and only the elements of Σ^* that contain at least one occurrence of e :



Now suppose that we require that there come a point after which only e 's can occur. The following Büchi automaton (described using our convention that the dead state need not be written explicitly) accepts all and only the elements of Σ^* that eventually reach a point after which no events other than e 's occur:



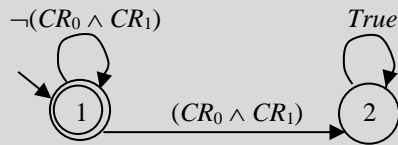
Finally, suppose that we require that every c event be immediately followed by an e event. The following Büchi automaton (this time with the dead state, 3, shown explicitly) accepts all and only the elements of Σ^* that satisfy that requirement:



Example 5.39 Mutual Exclusion

Suppose that we want to model a concurrent system with two processes and enforce the constraint, often called a mutual exclusion property, that it never happens that both processes are in their critical regions at the same time. We could do this in the usual way, using an alphabet of atomic symbols such as $\{\text{Both}, \text{NotBoth}\}$, where the system receives the input Both at any time interval at which both processes are in their critical region and the input NotBoth at any other time interval. But a more direct way to model the behavior of complex concurrent systems is to allow inputs that correspond to Boolean expressions that capture the properties of interest. That way, the same Boolean predicates can be combined into different expressions in different machines that correspond to different desirable properties. To capture the mutual exclusion constraint, we'll use two Boolean predicates, CR_0 , which will be *True* iff $process_0$ is in its critical region and CR_1 , which will be *True* iff $process_1$ is in its critical region. The inputs to the

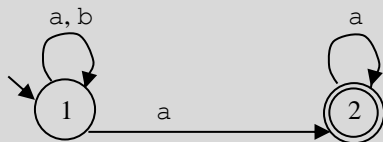
system will then be drawn from a set of three Boolean expressions: $\{(CR_0 \wedge CR_1), \neg(CR_0 \wedge CR_1), True\}$. The following Büchi automaton accepts all and only the input sequences that satisfy the property that $(CR_0 \wedge CR_1)$ never occurs:



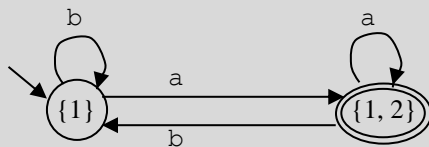
While there is an obvious similarity between Büchi automata and FSMs, and the languages they accept are related, as described below, there is one important difference. For Büchi automata, nondeterminism matters.

Example 5.40 For Büchi Automata, Nondeterminism Matters

Let $L = \{w \in \{a, b\}^\omega : \#_b(w) \text{ is finite}\}$. Note that every string in L must contain an infinite number of a 's. The following nondeterministic Büchi automaton accepts L :



We can try to build a corresponding deterministic machine by using the construction that we used in the proof of Theorem 5.3 (which says that for every NDFSM there does exist an equivalent DFSM). The states of the new machine will then correspond to subsets of states of the original machine and we'll have:



This new machine is indeed deterministic and it does accept all strings in L . Unfortunately, it also accepts an infinite number of strings that are not in L , including $(ba)^\omega$. More unfortunately, we cannot do any better.

Theorem 5.7 Nondeterministic versus Deterministic Büchi Automata

Theorem: There exist languages that can be accepted by a nondeterministic Büchi automaton (i.e., one that meets the definition we have given), but for which there exists no equivalent deterministic Büchi automaton (i.e., one that has a single start state and whose transitions are defined by a function from $(K \times \Sigma)$ to K).

Proof: The proof is by a demonstration that no deterministic Büchi automaton accepts the language $L = \{w \in \{a, b\}^\omega : \#_b(w) \text{ is finite}\}$ of Example 5.40. Suppose that there were such a machine B . Then, among the strings accepted by B , would be every string of the form wa^ω , where w is some finite string in $\{a, b\}^*$. This must be true since all such strings contain only a finite number of b 's. Remove from B any states that are not reachable from the start state. Now consider any remaining state q in B . Since q is reachable from the start state, there must exist at least one finite string that drives B from the start state to q . Call that string w . Then, as we just observed, wa^ω is in L and so must be accepted by B . In order for B to accept it, there must be at least one accepting state q_a that occurs infinitely often in the computation of B on wa^ω . That accepting state must be reachable from q (the state of B when just w has been read) by some finite number, which we'll call a_q , of a 's (since B has only a finite number of states). Compute a_q for every state q in B . Let m be the maximum number of the a_q values.

We can now show that B accepts the string $(ba^m)^\omega$, which is not in L . Since B is deterministic, its transition function is defined on all (state, input) pairs, so it must run forever on all strings including $(ba^m)^\omega$. From the last paragraph we

know that, from any state, there is a string of m or fewer a 's that can drive B to an accepting state. So, in particular, after each time it reads a b , followed by a sequence of a 's, B must reach some accepting state within m a 's. But B has only a finite number of accepting states. So, on input $(ba^m)^\omega$, B reaches some accepting state an infinite number of times and it accepts. ■

There is a natural relationship between the languages of infinite strings accepted by Büchi automata and the regular languages (i.e., the languages of finite strings accepted by FSMs). To describe this relationship requires an understanding of the closure properties of the regular languages that we will present in Section 8.3, as well as some of the decision procedures for regular languages that we will present in Chapter 9. It would be helpful to read those sections before continuing to read this discussion of Büchi automata.

Any Büchi-acceptable language can be described in terms of regular languages. To see how, observe that any Büchi automaton B can almost be viewed as an FSM, if we simply consider input strings of finite length. The only reason that that can't quite be done is that Büchi automata may have multiple start states. So, from any Büchi automaton B , we can build what we'll call the *mirror FSM* M to B as follows: let $M = B$ except that, if B has more than one start state, then, in M , create a new start state that has an ϵ -transition to each of the start states of B . Notice that the set of finite length strings that can drive B from a start state to some state q is identical to the set of finite length strings that can drive M from its start state to state q .

Now consider any Büchi automaton B and any string w that B accepts. Since w is accepted, there is some accepting state in B that is visited an infinite number of times while B processes w . Call that state q . (There may be more than one such state. Pick one.) Then we can divide w into two parts, x and y . The first part, x , has finite length and it drives B from a start state to q for the first time. The second part, y , has infinite length and it simply pushes B through one loop after another, each of which starts and ends in q (although there may be more than one path that does this). The set of possible values for x is regular: it is exactly the set that can be accepted by the FSM M that mirrors B , if we let q be M 's only accepting state. Call a path from q back to itself *minimal* iff it does not pass through q . Then we also notice that the set of strings that can force B through such a minimal path is also regular. It is the set accepted by the FSM M that mirrors B , if we let q be both M 's start state and its only accepting state. These observations lead to the following theorem:

Theorem 5.8 Büchi-Acceptable and Regular Languages

Theorem: L is a Büchi-acceptable language iff it is the finite union of sets each of which is of the form XY^ω , where each X and Y is a regular language.

Proof: Given any Büchi automaton $B = (K, \Sigma, \Delta, S, A)$, let $W_{q_0q_1}$ be the set of all strings that drive B from state q_0 to state q_1 . Then, by the definition of what it means for a Büchi automaton to accept a string, we have:

$$L(B) = \bigcup_{s \in S} \bigcup_{q \in A} W_{sq} (W_{qq})^\omega.$$

If L is a Büchi-acceptable language, then there is some Büchi automaton B that accepts it. So the only-if part of the claim is true since:

- S and A are both finite,
- For each s and q , W_{sq} is regular since it is the set of strings accepted by B 's mirror FSM M with start state s and single accepting state q ,
- $W_{qq} = Y^*$, where Y is the set of strings that can force B along a minimal path from q back to q ,
- Y is regular since it is the set of strings accepted by B 's mirror FSM M with q as its start state and its only accepting state, and
- The regular languages are closed under Kleene star so $W_{qq} = Y^*$ is also regular.

The if part follows from a set of properties of the Büchi-acceptable and regular languages that are described in Theorem 5.9. ■

Theorem 5.9 Closure Properties of Büchi Automata

Theorem and Proof: The Büchi-acceptable languages (like the regular languages) are closed under:

- Concatenation with a regular language: if L_1 is a regular language and L_2 is a Büchi-acceptable language, then L_1L_2 is Büchi-acceptable. The proof is similar to the proof that the regular languages are closed under concatenation except that, since ϵ transitions are not allowed, the machines for the two languages must be “glued together” differently. If q is a state in the FSM that accepts L_1 , and there is a transition from q , labeled c , to some accepting state, then add a transition from q , labeled c , to each start state of the Büchi automaton that accepts L_2 .
- Union: if L_1 and L_2 are Büchi-acceptable, then $L_1 \cup L_2$ is also Büchi-acceptable. The proof is analogous to the proof that the regular languages are closed under union. Again, since ϵ transitions are not allowed, we must use a slightly different glue. The new machine we will build will have transitions directly from a new start state to the states that the original machines can reach after reading one input character.
- Intersection: if L_1 and L_2 are Büchi-acceptable, then $L_1 \cap L_2$ is also Büchi-acceptable. The proof is by construction of a Büchi automaton that effectively runs a Büchi automaton for L_1 in parallel with one for L_2 .
- Complement: if L is Büchi-acceptable, then $\neg L$ is also Büchi-acceptable. The proof of this claim is less obvious. It is given in [Thomas 1990].

Further, if L is a regular language, then L^* is Büchi-acceptable. The proof is analogous to the proof that the regular languages are closed under Kleene star, but we must again use the modification that was used above in the proof of closure under concatenation. ■

Büchi automata are useful as models for computer systems whose properties we wish to reason about because a set of important questions can be answered about them. In particular, Büchi automata share with FSMs the existence of decision procedures for all of the properties described in the following theorem:

Theorem 5.10 Decision Procedures for Büchi Automata

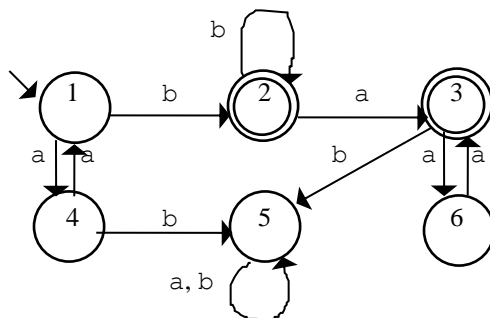
Theorem: There exist decision procedures for all of the following properties:

- Emptiness: Given a Büchi automaton B , is $L(B)$ empty?
- Nonemptiness: Given a Büchi automaton B , is $L(B)$ nonempty?
- Inclusion: Given two Büchi automata B_1 and B_2 , is $L(B_1) \subseteq L(B_2)$?
- Equivalence: Given two Büchi automata B_1 and B_2 , is $L(B_1) = L(B_2)$?

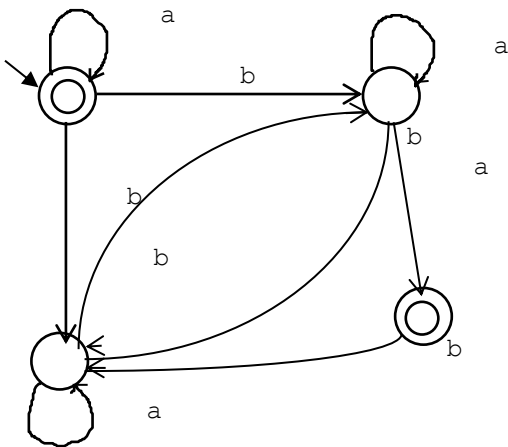
Proof: The proof of each of these claims can be found in [Thomas 1990]. ■

5.13 Exercises

1) Give a clear English description of the language accepted by the following DFSM:



- 2) Show a DFSM to accept each of the following languages:
- $\{w \in \{a, b\}^* : \text{every } a \text{ in } w \text{ is immediately preceded and followed by } b\}$.
 - $\{w \in \{a, b\}^* : w \text{ does not end in } ba\}$.
 - $\{w \in \{0, 1\}^* : w \text{ corresponds to the binary encoding, without leading } 0\text{'s, of natural numbers that are evenly divisible by } 4\}$.
 - $\{w \in \{0, 1\}^* : w \text{ corresponds to the binary encoding, without leading } 0\text{'s, of natural numbers that are powers of } 4\}$.
 - $\{w \in \{0-9\}^* : w \text{ corresponds to the decimal encoding, without leading } 0\text{'s, of an odd natural number}\}$.
 - $\{w \in \{0, 1\}^* : w \text{ has } 001 \text{ as a substring}\}$.
 - $\{w \in \{0, 1\}^* : w \text{ does not have } 001 \text{ as a substring}\}$.
 - $\{w \in \{a, b\}^* : w \text{ has } bbab \text{ as a substring}\}$.
 - $\{w \in \{a, b\}^* : w \text{ has neither } ab \text{ nor } bb \text{ as a substring}\}$.
 - $\{w \in \{a, b\}^* : w \text{ has both } aa \text{ and } bb \text{ as substrings}\}$.
 - $\{w \in \{a, b\}^* : w \text{ contains at least two } b\text{'s that are not immediately followed by an } a\}$.
 - $\{w \in \{0, 1\}^* : w \text{ has no more than one pair of consecutive } 0\text{'s and no more than one pair of consecutive } 1\text{'s}\}$.
 - $\{w \in \{0, 1\}^* : \text{none of the prefixes of } w \text{ ends in } 0\}$.
 - $\{w \in \{a, b\}^* : (\#_a(w) + 2 \cdot \#_b(w)) \equiv_5 0\}$. ($\#_a w$ is the number of a 's in w).
- 3) Consider the children's game Rock, Paper, Scissors \square . We'll say that the first player to win two rounds wins the game. Call the two players A and B .
- Define an alphabet Σ and describe a technique for encoding Rock, Paper, Scissors games as strings over Σ . (Hint: each symbol in Σ should correspond to an ordered pair that describes the simultaneous actions of A and B .)
 - Let L_{RPS} be the language of Rock, Paper, Scissors games, encoded as strings as described in part (a), that correspond to wins for player A . Show a DFSM that accepts L_{RPS} .
- 4) If M is a DFSM and $\varepsilon \in L(M)$, what simple property must be true of M ?
- 5) Consider the following NDFSM M :

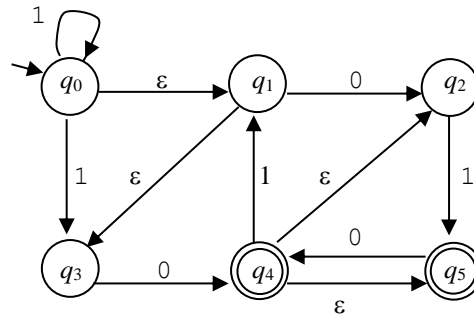


For each of the following strings w , determine whether $w \in L(M)$:

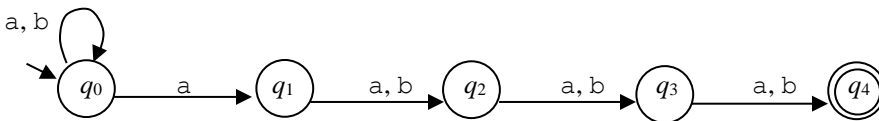
- aabbba.
- bab.
- baba.

- 6) Show a possibly nondeterministic FSM to accept each of the following languages:
- $\{a^m b a^m : n, m \geq 0, n \equiv_3 m\}$.
 - $\{w \in \{a, b\}^* : w \text{ contains at least one instance of } aaba, bbb \text{ or } ababa\}$.
 - $\{w \in \{0-9\}^* : w \text{ corresponds to the decimal encoding of a natural number whose encoding contains, as a substring, the encoding of a natural number that is divisible by 3}\}$.
 - $\{w \in \{0, 1\}^* : w \text{ contains both } 101 \text{ and } 010 \text{ as substrings}\}$.
 - $\{w \in \{0, 1\}^* : w \text{ corresponds to the binary encoding of a positive integer that is divisible by 16 or is odd}\}$.
 - $\{w \in \{a, b, c, d, e\}^* : |w| \geq 2 \text{ and } w \text{ begins and ends with the same symbol}\}$.
- 7) Show an FSM (deterministic or nondeterministic) that accepts $L = \{w \in \{a, b, c\}^* : w \text{ contains at least one substring that consists of three identical symbols in a row}\}$. For example:
- The following strings are in L : $aabbb, baaccbbbb$.
 - The following strings are not in L : $\epsilon, aba, abababab, abcbcab$.
- 8) Show a DFSM to accept each of the following languages. The point of this exercise is to see how much harder it is to build a DFSM for tasks like these than it is to build an NDFSM. So do not simply build an NDFSM and then convert it. But do, after you build a DFSM, build an equivalent NDFSM.
- $\{w \in \{a,b\}^* : \text{the fourth from the last character is } a\}$.
 - $\{w \in \{a, b\}^* : \exists x, y \in \{a,b\}^* : ((w = x \text{ abbaa } y) \vee (w = x \text{ baba } y))\}$.
- 9) For each of the following NDFSMs, use *ndfsmtodfs* to construct an equivalent DFSM. Begin by showing the value of *eps*(*q*) for each state *q*:

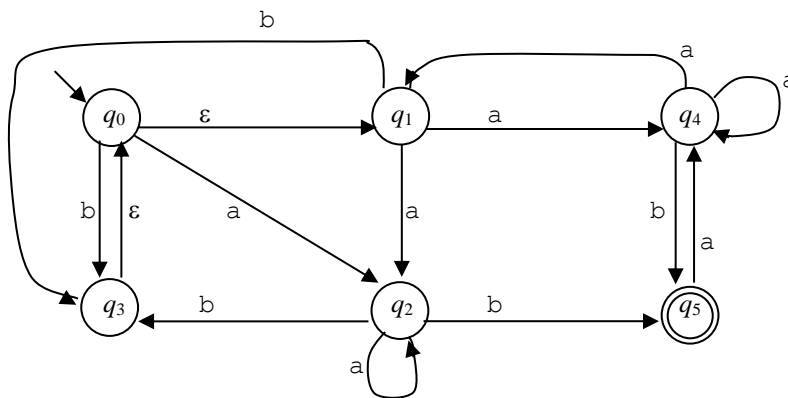
a)



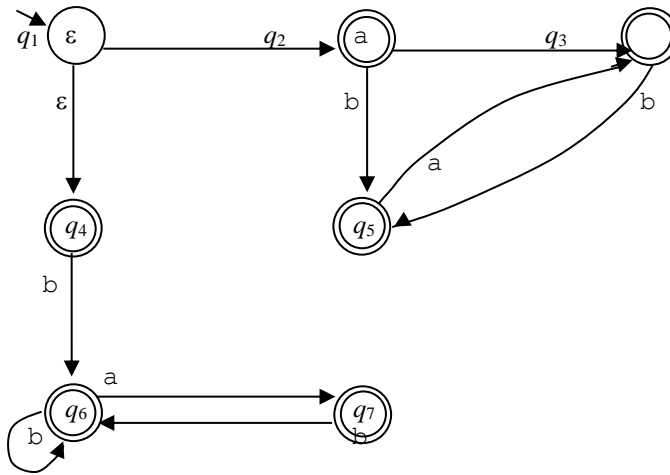
b)



c)



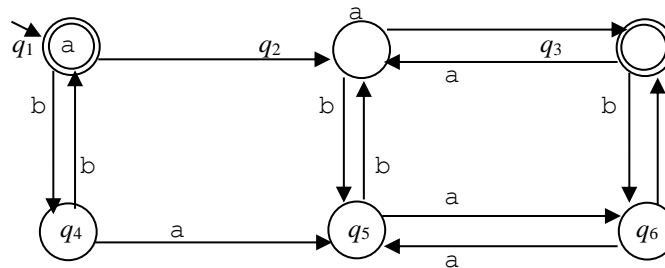
10) Let M be the following NDFSM. Construct (using *ndfsmtod fsm*), a DFSM that accepts $\neg L(M)$.



11) For each of the following languages L :

- (i) Describe the equivalence classes of \approx_L .
- (ii) If the number of equivalence classes of \approx_L is finite, construct the minimal DFSM that accepts L .
 - a) $\{w \in \{0, 1\}^* : \text{every } 0 \text{ in } w \text{ is immediately followed by the string } 11\}$.
 - b) $\{w \in \{0, 1\}^* : w \text{ has either an odd number of } 1\text{'s and an odd number of } 0\text{'s or it has an even number of } 1\text{'s and an even number of } 0\text{'s}\}$.
 - c) $\{w \in \{a, b\}^* : w \text{ contains at least one occurrence of the string } aababa\}$.
 - d) $\{ww^R : w \in \{a, b\}^*\}$.
 - e) $\{w \in \{a, b\}^* : w \text{ contains at least one } a \text{ and ends in at least two } b\text{'s}\}$.
 - f) $\{w \in \{0, 1\}^* : \text{there is no occurrence of the substring } 000 \text{ in } w\}$.

12) Let M be the following DFSM. Use *minDFSM* to minimize M .



13) Construct a deterministic finite state transducer with input alphabet $\{a, b\}$ for each of the following tasks:

- a) On input w , produce 1^n , where $n = \#_a(w)$.
- b) On input w , produce 1^n , where $n = \#_a(w)/2$.
- c) On input w , produce 1^n , where n is the number of occurrences of the substring aba in w .

14) Construct a deterministic finite state transducer that could serve as the controller for an elevator. Clearly describe the input and output alphabets, as well as the states and the transitions between them.

15) Consider the problem of counting the number of words in a text file that may contain letters plus any of the following non-letter characters:


$\langle \text{blank} \rangle \langle \text{linefeed} \rangle \langle \text{end-of-file} \rangle , . ; : ? !$

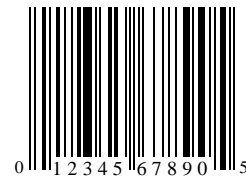
Define a word to be a string of letters that is preceded by either the beginning of the file or some non-letter character and that is followed by some non-letter character. For example, there are 11 words in the following text:



```
The <blank> <blank> cat <blank> <linefeed>
saw <blank> the <blank> <blank> <blank> rat <linefeed>
<blank> with
<linefeed> a <blank> hat <linefeed>
on <blank> the <blank> <blank> mat <end-of-file>
```

Describe a very simple finite-state transducer that reads the characters in the file one at a time and solves the word-counting problem. Assume that there exists an output symbol with the property that, every time it is generated, an external counter gets incremented.

- 16) Real traffic light controllers are more complex than the one that we drew in Example 5.29.
 - a) Consider an intersection of two roads controlled by a set of four lights (one in each direction). Don't worry about allowing for a special left-turn signal. Design a controller for this four-light system.
 - b) As an emergency vehicle approaches an intersection, it should be able to send a signal that will cause the light in its direction to turn green and the light in the cross direction to turn yellow and then red. Modify your design to allow this.

- 17) Real bar code systems are more complex than the one that we sketched in Example 5.31. They must be able to encode all ten digits, for example. There are several industry-standard formats for bar codes, including the common UPC code  found on nearly everything we buy. Describe a finite state transducer that reads the bars and outputs the corresponding decimal number.




- 18) Extend the description of the Soundex FSM that was started in Example 5.33 so that it can assign a code to the name Pfifer. Remember that you must take into account the fact that every Soundex code is made up of exactly four characters.
- 19) Consider the weather/passport HMM of Example 5.37. Trace the execution of the Viterbi and forward algorithms to answer the following questions:
 - a) Suppose that the report ###L is received from Athens. What was the most likely weather during the time of the report?
 - b) Is it more likely that ###L came from London or from Athens?
- 20) Construct a Büchi automaton to accept each of the following languages of infinite length strings:
 - a) $\{w \in \{a, b, c\}^\omega : \text{after any occurrence of an } a \text{ there is eventually an occurrence of a } b\}$.
 - b) $\{w \in \{a, b, c\}^\omega : \text{between any two } a\text{'s there is an odd number of } b\text{'s}\}$.
 - c) $\{w \in \{a, b, c\}^\omega : \text{there never comes a time after which no } b\text{'s occur}\}$.
- 21) In \mathbb{C} 685, we describe the use of statecharts as a tool for building complex systems. A statechart is a hierarchically structured transition network model. Statecharts aren't the only tools that exploit this idea. Another is Simulink[®] , which is one component of the larger programming environment Matlab[®] . Use Simulink to build an FSM simulator.
- 22) In \mathbb{C} 696, we describe the Alternating Bit protocol for handling message transmission in a network. Use the FSM that describes the sender to answer the question, "Is there any upper bound on the number of times a message may be retransmitted?"
- 23) In \mathbb{C} 717, we show an FSM model of a simple intrusion detection device that could be part of a building security system. Extend the model to allow the system to have two zones that can be armed and disarmed independently of each other.

6 Regular Expressions

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

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


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

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

6.1 What is a Regular Expression?

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

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

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

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

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

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

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

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

notice the similarity between the operations that are allowed in regular expressions and the operations that we defined on languages in Section 2.2.

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

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

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

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

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

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

Next consider rule 7: as we showed in the statement of rule 7 itself, the regular expression α^+ is equivalent to the slightly longer regular expression $\alpha\alpha^*$. The form α^+ is a convenient shortcut, and we will use it. But we need not consider rule 7 in any analysis that we may choose to do of regular expressions or the languages that they generate.

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

Example 6.1 Analyzing a Simple Regular Expression

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

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

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

Regular expressions can be used to scan text and pick out email addresses. © 793.

Example 6.2 Another Simple Regular Expression

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

So the meaning of the regular expression $((a \cup b)(a \cup b))a(a \cup b)^*$ is:

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

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

Example 6.3 Given a Language, Find a Regular Expression

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

$((a \cup b)(a \cup b))^*$	This one can be read as, “Go through a loop zero or more times. Each time through, choose an a or b , then choose a second character (a or b).”
----------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------

$(aa \cup ab \cup ba \cup bb)^*$	This one can be read as, “Go through a loop zero or more times. Each time through, choose one of the two-character sequences.”
----------------------------------	--------------------------------------------------------------------------------------------------------------------------------

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

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

Example 6.4 More than One Regular Expression for a Language

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

$$b^* (ab^*ab^*)^* a b^*.$$

$$b^* a b^* (ab^*ab^*)^*.$$

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

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

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

So the expression $(a \cup bb^*a)$ will be interpreted as $(a \cup (b(b^*a)))$.

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

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

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

$a^* \cup b^* \neq (a \cup b)^*$	The language on the right contains the string ab , while the language on the left does not. Every string in the language on the left contains only a 's or only b 's.
$(ab)^* \neq a^*b^*$	The language on the left contains the string $abab$, while the language on the right does not. The language on the right contains the string $aaabbbb$, while the language on the left does not.

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

6.2 Kleene's Theorem

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

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

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

6.2.1 Building an FSM from a Regular Expression

Theorem 6.1 For Every Regular Expression There is an Equivalent FSM

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

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

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

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



Figure 6.1 FSMs for primitive regular expressions

Next we must show how to build FSMs to accept languages that are defined by regular expressions that exploit the operations of concatenation, union, and Kleene star. Let β and γ be regular expressions that define languages over the alphabet Σ . If $L(\beta)$ is regular, then it is accepted by some FSM $M_1 = (K_1, \Sigma, \Delta_1, s_1, A_1)$. If $L(\gamma)$ is regular, then it is accepted by some FSM $M_2 = (K_2, \Sigma, \Delta_2, s_2, A_2)$.

- If α is the regular expression $\beta \cup \gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular, then we construct $M_3 = (K_3, \Sigma, \Delta_3, s_3, A_3)$ such that $L(M_3) = L(\alpha) = L(\beta) \cup L(\gamma)$. If necessary, rename the states of M_1 and M_2 so that $K_1 \cap K_2 = \emptyset$. Create a new start state, s_3 , and connect it to the start states of M_1 and M_2 via ϵ -transitions. M_3 accepts if either M_1 or M_2 accepts. So $M_3 = (\{s_3\} \cup K_1 \cup K_2, \Sigma, \Delta_3, s_3, A_1 \cup A_2)$, where $\Delta_3 = \Delta_1 \cup \Delta_2 \cup \{(s_3, \epsilon), s_1), ((s_3, \epsilon), s_2)\}$.
- If α is the regular expression $\beta\gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular, then we construct $M_3 = (K_3, \Sigma, \Delta_3, s_3, A_3)$ such that $L(M_3) = L(\alpha) = L(\beta)L(\gamma)$. If necessary, rename the states of M_1 and M_2 so that $K_1 \cap K_2 = \emptyset$. We will build M_3 by connecting every accepting state of M_1 to the start state of M_2 via an ϵ -transition. M_3 will start in the start state of M_1 and will accept if M_2 does. So $M_3 = (K_1 \cup K_2, \Sigma, \Delta_3, s_1, A_2)$, where $\Delta_3 = \Delta_1 \cup \Delta_2 \cup \{(q, \epsilon), s_2) : q \in A_1\}$.
- If α is the regular expression β^* and if $L(\beta)$ is regular, then we construct $M_2 = (K_2, \Sigma, \Delta_2, s_2, A_2)$ such that $L(M_2) = L(\alpha) = L(\beta)^*$. We will create a new start state s_2 and make it accepting, thus assuring that M_2 accepts ϵ . (We need a new start state because it is possible that s_1 , the start state of M_1 , is not an accepting state. If it isn't and if it is reachable via any input string other than ϵ , then simply making it an accepting state would cause M_2 to accept strings that are not in $(L(M_1))^*$.) We link the new s_2 to s_1 via an ϵ -transition. Finally, we create ϵ -transitions from

each of M_1 's accepting states back to s_1 . So $M_2 = (\{s_2\} \cup K_1, \Sigma, \Delta_2, s_2, \{s_2\} \cup A_1)$, where $\Delta_2 = \Delta_1 \cup \{((s_2, \epsilon), s_1)\} \cup \{((q, \epsilon), s_1) : q \in A_1\}$.

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

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

regextofsm(α : regular expression) =

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

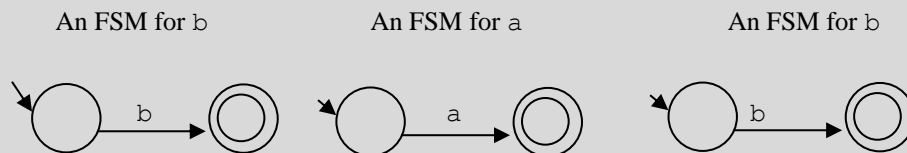
Construct an FSM as described above.

■

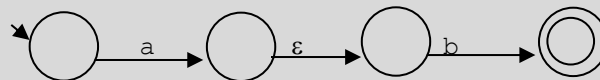
The fact that regular expressions can be transformed into executable finite state machines is important. It means that people can specify programs as regular expressions and then have those expressions “compiled” into efficient processes. For example, hierarchically structured regular expressions, with the same formal power as the regular expressions we have been working with, can be used to describe a lightweight parser for analyzing legacy software. © 689.

Example 6.5 Building an FSM from a Regular Expression

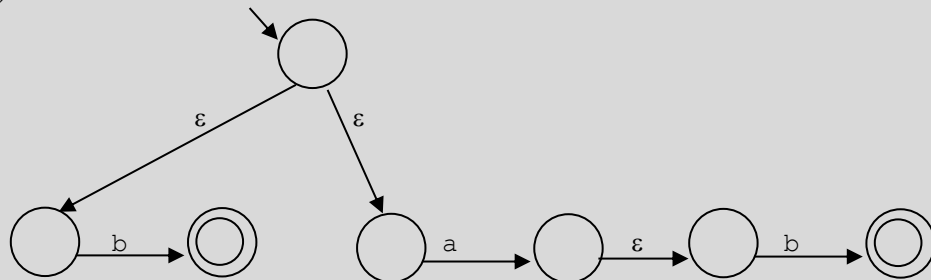
Consider the regular expression $(b \cup ab)^*$. We use *regextofsm* to build an FSM that accepts the language defined by this regular expression:



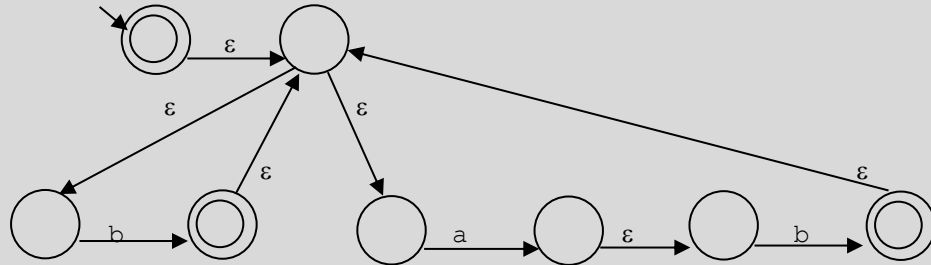
An FSM for ab:



An FSM for $(b \cup ab)^*$:



An FSM for $(b \cup ab)^*$:

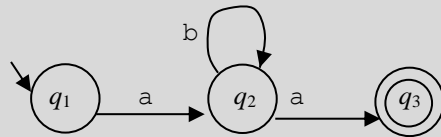


6.2.2 Building a Regular Expression from an FSM

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

Example 6.6

Let M be:



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



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

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

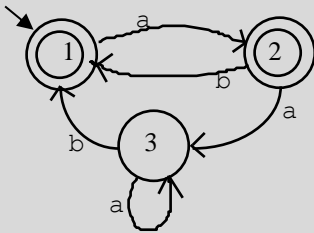
$fsmtoregexheuristic(M: \text{FSM}) =$

1. Remove from M any states that are unreachable from the start state.
2. If M has no accepting states then halt and return the simple regular expression \emptyset .
3. If the start state of M is part of a loop (i.e., it has any transitions coming into it), create a new start state s and connect s to M 's start state via an ϵ -transition. This new start state s will have no transitions into it.
4. If there is more than one accepting state of M or if there is just one but there are any transitions out of it, create a new accepting state and connect each of M 's accepting states to it via an ϵ -transition. Remove the old accepting states from the set of accepting states. Note that the new accepting state will have no transitions out from it.

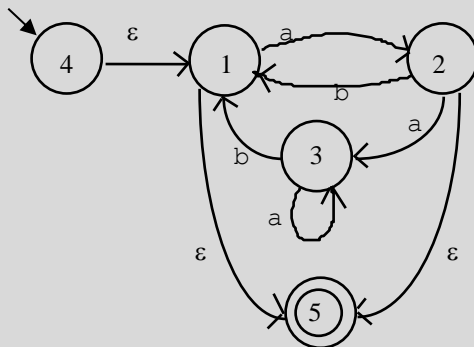
5. If, at this point, M has only one state, then that state is both the start state and the accepting state and M has no transitions. So $L(M) = \{\epsilon\}$. Halt and return the simple regular expression ϵ .
6. Until only the start state and the accepting state remain do:
 - 6.1. Select some state *rip* of M . Any state except the start state or the accepting state may be chosen.
 - 6.2. Remove *rip* from M .
 - 6.3. Modify the transitions among the remaining states so that M accepts the same strings. The labels on the rewritten transitions may be any regular expression.
7. Return the regular expression that labels the one remaining transition from the start state to the accepting state.

Example 6.7 Building a Regular Expression from an FSM

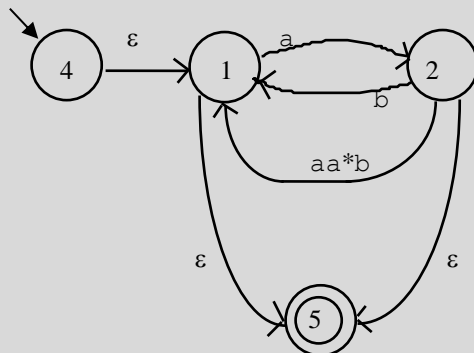
Let M be:



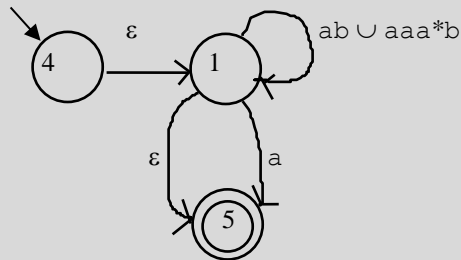
Create a new start state and a new accepting state and link them to M :



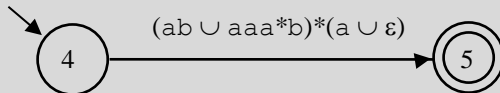
Remove state 3:



Remove state 2:

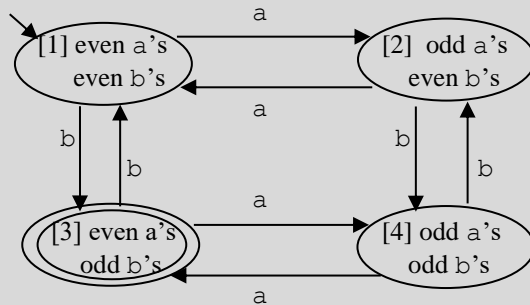


Remove state 1:



Example 6.8 A Simple FSM With No Simple Regular Expression

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



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

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

Theorem 6.2 For every FSM There is an Equivalent Regular Expression

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

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

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

other than the start state there must be exactly one transition from every state (including itself) except the accepting state. To make this true, we do two things:

- If there is more than one transition between states p and q , collapse them into a single transition: If the set of labels on the original set of such transitions is $\{c_1, c_2, \dots, c_n\}$, then delete those transitions and replace them by a single transition with the label $c_1 \cup c_2 \cup \dots \cup c_n$. For example, consider the FSM fragment shown in Figure 6.2 (a). We must collapse the two transitions between states 1 and 2. After doing so, we have the fragment shown in Figure 6.2 (b).



Figure 6.2 Collapsing multiple transitions into one

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

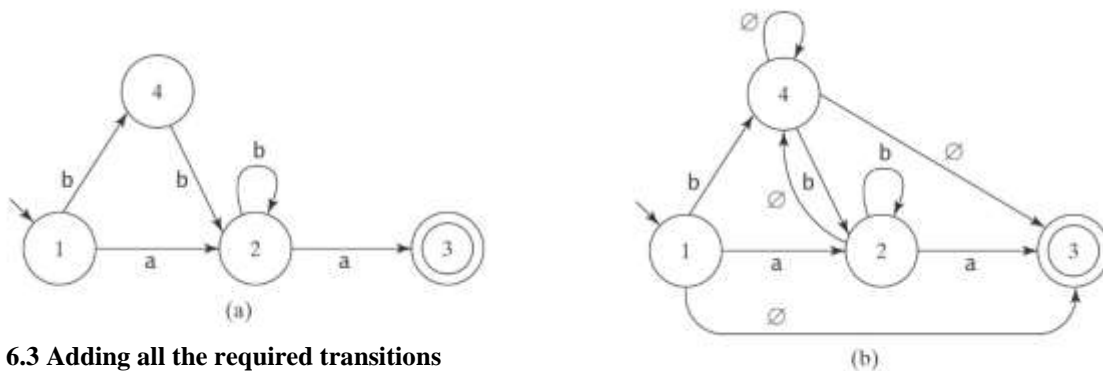


Figure 6.3 Adding all the required transitions

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

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

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

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

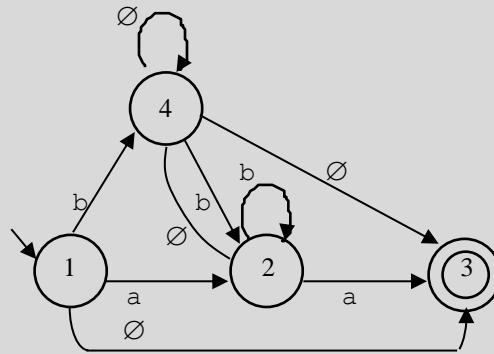
$$\begin{array}{l}
 R(p, q) \\
 \cup \\
 R(p, rip) \\
 R(rip, rip)^* \\
 R(rip, q)
 \end{array}
 \quad
 \begin{array}{l}
 /* \text{ Go directly from } p \text{ to } q, \\
 /* \text{ or} \\
 /* \text{ go from } p \text{ to } rip, \text{ then} \\
 /* \text{ go from } rip \text{ back to itself any number of times, then} \\
 /* \text{ go from } rip \text{ to } q.
 \end{array}$$

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

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

Example 6.9 Ripping States Out One at a Time

Again, let M be:



Let rip be state 2. Then:

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

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

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

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

standardize(M : FSM) =

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

buildregex(M : FSM) =

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

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

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

- 3.3. Remove *rip* and all transitions into and out of it.
4. Return the regular expression that labels the one remaining transition from the start state to the accepting state.


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

fsmtoregex(M : DSM) =

1. $M' = \textit{standardize}(M$: FSM).
2. Return *buildregex*(M').

■

6.2.3 The Equivalence of Regular Expressions and FSMs

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

Theorem 6.3 Kleene's Theorem

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

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

■

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

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

- As we said in the introduction to this chapter, the regular expression language is a pattern language. In particular, regular expressions must specify the order in which a sequence of symbols must occur. This is useful when we want to describe patterns such as phone numbers (it matters that the area code comes first) or email addresses (it matters that the user name comes before the domain).
- But there are some applications where order doesn't matter. The vending machine example that we considered at the beginning of Chapter 5 is an instance of this class of problem. The order in which the coins were entered doesn't matter. Parity checking is another. Only the total number of 1 bits matters, not where they occur in the string. Finite state machines can be very effective in solving problems such as this. But the regular expressions that correspond to those FSMs may be too complex to be useful.

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

Sometimes Writing Regular Expressions is Easy

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

Example 6.10 No More Than One b

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

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

Example 6.11 No Two Consecutive Letters Are the Same

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

$$(b \cup \epsilon) (ab)^* (a \cup \epsilon).$$

$$(a \cup \epsilon) (ba)^* (b \cup \epsilon).$$

Example 6.12 Floating Point Numbers

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

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

Then FLOAT is the language described the following regular expression:

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

It is useful to think of programs, queries, and other strings in practical languages as being composed of a sequence of tokens, where a token is the smallest string that has meaning. So variable and function names, numbers and other constants, operators, and reserved words are all tokens. The regular expression we just wrote for the language FLOAT describes one kind of token. The first thing a compiler does, after reading its input, is to divide it into tokens. That process is called lexical analysis. It is common to use regular expressions to define the behavior of a lexical analyzer. © 669.

Sometimes Building a Deterministic FSM is Easy

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

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

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

By letting the keywords correspond to sequences of amino acids, this idea can be used to build a fast search engine for protein databases. © 734.

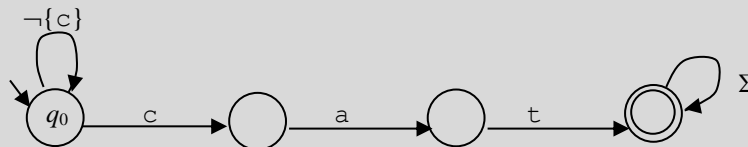
In any of these special cases, we can build a deterministic FSM M by first building a decision tree out of the set of keywords and then adding arcs as necessary to tell M what to do when it reaches a dead end branch of the tree. The following algorithm builds an FSM that accepts any string that contains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

buildkeywordFSM(K : set of keywords) =

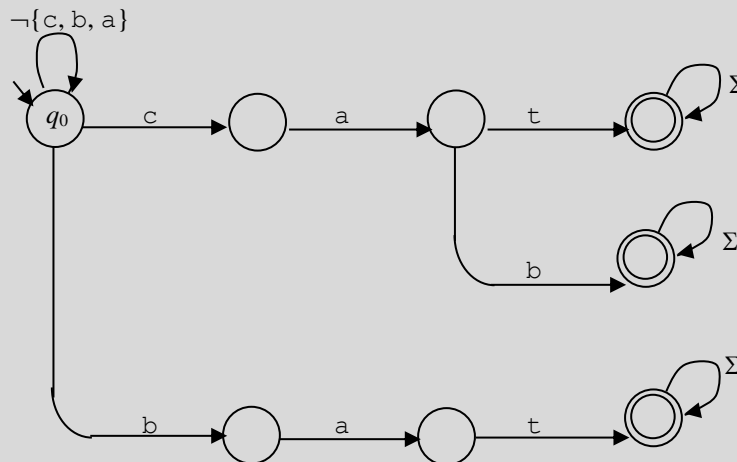
1. Create a start state q_0 .
2. For each element k of K do:
 - Create a branch corresponding to k .
3. Create a set of transitions that describe what to do when a branch dies, either because its complete pattern has been found or because the next character is not the correct one to continue the pattern.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

Example 6.13 Recognizing a Set of Keywords

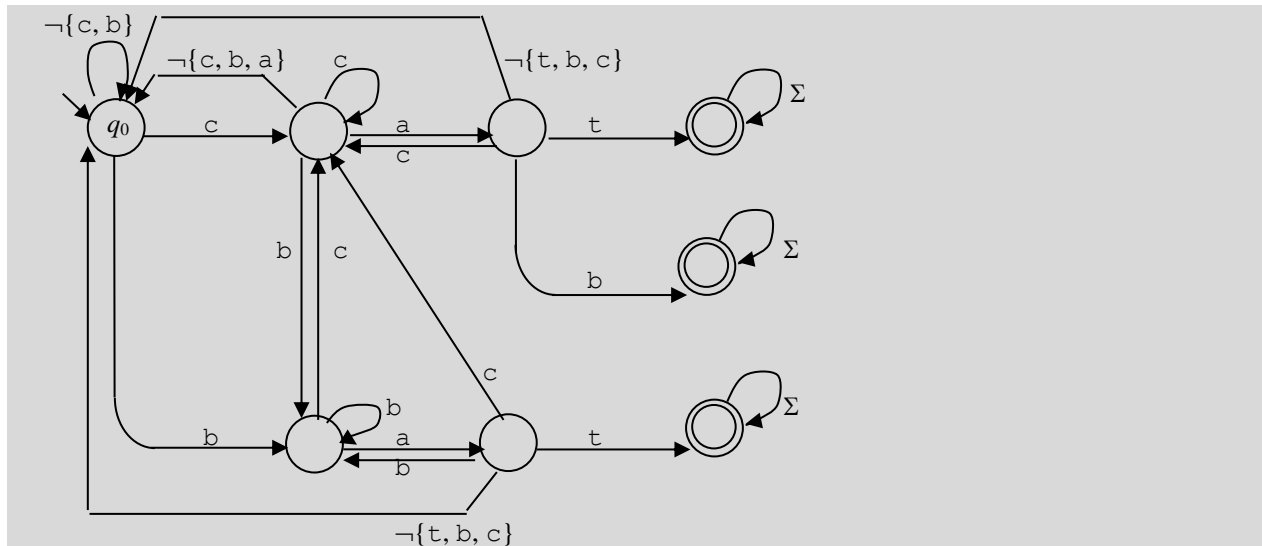
Consider the set of keywords $\{\text{cat}, \text{bat}, \text{cab}\}$. We can use *buildkeywordFSM* to build a DFSM to accept strings that contain at least one of these keywords. We begin by creating a start state and then a path to accept the first keyword, *cat*:



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



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



6.3 Applications of Regular Expressions

Patterns are everywhere.

Regular expressions can be matched against the subject fields of emails to find at least some of the ones that are likely to be spam. © 793.

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

The programming language Perl, for example, supports regular expression matching. © 792. In Exercise 6.19), we’ll consider the formal power of the Perl regular expression language.

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

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

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

Example 6.14 Decimal Numbers

The following regular expression matches decimal encodings of numbers:

$-? ([0-9]^+(\backslash.[0-9]^*)? | \backslash.[0-9]^+)$.

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

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

Meaningful “words” in protein sequences are called motifs. They can be described with regular expressions. © 734.

Example 6.15 Legal Passwords

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

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

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

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

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

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

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

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

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

```
((a-z) ∪ (A-Z)) ((a-z) ∪ (A-Z) ∪ (0-9) ∪ _){3,7}.
```

Example 6.16 IP Addresses

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

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

In XML, regular expressions are one way to define parts of new document types. © 807.

6.4 Manipulating and Simplifying Regular Expressions

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

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

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

Next we consider identities involving concatenation:

- Concatenation is associative: for any regular expressions α , β , and γ , $(\alpha\beta)\gamma = \alpha(\beta\gamma)$.
- ε is the identity for concatenation: for any regular expression α , $\alpha\varepsilon = \varepsilon\alpha = \alpha$.
- \emptyset is a zero for concatenation: for any regular expression α , $\alpha\emptyset = \emptyset\alpha = \emptyset$.

Concatenation distributes over union:

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

Finally, we introduce identities involving Kleene star:

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

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

α is redundant because any string it can generate and place at the beginning of a string to be generated by the combined expression $\alpha^*\beta^*$ can also be generated by β^* .


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

- For any regular expressions α and β , if $L(\beta) \subseteq L(\alpha^*)$ then $(\alpha \cup \beta)^* = \alpha^*$. For example, $(a \cup \epsilon)^* = a^*$, since $\{\epsilon\} \subseteq L(a^*)$. β is redundant since any string it can generate can also be generated by α^* .

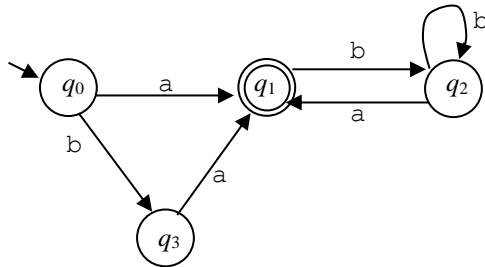
Example 6.17 Simplifying a Regular Expression

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

6.5 Exercises

- Describe in English, as briefly as possible, the language defined by each of these regular expressions:
 - $(b \cup ba)(b \cup a)^*(ab \cup b)$.
 - $((a^*b^*)^*ab) \cup ((a^*b^*)^*ba)(b \cup a)^*$.
- Write a regular expressions to describe each of the following languages:
 - $\{w \in \{a, b\}^* : \text{every } a \text{ in } w \text{ is immediately preceded and followed by } b\}$.
 - $\{w \in \{a, b\}^* : w \text{ does not end in } ba\}$.
 - $\{w \in \{0, 1\}^* : \exists y \in \{0, 1\}^* (|xy| \text{ is even})\}$.
 - $\{w \in \{0, 1\}^* : w \text{ corresponds to the binary encoding, without leading } 0\text{'s, of natural numbers that are evenly divisible by } 4\}$.
 - $\{w \in \{0, 1\}^* : w \text{ corresponds to the binary encoding, without leading } 0\text{'s, of natural numbers that are powers of } 4\}$.
 - $\{w \in \{0-9\}^* : w \text{ corresponds to the decimal encoding, without leading } 0\text{'s, of an odd natural number}\}$.
 - $\{w \in \{0, 1\}^* : w \text{ has } 001 \text{ as a substring}\}$.
 - $\{w \in \{0, 1\}^* : w \text{ does not have } 001 \text{ as a substring}\}$.
 - $\{w \in \{a, b\}^* : w \text{ has } bba \text{ as a substring}\}$.
 - $\{w \in \{a, b\}^* : w \text{ has both } aa \text{ and } bb \text{ as substrings}\}$.
 - $\{w \in \{a, b\}^* : w \text{ has both } aa \text{ and } aba \text{ as substrings}\}$.
 - $\{w \in \{a, b\}^* : w \text{ contains at least two } b\text{'s that are not followed by an } a\}$.
 - $\{w \in \{0, 1\}^* : w \text{ has at most one pair of consecutive } 0\text{'s and at most one pair of consecutive } 1\text{'s}\}$.
 - $\{w \in \{0, 1\}^* : \text{none of the prefixes of } w \text{ ends in } 0\}$.
 - $\{w \in \{a, b\}^* : \#_a(w) \equiv_3 0\}$.
 - $\{w \in \{a, b\}^* : \#_a(w) \leq 3\}$.
 - $\{w \in \{a, b\}^* : w \text{ contains exactly two occurrences of the substring } aa\}$.
 - $\{w \in \{a, b\}^* : w \text{ contains no more than two occurrences of the substring } aa\}$.
 - $\{w \in \{a, b\}^* - L\}$, where $L = \{w \in \{a, b\}^* : w \text{ contains } bba \text{ as a substring}\}$.
 - $\{w \in \{0, 1\}^* : \text{every odd length string in } L \text{ begins with } 11\}$.
 - $\{w \in \{0-9\}^* : w \text{ represents the decimal encoding of an odd natural number without leading } 0\text{'s}\}$.
 - $L_1 - L_2$, where $L_1 = a^*b^*c^*$ and $L_2 = c^*b^*a^*$.
 - The set of legal United States zip codes .
 - The set of strings that correspond to domestic telephone numbers in your country.

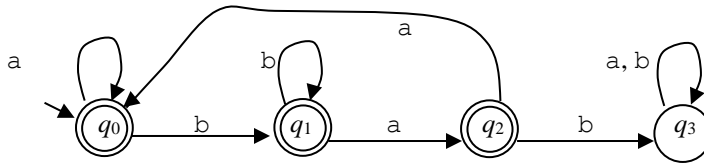
- 3) Simplify each of the following regular expressions:
- $(a \cup b)^*(a \cup \epsilon) b^*$.
 - $(\emptyset^* \cup b) b^*$.
 - $(a \cup b)^* a^* \cup b$.
 - $((a \cup b)^*)^*$.
 - $((a \cup b)^+)^*$.
 - $a((a \cup b)(b \cup a))^* \cup a((a \cup b)a)^* \cup a((b \cup a)b)^*$.
- 4) For each of the following expressions E , answer the following three questions and prove your answer:
- Is E a regular expression?
 - If E is a regular expression, give a simpler regular expression.
 - Does E describe a regular language?
- $((a \cup b) \cup (ab))^*$.
 - $(a^+ a^n b^n)$.
 - $((ab)^* \emptyset)$.
 - $((ab \cup c)^* \cap (b \cup c^*))$.
 - $(\emptyset^* \cup (bb^*))$.
- 5) Let $L = \{a^n b^n : 0 \leq n \leq 4\}$.
- Show a regular expression for L .
 - Show an FSM that accepts L .
- 6) Let $L = \{w \in \{1, 2\}^* : \text{for all prefixes } p \text{ of } w, \text{ if } |p| > 0 \text{ and } |p| \text{ is even, then the last character of } p \text{ is } 1\}$.
- Write a regular expression for L .
 - Show an FSM that accepts L .
- 7) Use the algorithm presented in the proof of Kleene's Theorem to construct an FSM to accept the language generated by each of the following regular expressions:
- $(b(b \cup \epsilon)b)^*$.
 - $bab \cup a^*$.
- 8) Let L be the language accepted by the following finite state machine:



Indicate, for each of the following regular expressions, whether it correctly describes L :

- $(a \cup ba)bb^*a$.
- $(\epsilon \cup b)a(bb^*a)^*$.
- $ba \cup ab^*a$.
- $(a \cup ba)(bb^*a)^*$.

9) Consider the following FSM M :



- a) Show a regular expression for $L(M)$.
- b) Describe $L(M)$ in English.

10) Consider the FSM M of Example 5.3. Use *fsmtoregexheuristic* to construct a regular expression that describes $L(M)$.

11) Consider the FSM M of Example 6.9. Apply *fsmtoregex* to M and show the regular expression that results.

12) Consider the FSM M of Example 6.8. Apply *fsmtoregex* to M and show the regular expression that results. (Hint: this one is exceedingly tedious, but it can be done.)

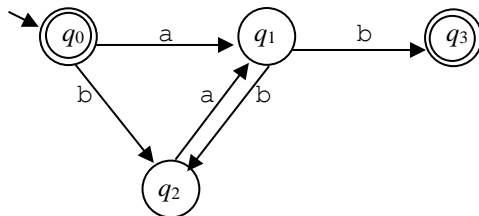
13) Show a possibly nondeterministic FSM to accept the language defined by each of the following regular expressions:

- a) $((a \cup ba) b \cup aa)^*$.
- b) $(b \cup \epsilon)(ab)^*(a \cup \epsilon)$.
- c) $(babb^* \cup a)^*$.
- d) $(ba \cup ((a \cup bb) a^*b))$.
- e) $(a \cup b)^* aa (b \cup aa) bb (a \cup b)^*$.

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

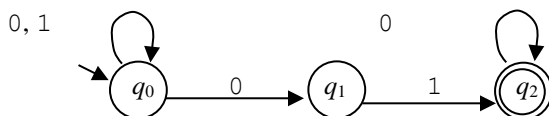
- a) $(aba \cup aabaa)^*$.
- b) $(ab)^*(aab)^*$.

15) Consider the following DFSM M :



- a) Write a regular expression that describes $L(M)$.
- b) Show a DFSM that accepts $\neg L(M)$.

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



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

- 18) Let $\Sigma = \{a, b\}$. Let $L = \{\varepsilon, a, b\}$. Let R be a relation defined on Σ^* as follows: $\forall xy (xRy \text{ iff } y = xb)$. Let R' be the reflexive, transitive closure of R . Let $L' = \{x : \exists y \in L (yR'x)\}$. Write a regular expression for L' .
- 19) In \mathcal{C} 792, we summarize the main features of the regular expression language in Perl. What feature of that regular expression language makes it possible to write regular expressions that describe languages that aren't regular?
- 20) For each of the following statements, state whether it is *True* or *False*. Prove your answer.
- $(ab)^*a = a(ba)^*$.
 - $(a \cup b)^*b(a \cup b)^* = a^*b(a \cup b)^*$.
 - $(a \cup b)^*b(a \cup b)^* \cup (a \cup b)^*a(a \cup b)^* = (a \cup b)^*$.
 - $(a \cup b)^*b(a \cup b)^* \cup (a \cup b)^*a(a \cup b)^* = (a \cup b)^+$.
 - $(a \cup b)^*ba(a \cup b)^* \cup a^*b^* = (a \cup b)^*$.
 - $a^*b(a \cup b)^* = (a \cup b)^*b(a \cup b)^*$.
 - If α and β are any two regular expressions, then $(\alpha \cup \beta)^* = \alpha(\beta\alpha \cup \alpha)$.
 - If α and β are any two regular expressions, then $(\alpha\beta)^*\alpha = \alpha(\beta\alpha)^*$.

7 Regular Grammars ◆

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

- Finite state machines.
- Regular expressions.

We now introduce a third:

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

7.1 Definition of a Regular Grammar

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

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

In a regular grammar, all rules in R must:

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

So $S \rightarrow a$, $S \rightarrow \epsilon$, and $T \rightarrow aS$ are legal rules in a regular grammar. $S \rightarrow aSa$ and $aSa \rightarrow T$ are not legal rules in a regular grammar.

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

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

Example 7.1 Even Length Strings

Let $L = \{w \in \{a, b\}^* : |w| \text{ is even}\}$.

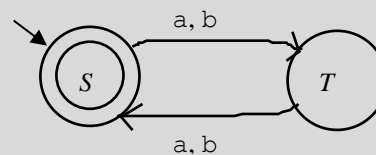
The following regular expression defines L :

$$((aa) \cup (ab) \cup (ba) \cup (bb))^*$$

The following regular grammar G also defines L :

$$\begin{array}{lll} S \rightarrow \epsilon & S \rightarrow aT & S \rightarrow bT \\ T \rightarrow aS & T \rightarrow bS & \end{array}$$

The following DFSM M accepts L :



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

remaining even length strings. T generates every string whose length is one greater than the length of some string S generates. So, if S generates all and only the even length strings, then T generates all and only the other odd length strings.

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

7.2 Regular Grammars and Regular Languages

Theorem 7.1 Regular Grammars Define Exactly the Regular Languages

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

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

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

grammartofsm(G : regular grammar) =

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

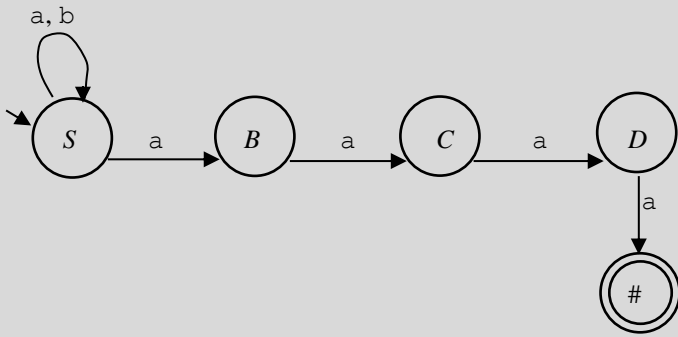
FSM \rightarrow Regular grammar : The construction is effectively the reverse of the one we just did. We leave this step as an exercise. ■

Example 7.2 Strings that End with aaaa

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

$S \rightarrow aS$	/* An arbitrary number of a's and b's can be generated before the
$S \rightarrow bS$	pattern starts.
$S \rightarrow aB$	/* Generate the first a of the pattern.
$B \rightarrow aC$	/* Generate the second a of the pattern.
$C \rightarrow aD$	/* Generate the third a of the pattern.
$D \rightarrow a$	/* Generate the last a of the pattern and quit.

Applying *grammartofsm* to this grammar, we get:



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

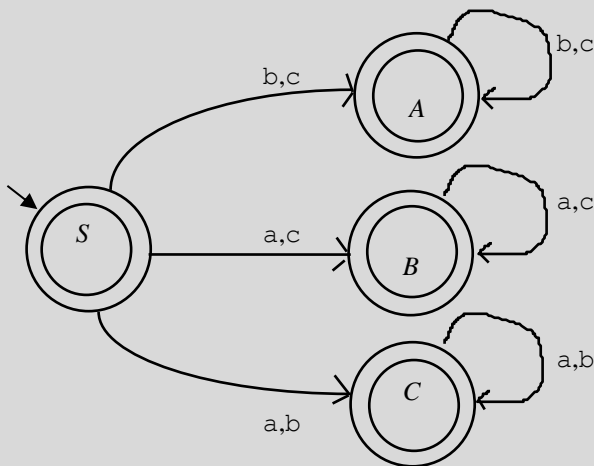
Example 7.3 The Missing Letter Language

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

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

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

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



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

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

Example 7.4 Satisfying Multiple Criteria

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

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

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

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

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

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

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

7.3 Exercises

- 1) Show a regular grammar for each of the following languages:
 - a) $\{w \in \{a, b\}^* : w \text{ contains an even number of } a\text{'s and an odd number of } b\text{'s}\}$.
 - b) $\{w \in \{a, b\}^* : w \text{ does not end in } aa\}$.
 - c) $\{w \in \{a, b\}^* : w \text{ contains the substring } abb\}$.
 - d) $\{w \in \{a, b\}^* : \text{if } w \text{ contains the substring } aa \text{ then } |w| \text{ is odd}\}$.
 - e) $\{w \in \{a, b\}^* : w \text{ does not contain the substring } aabb\}$.

- 2) Consider the following regular grammar G :
$$\begin{aligned} S &\rightarrow aT \\ T &\rightarrow bT \\ T &\rightarrow a \\ T &\rightarrow aW \\ W &\rightarrow \varepsilon \\ W &\rightarrow aT \end{aligned}$$
 - a) Write a regular expression that generates $L(G)$.
 - b) Use *grammartofsm* to generate an FSM M that accepts $L(G)$.

- 3) Consider again the FSM M shown in Exercise 5.1). Show a regular grammar that generates $L(M)$.

- 4) Show by construction that, for every FSM M there exists a regular grammar G such that $L(G) = L(M)$.

- 5) Let $L = \{w \in \{a, b\}^* : \text{every } a \text{ in } w \text{ is immediately followed by at least one } b\}$.
 - a) Write a regular expression that describes L .
 - b) Write a regular grammar that generates L .
 - c) Construct an FSM that accepts L .

8 Regular and Nonregular Languages

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

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

8.1 How Many Regular Languages Are There?

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

Theorem 8.1 The Regular Languages are Countably Infinite

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

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

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

■

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

8.2 Showing That a Language Is Regular

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

Theorem 8.2 The Finite Languages

Theorem: Every finite language is regular.

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

$$s_1 \cup s_2 \cup \dots \cup s_n$$

So it too is regular.

■

Example 8.1 The Intersection of Two Infinite Languages

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

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

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

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

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

Let:

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

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

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

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

Not every regular language is computationally tractable. Consider the Towers of Hanoi language.
 $\text{C } 798.$

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

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

8.3 Some Important Closure Properties of Regular Languages

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

Theorem 8.3 Closure under Union, Concatenation and Kleene Star

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

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

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

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

Proof:

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

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

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

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

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

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

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

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

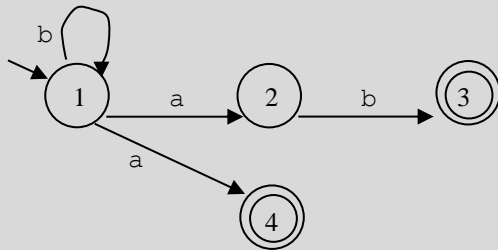
We have already shown that the regular languages are closed under both complement and intersection. Thus they are also closed under set difference.

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

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

Example 8.4 Closure Under Complement

Consider the following NDFSM $M =$



If we use the algorithm that we just described to convert M to a new machine M' that accepts $\neg L(M)$, the last step is to swap the accepting and the nonaccepting states. A quick look at M makes it clear why it is necessary first to make M deterministic and then to complete it by adding the dead state. M accepts the input a in state 4. If we simply swapped accepting and nonaccepting states, without making the other changes, M' would also accept a . It would do so in state 2. The problem is that M is nondeterministic, and has one path along which a is accepted and one along which it is rejected.

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

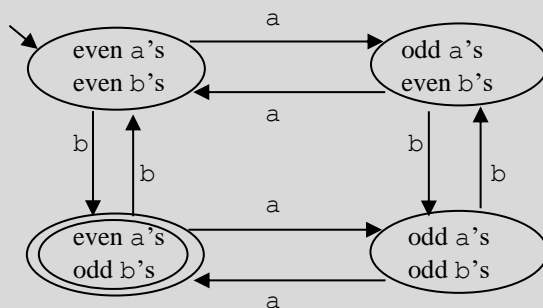
The closure theorems that we have now proved make it easy to take a divide-and-conquer approach to showing that a language is regular. They also let us reuse proofs and constructions that we've already done.

Example 8.5 The Divide-and-Conquer Approach

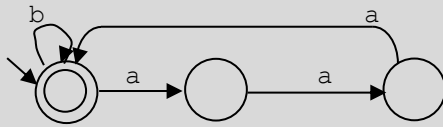
Let $L = \{w \in \{a, b\}^* : w \text{ contains an even number of } a\text{'s and an odd number of } b\text{'s and all } a\text{'s come in runs of three}\}$. L is regular because it is the intersection of two regular languages. $L = L_1 \cap L_2$, where:

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

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



Of course, we could start with this machine and modify it so that it accepts L . But an easier approach is exploit a divide-and-conquer approach. We'll simply use the machine we have and then build a second simple machine, this one to accept L_2 . Then we can prove that L is regular by exploiting the fact that the regular languages are closed under intersection. The following machine accepts L_2 :



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

Example 8.6 What the Closure Theorem for Union Does Not Say

The closure theorem for union says that:

if L_1 and L_2 are regular *then* $L = L_1 \cup L_2$ is regular.

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

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

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

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

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

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

In this case, both L_1 and L_2 are regular.

Example 8.7 What the Closure Theorem for Concatenation Does Not Say

The closure theorem for concatenation says that:

if L_1 and L_2 are regular *then* $L = L_1 L_2$ is regular.

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

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

$$L = L_1 L_2.$$

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

$$\{aaa^*\} = \{a^*\} \{a^p : p \text{ is prime}\}.$$

$$L = L_1 L_2.$$

While again L_2 is not regular, now L is.

8.4 Showing That a Language is Not Regular

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

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

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

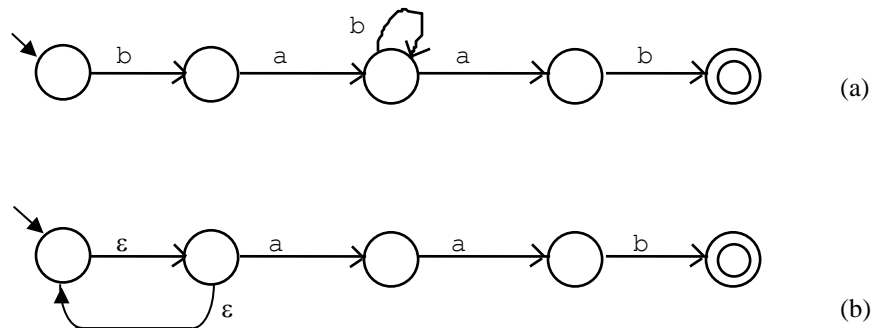


Figure 8.1 What is the longest string that a 5-state FSM can accept?

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

Theorem 8.5 Long Strings Force Repeated States

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

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

Let $M = (K, \Sigma, \delta, s, A)$ be any DFSM. Suppose that there exists some “long” string w (i.e., $|w| \geq |K|$) such that $w \in L(M)$. Then M must go through at least one loop when it reads w . So there is some substring y of w that drove M through at least one loop. Suppose we excise y from w . The resulting string must also be in $L(M)$ since M can accept it just as it accepts w but skipping one pass through one loop. Further, suppose that we splice in one or more extra copies of y , immediately adjacent to the original one. All the resulting strings must also be in $L(M)$ since M can accept them by

going through its loop one or more additional times. Using an analogy with a pump, we'll say that we can *pump* y out once or in an arbitrary number of times and the resulting string must still be in L .

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

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

8.4.2 The Pumping Theorem for Regular Languages

Theorem 8.6 The Pumping Theorem for Regular Languages

Theorem: If L is a regular language, then:

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

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

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

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

Example 8.8 A^nB^n is not Regular

Let L be $A^nB^n = \{a^n b^n : n \geq 0\}$. We can use the Pumping Theorem to show that L is not regular. If it were, then there would exist some k such that any string w , where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = a^k b^k$. Since $|w| = 2k$, w is long enough and it is in L , so it must satisfy the conditions

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

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

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

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

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

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

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

Example 8.9 A Less Judicious Choice for w

Again let L be $A^nB^n = \{a^n b^n : n \geq 0\}.$ If A^nB^n were regular, then there would exist some k such that any string $w,$ where $|w| \geq k,$ must satisfy the conditions of the theorem. Let $w = a^{\lceil k/2 \rceil} b^{\lceil k/2 \rceil}.$ (We must use $\lceil k/2 \rceil,$ i.e., the smallest integer greater than $k/2,$ rather than truncating the division, since k might be odd.) Since $|w| \geq k$ and w is in $L,$ w must satisfy the conditions of the Pumping Theorem. So, there must exist $x, y,$ and $z,$ such that $w = xyz, |xy| \leq k, y \neq \epsilon,$ and $\forall q \geq 0 (xy^qz \in L).$ We show that no such $x, y,$ and z exist. This time, if they did, y could be almost anywhere in w (since all the Pumping Theorem requires is that it occur in the first k characters and there are only at most $k+1$ characters). So we must consider three cases and show that, in all three, there is no y that satisfies all conditions of the Pumping Theorem. A useful way to describe the cases is to imagine w divided into two regions:


aaaaa.....aaaaaa		bbbbbb.....bbbbbb
1		2

Now we see that y can fall:

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

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

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

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

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

Example 8.10 The Balanced Parenthesis Language is Not Regular

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

Example 8.11 The Even Palindrome Language is Not Regular

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

The Pumping Theorem says that, for any language L , if L is regular, then all long strings in L must be pumpable. Our strategy in using it to show that a language L is not regular is to find *one* string that fails to meet that requirement. Often, there are many long strings that *are* pumpable. If we try to work with them, we will fail to derive the contradiction that we seek. In that case, we will know nothing about whether or not L is regular. To find a w that is not pumpable, think about what property of L is not checkable by an FSM and choose a w that exhibits that property.

Consider again our last example. The thing that an FSM cannot do is to remember an arbitrarily long first half and check it against the second half. So we chose a w that would have forced it to do that. Suppose instead that we had let $w = a^k a^k$. It is in L and long enough. But y could be aa and we could pump it out or in and all the resulting strings would be in L .

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

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

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

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

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

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

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

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

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

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

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

- To choose w :
 - Choose a w that is in the part of L that makes it not regular.
 - Choose a w that is only barely in L .
 - Choose a w with as homogeneous as possible an initial region of length at least k .

- To choose q :
 - Try letting q be either 0 or 2.
 - If that doesn't work, analyze L to see if there is some other specific value that will work.

8.4.3 Using Closure Properties

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

Example 8.14 Using Intersection to Force Order Constraints

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

Example 8.15 Using Closure Under Complement

Let $L = \{a^i b^j : i, j \geq 0 \text{ and } i \neq j\}$. It seems unlikely that L is regular since any machine to accept it would have to count the a 's. It is possible to use the Pumping Theorem to prove that L is not regular but it is not easy to see how. Suppose, for example, that we let $w = a^{k+1} b^k$. But then y could be aa and it would pump since $a^{k+1} b^k$ is in L , and so is $a^{k+1+2(q-1)} b^k$, for all nonnegative values of q .

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

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

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

Example 8.16 Sometimes We Must Use the Closure Theorems

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

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

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

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

8.5 Exploiting Problem-Specific Knowledge

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

Example 8.17 The Octal Representation of a Number Divisible by 7

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

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

Example 8.18 A Music Language

Let $\Sigma = \{\circ, \downarrow, \cdot, \bullet, \blacktriangleright, \blacktriangleleft, \blacktriangleright\}$. Let $L = \{w : w \text{ represents a song written in } 4/4 \text{ time}\}$. L is regular. It can be accepted by an FSM that checks for 4 beats between measure bars, where \circ counts as 4, \downarrow counts as 2, \cdot counts as 1, \bullet counts as $1/2$, \blacktriangleright counts as $1/4$, and \blacktriangleleft counts as $1/8$.

Other techniques described in this book can also be applied to the language of music. © 776.

Example 8.19 English

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

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

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

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

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

Is there a longest English sentence? Are there other ways of showing that English isn't regular? Would it be useful to describe English as a regular language even if we could? © 743.

8.6 Functions on Regular Languages

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

Example 8.20 The Function *firstchars*

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

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

1. Mark all the states in M from which there exists some path to some accepting state.
 - /* Find all the characters that are initial characters in some string in L .
2. $clist = \emptyset$.
3. For each character c in Σ do:
 - If there is a transition from s , with label c , to some state q , and q was marked in step 1 then:

$$clist = clist \cup \{c\}.$$
- /* Build M' .
4. If $clist = \emptyset$ then construct M' with a single state s' , which is not accepting.
5. Else do:
 - Create a start state s' and make it the first state in A' .
 - For each character c in $clist$ do:
 - Create a new state q_c and add it to A' .
 - Add a transition from s' to q_c labeled c .
 - Add a transition from q_c to q_c labeled c .

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

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

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

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

Example 8.21 The Function *chop*

Consider again the function *chop*, which we defined in Example 4.10. $Chop(L) = \{w : \exists x \in L (x = x_1cx_2, x_1 \in \Sigma_L^*, x_2 \in \Sigma_L^*, c \in \Sigma_L, |x_1| = |x_2|, \text{ and } w = x_1x_2)\}$. In other words, *chop*(L) is all the odd length strings in L with their middle character chopped out.

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

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

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

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

Example 8.22 The Function *maxstring*

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

L	<i>maxstring</i> (L)
\emptyset	\emptyset
a^*b^*	\emptyset
ab^*a	ab^*a
a^*b^*a	a^*b^a

Example 8.23 The Function *mix*

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

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

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

8.7 Exercises

- For each of the following languages L , state whether L is regular or not and prove your answer:
 - $\{a^i b^j : i, j \geq 0 \text{ and } i + j = 5\}$.
 - $\{a^i b^j : i, j \geq 0 \text{ and } i - j = 5\}$.
 - $\{a^i b^j : i, j \geq 0 \text{ and } |i - j| \equiv_5 0\}$.
 - $\{w \in \{0, 1, \#\}^* : w = x\#y, \text{ where } x, y \in \{0, 1\}^* \text{ and } |x| \cdot |y| \equiv_5 0\}$. (Let \cdot mean integer multiplication).
 - $\{a^i b^j : 0 \leq i < j < 2000\}$.
 - $\{w \in \{Y, N\}^* : w \text{ contains at least two } Y\text{'s and at most two } N\text{'s}\}$.
 - $\{w = xy : x, y \in \{a, b\}^* \text{ and } |x| = |y| \text{ and } \#_a(x) \geq \#_a(y)\}$.
 - $\{w = xyz y^R x : x, y, z \in \{a, b\}^*\}$.
 - $\{w = xyz y : x, y, z \in \{0, 1\}^+\}$.
 - $\{w \in \{0, 1\}^* : \#_0(w) \neq \#_1(w)\}$.
 - $\{w \in \{a, b\}^* : w = w^R\}$.
 - $\{w \in \{a, b\}^* : \exists x \in \{a, b\}^+ (w = x x^R x)\}$.
 - $\{w \in \{a, b\}^* : \text{the number of occurrences of the substring } ab \text{ equals the number of occurrences of the substring } ba\}$.
 - $\{w \in \{a, b\}^* : w \text{ contains exactly two more } b\text{'s than } a\text{'s}\}$.
 - $\{w \in \{a, b\}^* : w = xyz, |x| = |y| = |z|, \text{ and } z = x \text{ with every } a \text{ replaced by } b \text{ and every } b \text{ replaced by } a\}$.
Example: $abbbabbaa \in L$, with $x = abb, y = bab$, and $z = baa$.
 - $\{w : w \in \{a - z\}^* \text{ and the letters of } w \text{ appear in reverse alphabetical order}\}$. For example, $spoonfeed \in L$.
 - $\{w : w \in \{a - z\}^* \text{ every letter in } w \text{ appears at least twice}\}$. For example, $unprosperousness \in L$.
 - $\{w : w \text{ is the decimal encoding of a natural number in which the digits appear in a non-decreasing order without leading zeros}\}$.
 - $\{w \text{ of the form: } \langle integer_1 \rangle + \langle integer_2 \rangle = \langle integer_3 \rangle, \text{ where each of the substrings } \langle integer_1 \rangle, \langle integer_2 \rangle, \text{ and } \langle integer_3 \rangle \text{ is an element of } \{0 - 9\}^* \text{ and } integer_3 \text{ is the sum of } integer_1 \text{ and } integer_2\}$. For example, $124+5=129 \in L$.
 - L_0^* , where $L_0 = \{b^i a^j a^k, j \geq 0, 0 \leq i \leq k\}$.
 - $\{w : w \text{ is the encoding of a date that occurs in a year that is a prime number}\}$. A date will be encoded as a string of the form $mm/dd/yyyy$, where each m, d , and y is drawn from $\{0-9\}$.
 - $\{w \in \{1\}^* : w \text{ is, for some } n \geq 1, \text{ the unary encoding of } 10^n\}$. (So $L = \{1111111111, 1^{100}, 1^{1000}, \dots\}$.)
- For each of the following languages L , state whether L is regular or not and prove your answer:
 - $\{w \in \{a, b, c\}^* : \text{in each prefix } x \text{ of } w, \#_a(x) = \#_b(x) = \#_c(x)\}$.
 - $\{w \in \{a, b, c\}^* : \exists \text{ some prefix } x \text{ of } w (\#_a(x) = \#_b(x) = \#_c(x))\}$.
 - $\{w \in \{a, b, c\}^* : \exists \text{ some prefix } x \text{ of } w (x \neq \varepsilon \text{ and } \#_a(x) = \#_b(x) = \#_c(x))\}$.

- 3) Define the following two languages:
 $L_a = \{w \in \{a, b\}^* : \text{in each prefix } x \text{ of } w, \#_a(x) \geq \#_b(x)\}.$
 $L_b = \{w \in \{a, b\}^* : \text{in each prefix } x \text{ of } w, \#_b(x) \geq \#_a(x)\}.$
- a) Let $L_1 = L_a \cap L_b$. Is L_1 regular? Prove your answer.
b) Let $L_2 = L_a \cup L_b$. Is L_2 regular? Prove your answer.
- 4) For each of the following languages L , state whether L is regular or not and prove your answer:
a) $\{uww^Rv : u, v, w \in \{a, b\}^+\}.$
b) $\{xyz^Ry^R : x, y, z \in \{a, b\}^+\}.$
- 5) Use the Pumping Theorem to complete the proof, given in \mathbb{C} 743, that English isn't regular.
- 6) Prove *by construction* that the regular languages are closed under:
a) intersection.
b) set difference.
- 7) Prove that the regular languages are closed under each of the following operations:
a) $\text{pref}(L) = \{w : \exists x \in \Sigma^* (wx \in L)\}.$
b) $\text{suff}(L) = \{w : \exists x \in \Sigma^* (xw \in L)\}.$
c) $\text{reverse}(L) = \{x \in \Sigma^* : x = w^R \text{ for some } w \in L\}.$
d) letter substitution (as defined in Section 8.3).
- 8) Using the definitions of *maxstring* and *mix* given in Section 8.6, give a precise definition of each of the following languages:
a) $\text{maxstring}(A^nB^n).$
b) $\text{maxstring}(a^i b^j c^k, 1 \leq k \leq j \leq i).$
c) $\text{maxstring}(L_1 L_2)$, where $L_1 = \{w \in \{a, b\}^* : w \text{ contains exactly one } a\}$ and $L_2 = \{a\}.$
d) $\text{mix}((aba)^*).$
e) $\text{mix}(a^*b^*).$
- 9) Prove that the regular languages are not closed under *mix*.
- 10) Recall that $\text{maxstring}(L) = \{w : w \in L \text{ and } \forall z \in \Sigma^* (z \neq \varepsilon \rightarrow wz \notin L)\}.$
a) Prove that the regular languages are closed under *maxstring*.
b) If $\text{maxstring}(L)$ is regular, must L also be regular? Prove your answer.
- 11) Define the function $\text{midchar}(L) = \{c : \exists w \in L (w = ycz, c \in \Sigma_L, y \in \Sigma_L^*, z \in \Sigma_L^*, |y| = |z|)\}.$ Answer each of the following questions and prove your answer:
a) Are the regular languages closed under *midchar*?
b) Are the nonregular language closed under *midchar*?
- 12) Define the function $\text{twice}(L) = \{w : \exists x \in L (x \text{ can be written as } c_1 c_2 \dots c_n, \text{ for some } n \geq 1, \text{ where each } c_i \in \Sigma_L, \text{ and } w = c_1 c_1 c_2 c_2 \dots c_n c_n)\}.$
a) Let $L = (1 \cup 0)^*1$. Write a regular expression for $\text{twice}(L)$.
b) Are the regular languages closed under *twice*? Prove your answer.
- 13) Define the function $\text{shuffle}(L) = \{w : \exists x \in L (w \text{ is some permutation of } x)\}.$ For example, if $L = \{ab, abc\}$, then $\text{shuffle}(L) = \{ab, abc, ba, acb, bac, bca, cab, cba\}.$ Are the regular languages closed under *shuffle*? Prove your answer.
- 14) Define the function $\text{copyreverse}(L) = \{w : \exists x \in L (w = xx^R)\}.$ Are the regular languages closed under *copyandreverse*? Prove your answer.

- 15) Let L_1 and L_2 be regular languages. Let L be the language consisting of strings that are contained in exactly one of L_1 and L_2 . Prove that L is regular.
- 16) Define two integers i and j to be **twin primes** \iff both i and j are prime and $|j - i| = 2$.
- Let $L = \{w \in \{1\}^* : w \text{ is the unary notation for a natural number } n \text{ such that there exists a pair } p \text{ and } q \text{ of twin primes, both } > n.\}$ Is L regular?
 - Let $L = \{x, y : x \text{ is the decimal encoding of a positive integer } i, y \text{ is the decimal encoding of a positive integer } j, \text{ and } i \text{ and } j \text{ are twin primes}\}$. Is L regular?
- 17) Consider any function $f(L_1) = L_2$, where L_1 and L_2 are both languages over the alphabet $\Sigma = \{0, 1\}$. A function f is **nice** iff whenever L_2 is regular, L_1 is regular. For each of the following functions, f , state whether or not it is nice and prove your answer.
- $f(L) = L^R$.
 - $f(L) = \{w : w \text{ is formed by taking a string in } L \text{ and replacing all } 1\text{'s with } 0\text{'s and leaving the } 0\text{'s unchanged}\}$.
 - $f(L) = L \cup 0^*$.
 - $f(L) = \{w : w \text{ is formed by taking a string in } L \text{ and replacing all } 1\text{'s with } 0\text{'s and all } 0\text{'s with } 1\text{'s (simultaneously)}\}$.
 - $f(L) = \{w : \exists x \in L (w = x00)\}$.
 - $f(L) = \{w : w \text{ is formed by taking a string in } L \text{ and removing the last character}\}$.
- 18) We'll say that a language L over an alphabet Σ is **splitable** iff the following property holds: Let w be any string in L that can be written as $c_1c_2 \dots c_{2n}$, for some $n \geq 1$, where each $c_i \in \Sigma$. Then $x = c_1c_3 \dots c_{2n-1}$ is also in L .
- Give an example of a splitable regular language.
 - Is every regular language splitable?
 - Does there exist a nonregular language that is splitable?
- 19) Define the class IR to be the class of languages that are both infinite and regular. Is the class IR closed under:
- union.
 - intersection.
 - Kleene star.
- 20) Consider the language $L = \{x0^n y1^n z : n \geq 0, x \in P, y \in Q, z \in R\}$, where $P, Q,$ and R are nonempty sets over the alphabet $\{0, 1\}$. Can you find regular sets $P, Q,$ and R such that L is not regular? Can you find regular sets $P, Q,$ and R such that L is regular?
- 21) For each of the following claims, state whether it is *True* or *False*. Prove your answer.
- There are uncountably many non-regular languages over $\Sigma = \{a, b\}$.
 - The union of an infinite number of regular languages must be regular.
 - The union of an infinite number of regular languages is never regular.
 - If L_1 and L_2 are not regular languages, then $L_1 \cup L_2$ is not regular.
 - If L_1 and L_2 are regular languages, then $L_1 \otimes L_2 = \{w : w \in (L_1 - L_2) \text{ or } w \in (L_2 - L_1)\}$ is regular.
 - If L_1 and L_2 are regular languages and $L_1 \subseteq L \subseteq L_2$, then L must be regular.
 - The intersection of a regular language and a nonregular language must be regular.
 - The intersection of a regular language and a nonregular language must not be regular.
 - The intersection of two nonregular languages must not be regular.
 - The intersection of a finite number of nonregular languages must not be regular.
 - The intersection of an infinite number of regular languages must be regular.
 - It is possible that the concatenation of two nonregular languages is regular.
 - It is possible that the union of a regular language and a nonregular language is regular.
 - Every nonregular language can be described as the intersection of an infinite number of regular languages.
 - If L is a language that is not regular, then L^* is not regular.
 - If L^* is regular, then L is regular.
 - The nonregular languages are closed under intersection.
 - Every subset of a regular language is regular.

- s) Let $L_4 = L_1L_2L_3$. If L_1 and L_2 are regular and L_3 is not regular, it is possible that L_4 is regular.
- t) If L is regular, then so is $\{xy : x \in L \text{ and } y \notin L\}$.
- u) Every infinite regular language properly contains another infinite regular language.

9 Algorithms and Decision Procedures for Regular Languages

So far, we have considered five important properties of regular languages:

1. FSMs and regular expressions are useful design tools.
2. The fact that nondeterminism can be “compiled out” of an FSM makes it even easier, for many kinds of tasks, to design a simple machine that can relatively easily be shown to be correct.
3. DFSMs run in time that is linear in the length of the input.
4. There exists an algorithm to minimize a DFSM.
5. The regular languages are closed under many useful operators, so we can talk about programs that manipulate FSMs to construct new ones.

And now we will consider a sixth:

6. There exist decision procedures for many questions that we would like to ask about FSMs and regular expressions.

9.1 Fundamental Decision Procedures

Recall from Section 4.1 that a decision procedure is an algorithm whose result is a Boolean value. A decision procedure must be guaranteed to halt on all inputs and to return the correct value.

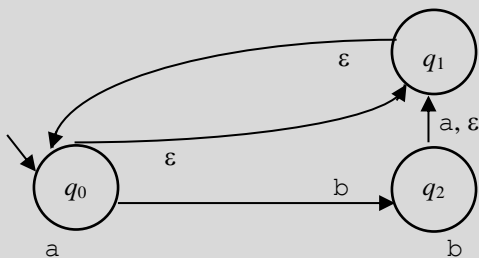
In this section, we describe some of the most useful decision procedures for regular languages:

9.1.1 Membership

Given an FSM M and a string w , does M accept w ? This is the most basic question we can ask about an FSM. It can be answered by running M on w , provided that we do so in a fashion that guarantees that the simulation halts. Recall that the simulation of an NDFSM M might not halt if M contains ϵ -loops that are not handled properly by the simulator.

Example 9.1 ϵ -Loops Can Cause Trouble in NDFSMs

If we are not careful, the simulation of the following NDFSM on input aa might get stuck chasing the ϵ -loop between q_0 and q_1 , never reading any input characters:



Theorem 9.1 Decidability of Regular Languages

Theorem: Given a regular language L (represented as an FSM or a regular expression or a regular grammar) and a string w , there exists a decision procedure that answers the question, is $w \in L$?

Proof: If L is represented as an FSM, we can answer the question using either of the simulation techniques described in Section 5.6. We’ll choose to use *ndfsmsimulate*:

```
decideFSM( $M$ : FSM,  $w$ : string) =  
  If ndfsmsimulate( $M$ ,  $w$ ) accepts then return True else return False.
```

Any question that can be answered about an FSM can be answered about a regular expression by first converting the regular expression into an FSM. So if L is represented as a regular expression α , we can answer the question, “Does α generate w ?” using the procedure *decideregex* defined as follows:

decideregex(α : regular expression, w : string) =

1. From α , use *regextofsm* to construct an FSM M such that $L(\alpha) = L(M)$.
2. Return *decideFSM*(M, w).

The same is true of regular grammars: any regular grammar G can be converted to an FSM that accepts $L(G)$. ■

While the solution to this problem was simple, the question itself is very important. We will see later that, in the case of some more powerful computational models (in particular the Turing machine), the basic membership question is not decidable. This fact is yet another powerful argument for the use of an FSM whenever one exists.

In the remainder of this discussion, we will focus on answering questions about FSMs. Each question that is decidable for FSMs is also decidable for regular expressions and for regular grammars because a regular expression or a regular grammar can be converted to an equivalent FSM.

9.1.2 Emptiness and Totality

The next question we will consider is, “Given an FSM M , is $L(M) = \emptyset$?” There are two approaches that we could take to answering a question like this about the overall behavior of M :

1. View M as a directed graph in which the states are the vertices and the transitions are directed edges. Find some property of the graph that corresponds to the situation in which $L(M) = \emptyset$.
2. Run M on some number of strings and observe its behavior.

Both work. We’ll first consider the approach in which we do a static analysis of M , without running it on any strings. We observe that $L(M)$ will be empty if K_M contains no accepting states. But then we realize that, for $L(M)$ not to be empty, it is not sufficient for there to be at least one accepting state. That state must be reachable, via some path, from the start state. So we can state the following algorithm for testing whether $L(M) = \emptyset$:

emptyFSMgraph(M : FSM) =

1. Mark all states that are reachable via some path from the start state of M .
2. If at least one marked state is an accepting state, return *False*. Else return *True*.

Another way to use the graph-testing method is to exploit the fact there exists a canonical form for FSMs. Recall that, in Section 5.8, we described the algorithm *buildFSMcanonicalform*, which built, from any FSM M , an equivalent unique minimal DFSM whose states are named in a standard way so that all equivalent FSMs will generate the same minimal deterministic machine. We can use that canonical form as the basis for a simple emptiness checker, since we note that $L(M)$ is empty iff the canonical form of M is the one-state FSM that accepts nothing. So we can define:

emptyFSMcanonicalgraph(M : FSM) =

1. Let $M\# = \text{buildFSMcanonicalform}(M)$.
2. If $M\#$ is the one-state FSM that accepts nothing, return *True*. Else return *False*.

A very different approach we could take to answering the emptiness question is to run M on some strings and see whether or not it accepts. We might start by running M on all strings in Σ^* to see if it accepts any of them. But there is an infinite number of possible strings (assuming that Σ_M is not empty). A decision procedure must be guaranteed to halt in a finite number of steps, even if the answer is *False*. But we make the same observation here that we used as the basis for the Pumping Theorem: if a DFSM M accepts any “long” strings, then it also accepts the strings that result from pumping out from those long strings the substrings that drove M through a loop. More precisely, if a DFSM M accepts any strings of length greater than or equal to $|K_M|$, then it must also accept at least one string of length less than $|K_M|$. In other words, it must accept at least one string without going through any loops. So we can define *emptyFSMsimulate*:

emptyFSMsimulate(M : FSM) =

1. Let $M' = \text{ndfsmtodfsm}(M)$.
2. For each string w in Σ^* such that $|w| < |K_M|$ do:
Run *decideFSM*(M', w).
3. If M' accepts at least one such string, return *False*; else return *True*.

This definition of *emptyFSMsimulate* exploits a powerful technique that we'll use in other decision procedures. We'll call it **bounded simulation**. It answers a question about $L(M)$ by simulating the execution of M . For bounded simulation to serve as the basis of a decision procedure, two things must be true:

- The simulation of M on a particular input string must be guaranteed to halt. DFSSMs always halt, so this requirement is easily met. We'll see later, however, that when we are considering more powerful machines, such as pushdown automata and Turing machines, this condition may not be satisfied.
- It must be possible to determine the answer we seek by simulating M on some *finite* number strings. So we need to be able to do an analysis, of the sort we did above, that shows that once we know how M works on some particular finite set of strings, we can conclude some more general property of its behavior.

The algorithms that we have just presented enable us to prove the following theorem:

Theorem 9.2 Decidability of Emptiness

Theorem: Given an FSM M , there exists a decision procedure that answers the question, is $L(M) = \emptyset$?

Proof: All three algorithms, *emptyFSMgraph*, *emptyFSMcanonicalgraph*, and *emptyFSMsimulate*, can easily be shown to be correct. We can pick any one of them and use it to define the procedure *emptyFSM*. We'll use *emptyFSMsimulate*:

emptyFSM(M : FSM) =
Return *emptyFSMsimulate*(M).

■

At the other extreme, we might like to ask the question, "Given an FSM M , is $L(M) = \Sigma^*$?" In other words, does M accept everything? The answer is yes iff $\neg L(M) = \emptyset$. So we have the following theorem:

Theorem 9.3 Decidability of Totality

Theorem: Given an FSM M , there exists a decision procedure that answers the question, is $L(M) = \Sigma^*$?

Proof: The following procedure answers the question:

totalFSM(M : FSM) =
1. Construct M' to accept $\neg L(M)$.
2. Return *emptyFSM*(M').

■

9.1.3 Finiteness

Suppose that $L(M)$ is not empty. Then we might like to ask, "Is $L(M)$ finite?" Again, we can attempt to answer the question either by analyzing M as a graph or by running it on strings.

Let's consider the graph approach first. $L(M)$ is clearly finite if M contains no loops. But the mere presence of a loop does not guarantee that $L(M)$ is infinite. The loop might be:

- labeled only with ϵ ,
- unreachable from the start state, or

- not on a path to an accepting state.

In any of those cases, the loop will not force M to accept an infinite number of strings. Taking all of those issues into account, we can build the following correct graph-based algorithm to answer the question:

```

finiteFSMgraph( $M$ : FSM) =
  1.  $M' = \text{ndfsmtodfsm}(M)$ .
  2.  $M'' = \text{minDFSM}(M')$ . /* At this point, there are no  $\epsilon$ -transitions and no unreachable states.
  3. Mark all states in  $M''$  that are on a path to an accepting state.
  4. Considering only marked states, determine whether there are any cycles in  $M''$ .
  5. If there are cycles, return True. Else return False.

```

While it is possible, as we have just seen, to design algorithms to answer questions about FSMs by analyzing them as graphs, it is quite easy to make mistakes, as we would have done had we not considered the three cases in which a loop does not mean that an infinite number of strings can be accepted.

It is often easier to design an algorithm and prove its correctness by appealing to the simulation strategy instead. Pursuing that approach, it may be tempting to try to answer the finiteness question by running M on all possible strings to see if it ever stops accepting. But, again, we can only use simulation in a decision procedure if we can put an upper bound on the amount of simulation that is required. Fortunately, we can do that in this case. Again we appeal to the argument that we used to prove the Pumping Theorem. We begin by making M deterministic so that we do not have to worry about ϵ -loops. Then observe that $L(M)$ is infinite iff it contains any strings that force M through some loop. Any string of length greater than $|K_M|$ must force M through a loop. So, if M accepts even one string of length greater than $|K_M|$, then $L(M)$ is infinite. Note also that if $L(M)$ is infinite then it contains no longest string. So it must contain an infinite number of strings of length greater than $|K_M|$. So $L(M)$ is infinite iff M accepts even one string of length greater than $|K_M|$.

Unfortunately, there is an infinite number of such long strings. So we cannot try them all. But suppose that M accepts some “very long” string, i.e., one that forces M through a loop twice. Then we could pump out the substring that corresponds to the first time through the loop. We’d then have a shorter string that is also accepted by M . So if M accepts any strings that force it through a loop twice, it must also accept at least one string that forces it through a loop only once. The longest loop M could contain would be one that drives it through all its states a second time. So, $L(M)$ is infinite iff M accepts at least one string w where:

$$|K_M| \leq |w| \leq 2 \cdot |K_M| - 1.$$

We can now define a simulation-based procedure to determine whether $L(M)$ is finite:

```

finiteFSMsimulate( $M$ : FSM) =
  1.  $M' = \text{ndfsmtodfsm}(M)$ .
  2. For each string  $w$  in  $\Sigma^*$  such that  $|K_M| \leq |w| \leq 2 \cdot |K_M| - 1$  do
      Run decideFSM( $M'$ ,  $w$ ).
  3. If  $M'$  accepts at least one such string, return False (since  $L$  is infinite and thus not finite); else return True.

```

Theorem 9.4 Decidability of Finiteness

Theorem: Given an FSM M , there exists a decision procedure that answers the questions, “Is $L(M)$ finite?” and “Is $L(M)$ infinite?”

Proof: We can pick either *finiteFSMgraph* or *finiteFSMsimulate* and use it to define the procedure *finiteFSM*:

```

finiteFSM( $M$ : FSM) =
  Return finiteFSMsimulate( $M$ ).

```

Of course, if we can decide whether $L(M)$ is finite, we can decide whether it is infinite:

infiniteFSM(M : FSM) =
 Return $\neg(\text{finiteFSMsimulate}(M))$.

■

9.1.4 Equivalence

Given two FSMs M_1 and M_2 , are they equivalent? In other words, is $L(M_1) = L(M_2)$? We can describe two different algorithms for answering this question.

The first algorithm takes advantage of the existence of a canonical form for FSMs. It works as follows:

equalFSMs₁(M_1 : FSM, M_2 : FSM) =

1. $M_1' = \text{buildFSMcanonicalform}(M_1)$.
2. $M_2' = \text{buildFSMcanonicalform}(M_2)$.
3. If M_1' and M_2' are equal, return *True*, else return *False*.

The second algorithm depends on the following observation: Let L_1 and L_2 be the languages accepted by M_1 and M_2 . Then M_1 and M_2 are equivalent iff $(L_1 - L_2) \cup (L_2 - L_1) = \emptyset$. Since the regular languages are closed under difference and union, we can build an FSM to accept $(L_1 - L_2) \cup (L_2 - L_1)$. We can then test to see whether that FSM accepts any strings. So we have:

equalFSMs₂(M_1 : FSM, M_2 : FSM) =

1. Construct M_A to accept $L(M_1) - L(M_2)$.
2. Construct M_B to accept $L(M_2) - L(M_1)$.
3. Construct M_C to accept $L(M_A) \cup L(M_B)$.
4. Return *emptyFSM*(M_C).

Theorem 9.5 Decidability of Equivalence

Theorem: Given two FSMs M_1 and M_2 , there exists a decision procedure that answers the questions, “Is $L(M_1) = L(M_2)$?”

Proof: We can pick the approach of either *equalFSMs₁* or *equalFSMs₂* and use it to define the procedure *equalFSMs*. Choosing *equalFSMs₂*, we get:

equalFSMs(M_1 : FSM, M_2 : FSM) =
 Return *equalFSMs₂*(M_1, M_2).

■

9.1.5 Minimality

Theorem 9.6 Decidability of Minimality

Theorem: Given a DFSM M , there exists a decision procedure that answers the question, “Is M minimal?”

Proof: The proof is by construction. We define:

minimalFSM(M : FSM) =

1. $M' = \text{minDFSM}(M)$.
2. If $|K_M| = |K_{M'}|$ return *True*; else return *False*.

■

Note that it is easy to modify *minimalFSM* so that, if M is not minimal, it returns $|K_M| - |K_{M'}|$.

9.1.6 Combining the Basics to Ask Specific Questions

With these fundamental decision algorithms in hand, coupled with the other functions (such as *ndfsmtodfsm* and *minDFSM*) that we have also defined, it is possible to answer a wide range of specific questions that might be of interest in a particular context.

Example 9.2 Combining Algorithms and Decision Procedures

Suppose that we would like to know, for two arbitrary patterns, whether there are any nontrivial (which we may define, for example, as not equal to ε) strings that could match both patterns. This might come up if we are attempting to categorize strings in such a way that no string falls into more than one category. We can formalize that question as, “Given two regular expressions α_1 and α_2 , is $(L(\alpha_1) \cap L(\alpha_2)) - \{\varepsilon\} \neq \emptyset$?” An algorithm to answer that question is:

1. From α_1 , construct an FSM M_1 such that $L(\alpha_1) = L(M_1)$.
2. From α_2 , construct an FSM M_2 such that $L(\alpha_2) = L(M_2)$.
3. Construct M' such that $L(M') = L(M_1) \cap L(M_2)$.
4. Construct M_ε such that $L(M_\varepsilon) = \{\varepsilon\}$.
5. Construct M'' such that $L(M'') = L(M') - L(M_\varepsilon)$.
6. If $L(M'')$ is empty return *False*; else return *True*.

9.2 Summary of Algorithms and Decision Procedures for Regular Languages

Sprinkled throughout our discussion of regular languages has been a collection of algorithms that can be applied to FSMs, regular expressions, and regular grammars. Together, those algorithms make it possible to:

- Optimize FSMs.
- Construct new FSMs and regular expressions from existing ones, thus enabling us to decompose complex problems into simpler ones and to reuse code that has already been written.
- Answer a wide variety of questions about any regular language or about the class of regular languages.

Because there are so many of these algorithms and they have been spread out over several chapters, we present a concise list of them here:

- Algorithms that operate on FSMs without altering the language that is accepted:
 - *Ndfsmtodfsm*: Given an NDFSM M , construct a DFSM M' such that $L(M) = L(M')$.
 - *MinDFSM*: Given a DFSM M , construct a minimal DFSM M' , such that $L(M) = L(M')$.
- Algorithms that compute functions of languages defined as FSMs:
 - Given two FSMs M_1 and M_2 , construct a new FSM M_3 such that $L(M_3) = L(M_2) \cup L(M_1)$.
 - Given two FSMs M_1 and M_2 , construct a new FSM M_3 such that $L(M_3) = L(M_2) L(M_1)$ (i.e., the concatenation of $L(M_2)$ and $L(M_1)$).
 - Given an FSM M , construct a new FSM M' such that $L(M') = (L(M))^*$.
 - Given an FSM M , construct a new FSM M' such that $L(M') = \neg L(M)$.
 - Given two FSMs M_1 and M_2 , construct a new FSM M_3 such that $L(M_3) = L(M_2) \cap L(M_1)$.
 - Given two FSMs M_1 and M_2 , construct a new FSM M_3 such that $L(M_3) = L(M_2) - L(M_1)$.
 - Given an FSM M , construct a new FSM M' such that $L(M') = (L(M))^R$ (i.e., the reverse of $L(M)$).
 - Given an FSM M , construct an FSM M' that accepts *letsub*($L(M)$), where *letsub* is a letter substitution function.
- Algorithms that convert between FSMs and regular expressions:
 - Given a regular expression α , construct an FSM M such that $L(\alpha) = L(M)$.
 - Given an FSM M , construct a regular expression α such that $L(\alpha) = L(M)$.
- Algorithms that convert between FSMs and regular grammars:
 - Given a regular grammar G , construct an FSM M such that $L(G) = L(M)$.

- Given an FSM M , construct a regular grammar G such that $L(G) = L(M)$.
- Algorithms that implement operations on languages defined by regular expressions or regular grammars: any operation that can be performed on languages defined by FSMs can be implemented by converting all regular expressions or regular grammars to equivalent FSMs and then executing the appropriate FSM algorithm.
- Decision procedures that answer questions about languages defined by FSMs:
 - Given an FSM M and a string w , is w accepted by M ?
 - Given an FSM M , is $L(M) = \emptyset$?
 - Given an FSM M , is $L(M) = \Sigma^*$?
 - Given an FSM M , is $L(M)$ finite (or infinite)?
 - Given two FSMs, M_1 and M_2 , is $L(M_1) = L(M_2)$?
 - Given a DFSM M , is M minimal?
- Decision procedures that answer questions about languages defined by regular expressions or regular grammars: Again, convert the regular expressions or regular grammars to FSMs and apply the FSM algorithms.

This list is important and it represents a strong argument for describing problems as regular languages and solutions as FSMs or regular expressions. As we will soon see, a few of these algorithms (but not most) exist for context-free languages and their associated representations (as pushdown automata or as context-free grammars). None of them exists for general purpose programming languages or Turing machines.

At this point, we are concerned primarily with the existence of the algorithms that we need. In Part V, we'll expand our inquiry to include the complexity of the algorithms that we have found. But we can note here that not all of the algorithms that we have presented so far are efficient in the common sense of running in time that is polynomial in the length of the input. For example, *ndfsmtodfsm* may construct a DFSM whose size grows exponentially in the size of the input NDFSM. Thus its time requirement (in the worst case) is also exponential.

9.3 Exercises

- 1) Define a decision procedure for each of the following questions. Argue that each of your decision procedures gives the correct answer and terminates.
 - a) Given two DFSMs M_1 and M_2 , is $L(M_1) = L(M_2)^R$?
 - b) Given two DFSMs M_1 and M_2 is $|L(M_1)| < |L(M_2)|$?
 - c) Given a regular grammar G and a regular expression α , is $L(G) = L(\alpha)$?
 - d) Given two regular expressions, α and β , do there exist any even length strings that are in $L(\alpha)$ but not $L(\beta)$?
 - e) Let $\Sigma = \{a, b\}$ and let α be a regular expression. Does the language generated by α contains all the even length strings in Σ^* .
 - f) Given an FSM M and a regular expression α , does M accept exactly two more strings than α generates?
 - g) Let $\Sigma = \{a, b\}$ and let α and β be regular expressions. Is the following sentence true:

$$(L(\beta) = a^*) \vee (\forall w (w \in \{a, b\}^* \wedge |w| \text{ even}) \rightarrow w \in L(\alpha)).$$

- h) Given a regular grammar G , is $L(G)$ regular?
- i) Given a regular grammar G , does G generate any odd length strings?

10 Summary and References

Theoretically, every machine we build is a finite state machine. There is only a finite number (probably about 10^{79}) of atoms in the causal universe \square (that part of the universe that is within a distance of the speed of light times the age of the universe). So we have access to only a finite number of molecules with which to build computer memories, hard drives, and external storage devices. That doesn't mean that every real problem should be described as a regular language or solved with an FSM. FSMs and regular expressions are powerful tools for describing problems that possess the kind of repetitive patterns that FSMs and regular expressions can capture. To handle other problems and languages, we will need the more powerful models that we will introduce in Parts III and IV. The abstract machines that are built using those models will be equipped with infinite storage devices. Describing problems using those devices may be useful even if there exists some practical upper bound on the size of the actual inputs that need to be considered (and so some bound on the amount of memory required to solve the problem).

A lighthearted view of the theory of automata and computability has inspired a collection of poems \square by Martin Cohn and Harry Mairson. We include one of the poems here. Unfortunately, the names of the important concepts aren't standard and the poem uses some that are different from ours. So:

- DFA (Deterministic Finite Automaton) is equivalent to DFSA.
- The symbol p is used as we used k in the Pumping Theorem.
- The term r.e. (recursively enumerable), in the last line, refers to the class of languages we are calling semidecidable.

The Pumping Lemma for DFAs By Martin Cohn

Any regular language L has a magic number p
And any long-enough 'word' in L has the following property:
Amongst its first p symbols is a segment you can find
Whose repetition or omission leaves 'word' amongst its kind.

So if you find a language L which fails this acid test,
And some long word you pump becomes distinct from all the rest,
By contradiction you have shown that language L is not
A regular L , resilient to the damage you have wrought.

But if, upon the other hand, 'word' stays within L ,
Then either L is regular, or else you chose not well.
For 'word' is parsed as xyz , and y cannot be null,
And y must come before p symbols have been read in full.

You cannot choose the length of y , nor can you specify
Just where within the word you chose it happens just to lie.
The DFA locates string y to your discomfiture.
Recall this moral to the grave: You can't fool Mother Nature.

As postscript mathematical, addendum to the wise:
The basic proof we outlined here does surely generalize.
So there's a pumping lemma for languages context-free,
But sadly we don't have the same for those that are r.e.

References

The idea of a finite state computer grew out of an early (i.e., predating modern computers) attempt [McCulloch and Pitts 1943] to describe the human brain as a logical computing device. The artificial neuron model \square described in that paper inspired the development of the modern neural networks that play an important role in artificial intelligence systems today. It also laid the groundwork for the development of the general model of finite state computing that we

have discussed. About a decade after the McCulloch and Pitts paper, several independent formulations of finite state computers appeared. Mealy and Moore machines were defined in [Mealy 1955] and [Moore 1956], respectively. [Kleene 1956] described the McCulloch and Pitts neurons as FSMs. It also defined regular expressions and then proved the result that we state as Theorem 6.3 and call Kleene's Theorem, namely that the class of languages that can be defined by regular expressions is identical to the class that can be accepted by finite state machines.

Many of the early results in finite automata, including Theorem 5.3 (that, for every nondeterministic FSM there exists an equivalent deterministic one) were given in [Rabin and Scott 1959]. For this work, Rabin and Scott received the 1976 Turing Award. The citation read, "For their joint paper "Finite Automata and Their Decision Problem," which introduced the idea of nondeterministic machines, which has proved to be an enormously valuable concept. Their classic paper has been a continuous source of inspiration for subsequent work in this field."

The definition of the missing letter language that we discussed in Example 5.12 and Example 5.15 and the proof given in [§ 627](#) for the correctness of *ndfsmto fsm* were taken from [Lewis and Papadimitriou 1998].

[Aho and Corasick 1975] presents a set of algorithms for building a finite state transducer that finds and reports all instances of a set of keywords in a target string. The algorithm *buildkeywordFSM* is derived from those algorithms, so the details of how it works can be found in the original paper.

The Myhill-Nerode Theorem was proved in [Myhill 1957] and [Nerode 1958].

Markov chains were first described (in Russian) by A. A. Markov in 1906. The mathematical theory of Hidden Markov models was described in [Baum, Petrie, Soules and Weiss 1970]. The Viterbi algorithm was presented in [Viterbi 1967].

Büchi automata were described in [Büchi 1960a] and [Büchi 1960b]. For a comprehensive discussion of them, as well as other automata on infinite strings, see [Thomas 1990] or [Khoussainov and Nerode, 2001]. The Büchi automaton that describes the mutual exclusion property and that we presented in Example 5.39 is taken from [Clarke, Grumberg and Peled 2000], which is a good introduction to model checking. The proof we presented for Theorem 5.7 is taken from [Roggenbach 2002], which presents a comprehensive discussion of nondeterminism in ω -automata, including a discussion of alternative models, including Muller and Rabin automata. Theorem 5.8 was stated in [Büchi 1960a]. Our presentation of it and of Theorem 5.9 and Theorem 5.10 is taken from [Thomas 1990], which supplies more details.

Regular grammars were defined as part of what we now call the Chomsky hierarchy (see Section **Error! Reference source not found.**) in [Chomsky 1959]. The equivalence of FSMs and regular grammars was shown in [Chomsky and Miller 1958].

The Pumping Theorem for regular languages (along with one for context-free languages that we will discuss in Section 13.3) was stated and proved in [Bar-Hillel, Perles and Shamir 1961]. The Pumping Theorem proof in Example 8.15 was taken from [Sipser 2006].