The Theory of Computation
Bernard M.E. Moret
Addison-Wesley-Longman, 1998

Solution Manual
Chapter 3, Text Problems

Exercise 3.1

Flipping the labels assigned to the states in order to complement the language accepted by a finite automaton works in all cases—at least, all cases within the range of behavior seen so far. Each input string takes the automaton through a well defined sequence of states; that sequence does not depend on the accepting/rejecting labels assigned to the states. If there is a unique sequence corresponding to each string, then flipping the labels does what we want: every sequence that was an accepting sequence becomes a rejecting one (and thus every string in the original language gets rejected) and every sequence that was a rejecting sequence becomes an accepting one (and thus every string not in the original language gets accepted). Problems can only arise when there are several possible sequences for a given input strings, with some accepting and some rejecting—a configuration that belongs to nondeterministic automata.

Exercise 3.2

We have seen that regular languages are closed under complementation. Thus a pumping lemma for strings not in some given regular language is a pumping lemma for the complement of that language, itself another regular language—so that such a pumping lemma should indeed exist. In fact, we can simply take the normal lemma and exchange throughout "is a member of" and "is not a member of". Let L be our regular language. For strings not in L, we choose such a string z of sufficient length, let the adversary break it into substrings $z = uvw$, and demonstrate the existence of an i such that $z' = uv^i w$ is in the language, thereby showing that the string z, although long enough to force a putative automaton of n states to loop, cannot be pumped and always stay outside the language, which implies that the complement of the language (and thus the language itself) is not regular.

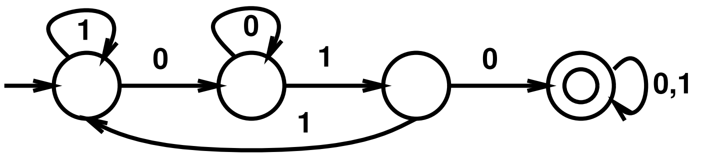
We can use this version of the pumping lemma to prove that a language is not regular by proving directly that its complement is not regular—although using our normal pumping lemma together with our knowledge of the closure of regular languages under complementation has exactly the same effect.

Exercise 3.3

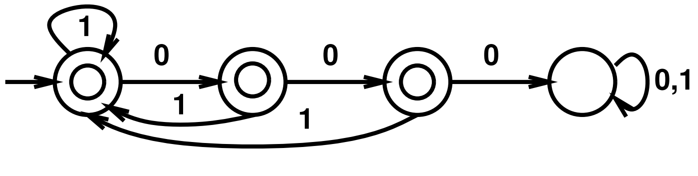
- The quotient L/L_2 is by definition the set $\{x \mid \exists y \in L_2, xy \in L\}$. Since L_2 includes ε, we can choose $y = \varepsilon$ and thus conclude that the quotient includes the subset $\{x \mid x\varepsilon \in L\}$, which is just the set $\{x \mid x \in L\}$, which is L itself.
- The quotient Σ^*/L is by definition the set $\{x \mid \exists y \in L, xy \in \Sigma^*\}$, which is just the set $\{x \mid \exists y \in L\}$—because the last requirement is met by any string. But that set is either the empty set (when L is empty) or the universe (when L is not empty).
- The quotient L/Σ^* is by definition the set $\{x \mid \exists y \in \Sigma^*, xy \in L\}$, that is, the set of strings that can be completed through the concatenation of any string at all into a string of L—the definition of the set of prefixes of the language L.

Exercise 3.4

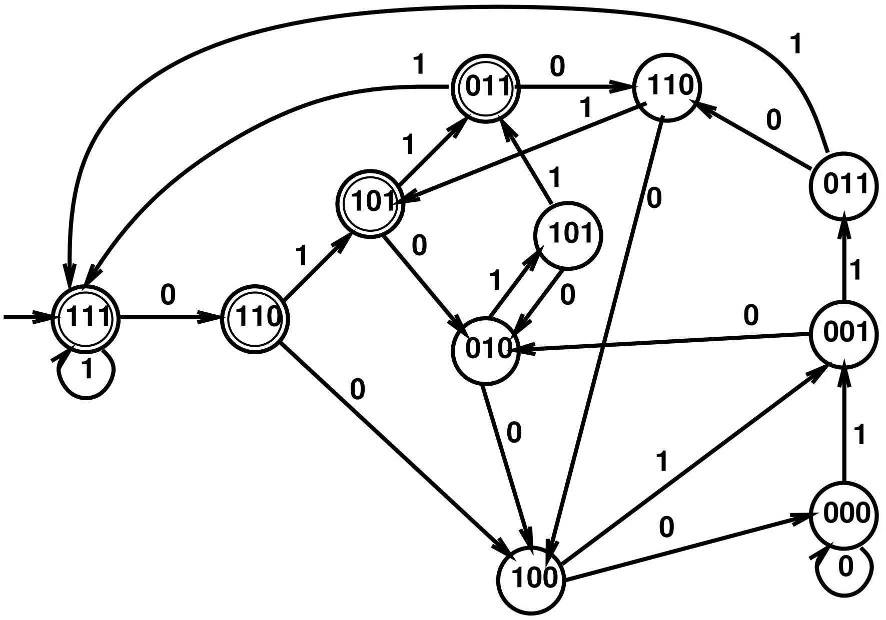
1. All that the automaton needs to do is to "watch" for the string 010 with three states, moving to an accepting trap as soon as the substring has been observed; so the machine has 4 states in all, one of them accepting, as shown below.



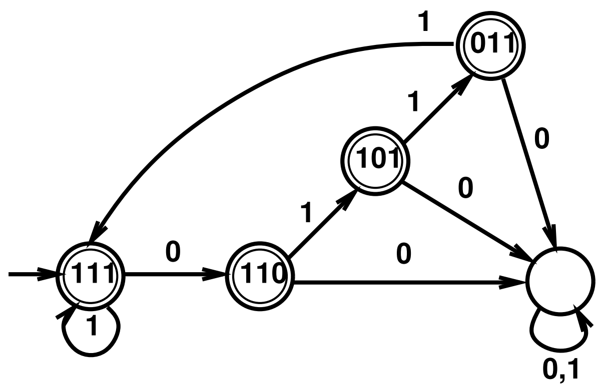
2. The same idea as in the previous problem works, but this time, once the critical substring has been observed, the FA moves to a rejecting trap. Again, we have 4 states, but now 3 of the four are accepting.



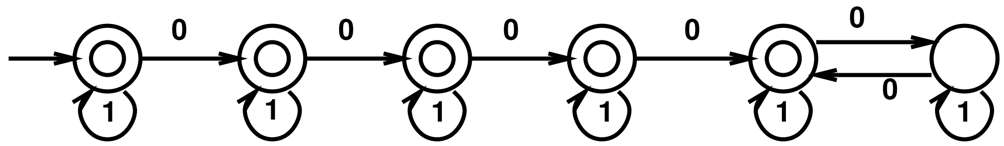
3. Before solving this problem, we need to finish defining it. The statement does not say what to do with strings of length 3 or less; since that leaves us free to make our own choices, we pick what will simplify the final FA as much as possible. Since the final FA keeps a window of 3 characters in its state memory (8 states, defining what has been seen so far, so that a move to the fourth character in any substring simply shifts the 3-character window), we let acceptance/rejection of short strings be defined by that mechanism. Note that windows with two 1s and one 0 exist in both an accepting and a rejecting version, while the window with three 1s exists only as an accepting window, and windows with two or three 0s exist only in rejecting versions. Thus we have 11 states in all, 4 of them accepting. We arbitrarily choose to start the machine in a state corresponding to an illusory window of 3 1s, i.e., an accepting state; thus all strings can be regarded as prefixed by three 1s, which means that our short strings (of length less than 4) are handled as follows: we accept all those that have at most one 0 and reject all those that have two or three 0s. (Note that forcing all strings of length less than 4 to be treated in the same way—all rejected or all accepted—considerably increases the length of the FA, since now the first three characters must be treated separately from everything else, giving us an additional 15 states . . .)



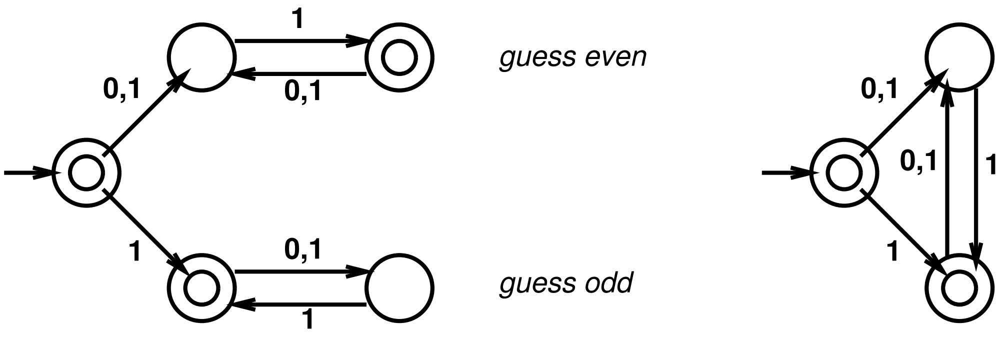
However, this machine does not really do what we are asked: it is forgiving, in that it is willing to go past a substring of length 4 with at most two 1s and still accept later . . . IN fact, we want to reject as soon as we move to a non-accepting state, so we can merge all our non-accepting states into a single trap, which cuts down the machine to just 5 states!



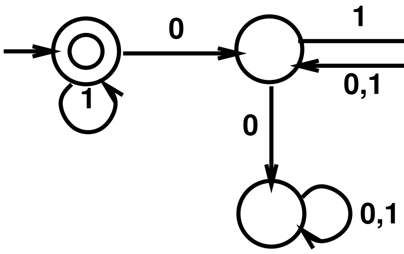
4. In spite of the apparent choice (odd or even), this machine is quite simple. It watches for 0s and counts up to 3 of them, accepting anything; after that, it stops counting, but checks parity of the number of 0s. This gives us a machine with 4 counting states (zero through three 0s) and 2 parity checking states, or 6 states in all, one of them rejecting.



5. It would be much simpler here to give a nondeterministic FA, since an NFA can guess whether the string has odd or even length and thus have two paths from the start state, one for odd-indexed checks and one for even-indexed checks. The resulting machine has five states and is very simple indeed; it can be further simplified down to just three states as seen on the right. (Note that we used the convention that an absence of applicable transitions implies immediate rejection.)

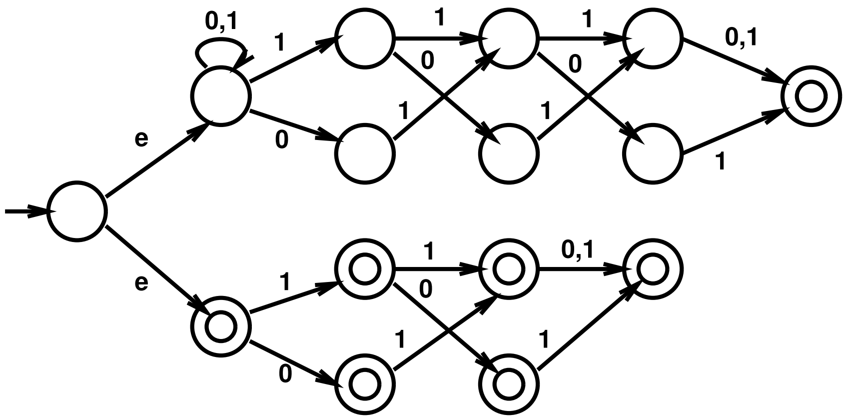


The deterministic version, however, must accomplish both tasks at once; Perhaps a better way to think about the problem is to realize that, as long as we see only 1s, we can accept both odd- and even-length strings; the first 0, however, forces the issue.

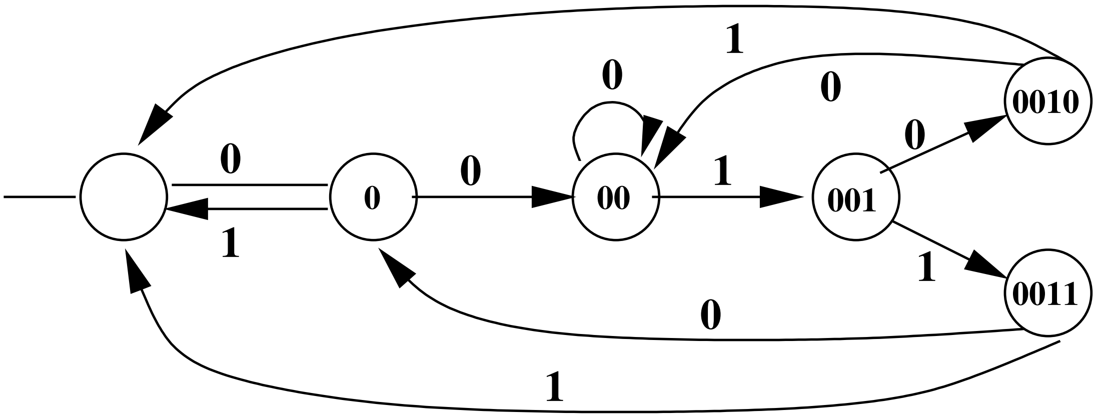


Exercise 3.5

1. This definition is unambiguous; since we have to keep track of the last four characters, it is probably easiest to design an NFA. We can have the NFA guess when 4 characters remain and check that they do not contain a pair of adjacent 0s; short strings (3 or fewer characters) need independent checking, so our NFA can start by guessing whether the string is short or long.

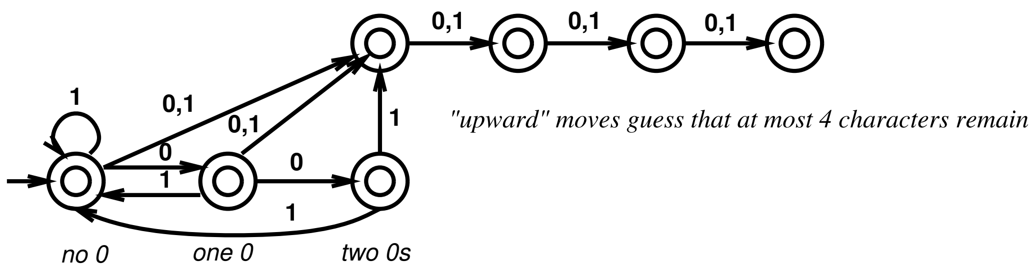


Actually, a very different outlook can net us an elegant and concise DFA: maintain a window of 4 characters, in which we want to avoid seeing two adjacent 0s. The resulting machine has just 6 states and is purely deterministic, as illustrated below.



2. This definition is somewhat ambiguous, because a pair of 0s could cross the "boundary", i.e., be in the 5th and 4th characters from the end, in which case we do not know whether to count it or not. We choose to count it, i.e., to consider it not covered by the clause about the last 4 characters, so that, for instance, the string 000100 will be rejected.

   We can design an NFA that checks for adjacent pairs of 0s (essentially, any three 0s in a row) and can guess at any time that it is entering the last four characters. Since the basic NFA has effectively 3 states (0, 1, and 2 0s in a row), we can have it move on any character from the first two states to a final string of 3 states that simply swallow four characters in a row; by not having a nondeterministic move on input 0 from the third state (already seen two 0s in a row), we ensure that strings with three 0s in a row where the third 0 is the 4th character from the end will be rejected.



Exercise 3.6

What all of these sets have in common is that they are finite. Every finite set is regular, hence all three sets in the problem are regular.

Exercise 3.7

1. This machine is a bit hard to describe, because its two loops have different lengths and can both be part of accepting paths. Fortunately, each loop is simple: one loop eats strings of length $3i$ that have a 0 at every third position and leads to acceptance of strings of length $3k$, for $k \geq 1$, where every third character is a 0 except the last, which is a 1; the other loop leads to acceptance of strings of length $4k + 3$ for $k \geq 0$, where the third character and every fourth character thereafter is a 1. Fortunately also, the two loops can be thought of as starting at the same state.

   Note that, regardless of which loops are used, acceptance requires a traversal of the four states and thus any accepted string ends with a block of three characters, the first two arbitrary and the last a 1. Every time we go through a loop, we prefix this block by a block of three characters, the first two arbitrary and the third a 0 (the short loop) or by a block of four characters, the first two and the fourth arbitrary and the third a 1. (Thus accepted strings can be of arbitrary length, except for 0, 1, 2, 4, 5, and 8.) A student remarked that we could think of the blocks of 3 as octal digits; in that case, the machine accepts all strings composed of octal digits, such that the last digit is odd and such that all other odd digits are always followed by one (arbitrary) extra bit.

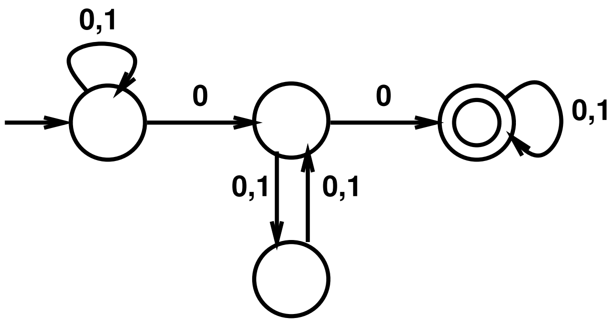
2. This machine accepts strings of length $3k$, for $k \geq 0$, where every substring of characters at positions $3i$, $3i+1$, and $3i+2$ (i.e., every block of 3 characters starting on multiples of 3) contains two 0s and one 1.
3. This machine accepts strings of 0s of length $5k+2$, for $k \geq 0$.

Exercise 3.8

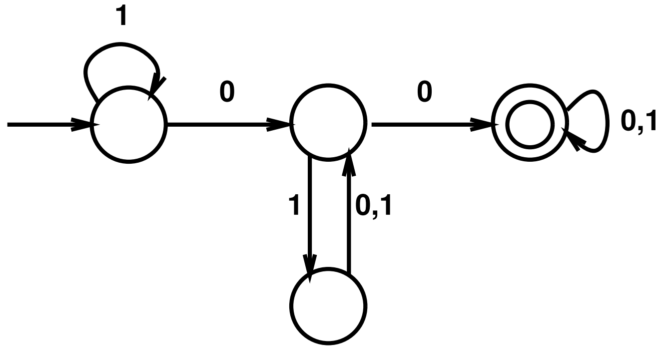
1. Since every finite set is regular, the answer is clearly yes: any non-empty set contains a non-empty finite set, even if that set contains a single item.
2. A language with nonempty complement is simply a language that is not Σ^*. Note that, because regular languages are closed under complementation and because every finite set is regular, every set of the form $\Sigma^* - F$, where F is a finite set, is regular. But every language with nonempty complement is contained within one such set—even if F is made of a single element. Thus the assertion is again proved.

Exercise 3.9

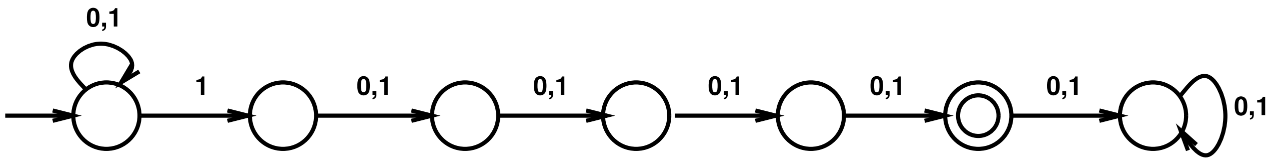
1. The nondeterministic automaton is quite simple: it chooses the two 0s, checking that they are separated by an even number of symbols. The



   deterministic automaton is essentially the same as the nondeterministic one, but requires a bit of insight. Once we have seen the first 0, we simply wait for another 0 to turn up an even number of positions later. This works because, should a 0 turn up an odd number of positions later, we could "use" either the first 0 or the "new" one as the starting point—they have the same relative parity in the string. The result is depicted below.
2. The nondeterministic automaton simply eats up symbols until it guesses that the fifth symbol from the end has arrived, at which point

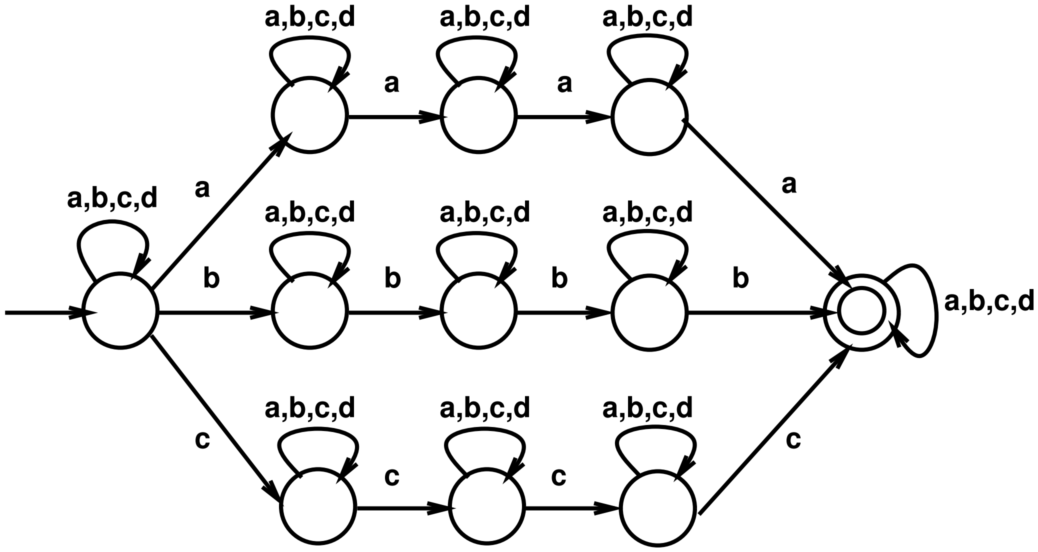


it checks that this symbol is a 1 and that it is followed by exactly four more symbols; at that point it can accept the string, but any additional symbol causes rejection. The deterministic automaton is quite painful



to set up. It has to remember the positions of the last few 1s it has seen—in a window of size five. If it reaches the end of the string in a state corresponding to having seen a 1 five positions back, it accepts, otherwise it rejects. Until then, it moves between the various states representing the possible windows of size five; since there are 32 such windows, we need a lot of states—since that is not quite sufficient, as we also have a series of states for initialization (to build the first five characters in the window), all rejecting. Overall, then, we can easily build the DFA, but it will have 63 states—which explains why we do not illustrate it here ... (Actually, it is easy to see that we need only 32 states, because we need only keep track of a window of 4 symbols—the next symbol's handling is automatic; this gives us 16 window states, 15 initializing states, and a final state—needed here, since no window state can be final.)

3. The nondeterministic automaton simply guesses which of the 3 symbols will be the one to appear at least 4 times in all and proceeds to verify that. A deterministic automaton is easy enough to build, but will need to keep track of the number of *a*s, of *b*s, and of *c*s seen so



far, until one of them has been seen 4 times. So we will have states of the type "seen x as, y bs, and z cs," for $0 \leq x, y, z \leq 3$, or at least $4^3 = 64$ states. The machine must have one more state for accepting and thus will have 65 states in all. Whenever it sees a d, it simply loops in place; whenever it sees one of the three symbols of interest, it moves to the state denoting the updated count, until it reaches the accepting state, which is a trap.

Exercise 3.10

If the FA does not accept ε, nothing needs to be done. If the FA does accept ε, one could conceivably make the start state (and any state reachable from it through an ε transition) non-accepting, but that will only work if none of these states is part of a loop that can lead to acceptance. So, a general method is to duplicate the start state and all states reachable from it through ε transitions, keeping the original accepting or not as they were, but making all duplicates rejecting. The new start state is the duplicate (nonaccepting) and moves through ε transitions, if any, to duplicates; any transition on an input symbol from a duplicate goes to an original state.

Exercise 3.11

The challenge comes from the need to keep the machine deterministic. Otherwise, it would be extremely simple: just add a new start state and a ε

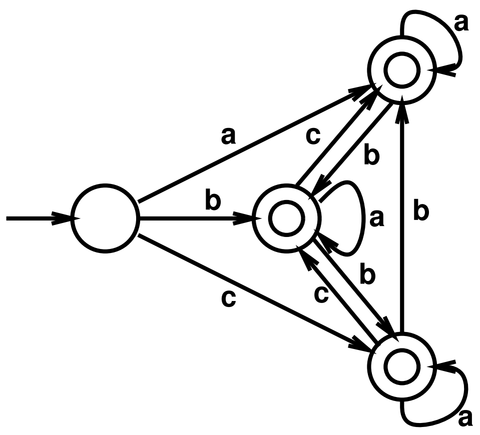
transition from it to the original start state. However, it is not much harder to do it deterministically. Let the original DFA be given by the quintuple $(\Sigma, Q, q_0, F, \delta)$. We define a new DFA as follows. The new set of states is $Q' = Q \cup \{q'\}$, with $q' \notin Q$; the new start state is q'; the set of accepting states is unchanged if we had $q_0 \notin F$ and is $F \cup \{q'\}$ otherwise; and the new transition function δ' includes all of the transitions of δ, plus the transitions $\delta'(q', a) = \delta(q_0, a)$ for each $a \in \Sigma$.

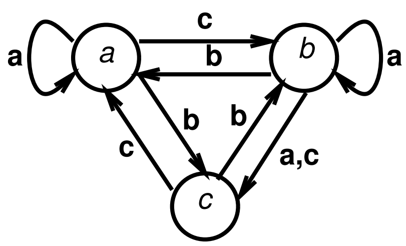
Obviously, the start state, once left, will never be re-entered, since no transition in the entire machine leads to it. Equally obviously, our new machine is deterministic whenever the original one is. To verify that our new machine accepts exactly the same language as the original, we observe first that it accepts ε exactly when the original does (because of our definition of the set of accepting states), then that, after reading one character, it finds itself in the "old machine"—that is, in a state of Q, from which point its behavior is identical to that of the old machine. Moreover, that state reached after one transition is the same as the old machine would have reached, because of our definition of $\delta'(q', a)$.

Exercise 3.12

As an example, assume that the string is *abbbcc*; in a left-to-right evaluation, we ask for the value of the expression $((((a \cdot b) \cdot b) \cdot b) \cdot c) \cdot c$; in a right-to-left evaluation, we ask for the value of the expression $a \cdot (b \cdot (b \cdot (b \cdot (c \cdot c))))$.

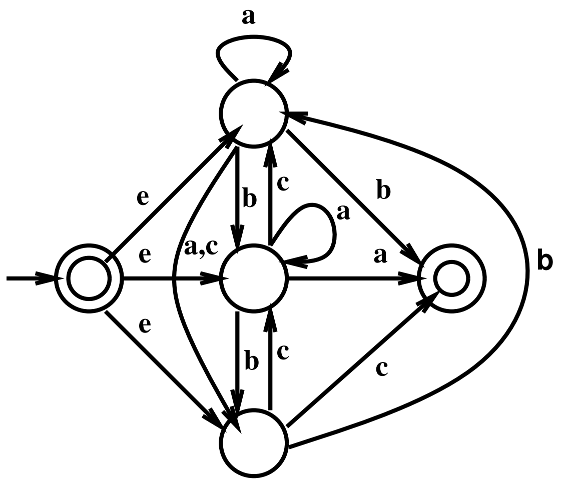
Left-to-right evaluation of the string is straightforward for either a DFA or an NFA: we simply keep the current value of the expression stored in states (hence 3 states) and move between these states as the next symbol dictates. For instance, if we are in "state a" and see the input symbol c, we move to "state b," corresponding to the operation $a \cdot c = b$. The DFA has 4 states in all, as shown below. The challenge—and the reason why an NFA is much preferable—is the right-to-left evaluation; indeed, it may appear impossible at first, since the machine gets the symbols in the worst possible order and cannot store them. Doing the right-to-left evaluation is not too bad once one gets the key insight: the NFA will guess every intermediate result and will verify that it could have been produced as guessed once the operands are both known. In our example, the NFA first guesses the value of the entire expression, say x, then guesses a second operand y such that $a \cdot y = x$; y is now the guessed value for the subexpression $b \cdot (b \cdot (b \cdot (c \cdot c)))$ and we proceed in the same way, guessing some second operand z such that $b \cdot z = y$. For our tables, this gives the following NFA fragment, which is entered at any of its 3 states; we have labelled each state with the value that denotes the corresponding guess for the remaining substring.

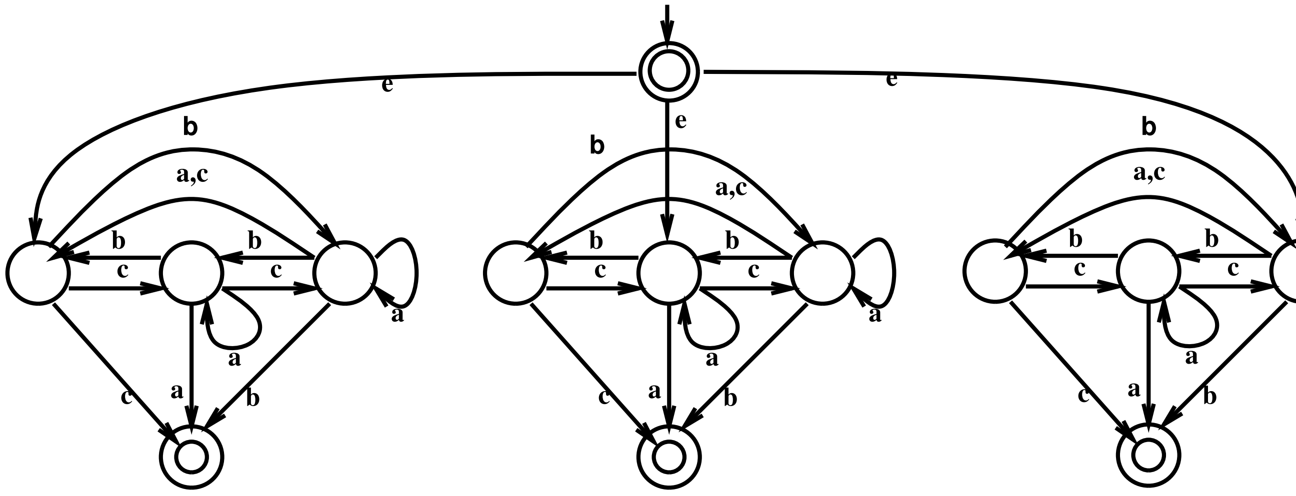




So, for instance, if we are in state "a' (i.e., we think that the remaining substring evaluates right-to-left to a) and we read a b, we move to state "c," because c is the only value x such that $a = b \cdot x$. In contrast, if we are in state "b" and we read an a, we can either stay in the same state or move to state "c," because both b and c are values y such that $b = a \cdot y$.

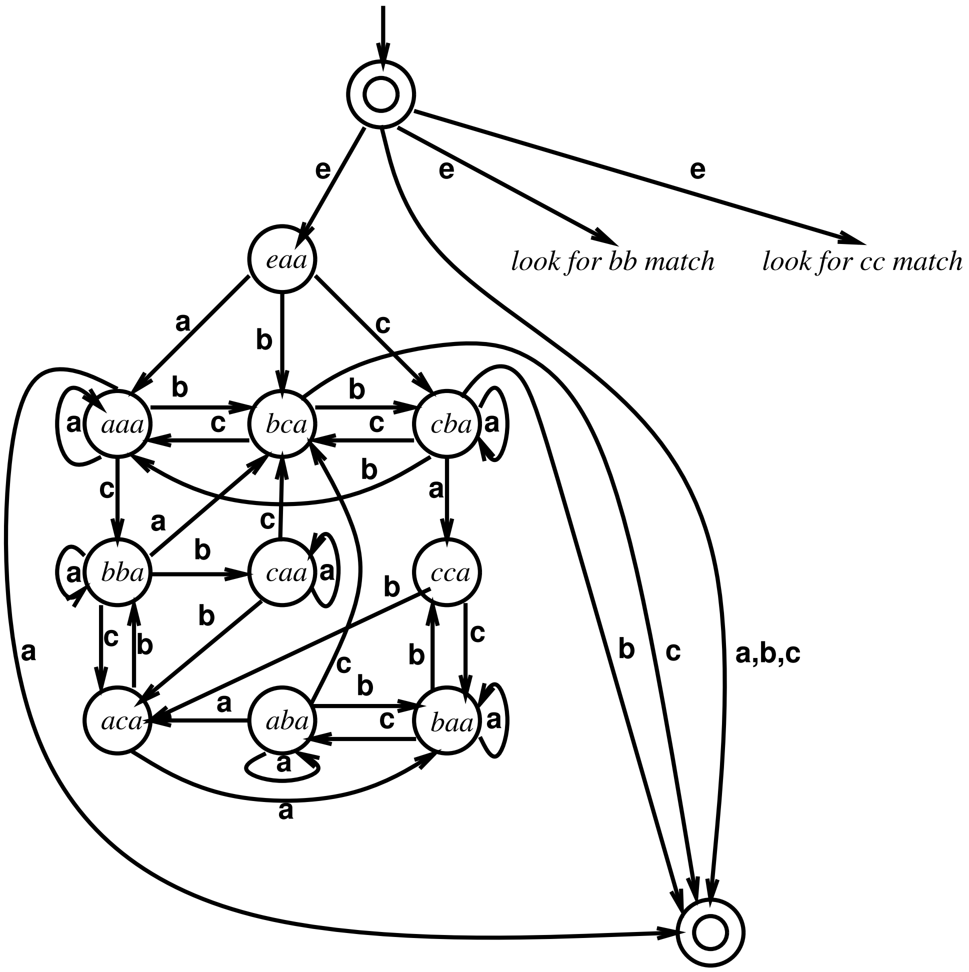
However, the behavior must change for the last symbol: at that point, there is nothing more to guess and we must just verify that the current guess can be accounted for by the last symbol. This adds another state: the NFA guesses that only one symbol is left and enters a final state, as depicted below; this time, we have added an initial state and ϵ transitions into one of the three guesses. The NFA must "remember" its first guess (the value of the whole expression)—we can do that by having a separate copy of the NFA for each possible initial guess, three in our case, so the complete NFA now has 13 states, as given below.





R−L value is c *R−L value is a* *R−L value is b*

The next challenge resides in doing both evaluations together, since we cannot re-read the input. Now that we know how to design a machine that will evaluate the string left-to-right and how to design another machine that will evaluate it right-to-left, how do we have them work together, at the same time, on the same input? The answer is to combine the two machines by designing a new machine whose states combine a state of the first machine with a state of the second. We do not need to combine the initial state and only 3 of the 9 possible combinations of final states are of interest (are final states of the new machines, corresponding to the same evaluation in both directions), but otherwise all combinations yield states, for a total of 32 states. The machine starts in an accepting state (for the null string); there is also a transition from the start state to a "final" accepting state under any single character input, in order to accept strings made of a single character. For all other strings, the machine now moves through ϵ transitions to one of three states, corresponding to the three guesses of the right-to-left NFA; denote these states (ϵ, a, a), (ϵ, b, b), and (ϵ, c, c), where the first part denotes the current value of the left-to-right evaluation, the second part denotes the current guess as to the right-to-left value of the remaining substring, and the third value denotes the original guess about the right-to-left value of the whole string. The first two entries exist to combine the two basic machines; the third entry serves to make three copies of the whole affair, to remember the initial guess. Thus, from state (ϵ, a, a), we can now move, on reading an a, to state (a, a, a); on reading a b, to state (b, c, a); and on reading a c, to state (c, b, a). From state (ϵ, b, b), on reading an a, we can move to one of two states, (a, b, b) or (a, c, b). The resulting machine is depicted on the next page—only one third is shown, for the guess that the string will evaluate to a in both directions; the other two thirds are the same, except for the transitions into the accepting states, which occur, for the first, from all states (x, y, b) such that $x \cdot y = b$ and under input y, and, for the second, from all states (v, w, c) such that $v \cdot w = c$ and under input w.



A DFA for this problem of course exists, but it is clearly a mess. A lower bound is somewhat hard to derive; all we can say easily is that any FA (deterministic or not) will need at least $3 \cdot 3 \cdot 3 = 27$ states: one factor of 3 is to keep information about the ongoing left-to-right evaluation, one is to keep information about the current right-to-left evaluation, and one is to remember the value guessed for the whole string. A DFA is likely to require many more states—although surely not the 2^{31} (!) that would be produced by a brute force mapping from our NFA.

Exercise 3.13

The simplest way to proceed is to use nondeterminism twice, first to get an equivalent machine with a single accepting state (a trivial exercise with nondeterminism—for instance, we can use ε transitions from the existing accepting states to a new state and make that new state the single accepting state) and secondly to set up a third equivalent machine that has a planar transition diagram. We do this second part by observing that, except for the number of times that a path can go through a loop, the number of distinct paths from any state of the NFA to its single accepting state is finite. We use this observation to create a new NFA that has all of the states of the old one and many more. Think of arraying all of the old states except the accepting state in a column on the left and the accepting state alone on the right; from each old state, we have a bundle of paths to the accepting state; ε transitions are used to chose a path in the bundle and additional states are used as necessary to implement each path—but all of these paths converge in a fan from the old states to the accepting state and thus do not cross. (Most of the states along these paths have only a few transitions, most often just one, defined and implicitly reject any other input, thereby drastically lowering the degree at each node.) From the accepting state, the old machine had some transitions back to other states; in our new machines, these transitions also exist and lead "around the back" and in the order in which old states were arrayed into the left column, so that they do not cross each other or any of the paths leading from the old states to the accepting state. The result is a planar, single-accepting state NFA that accepts exactly the same language as our original machine.

Exercise 3.14

From the hint, it suffices to show that there exists a language that is accepted by a DFA where every state has degree 6 and cannot be accepted by any DFA with at least one state of lower degree. Degree is the sum of in- and out-degrees, so it would be sufficient to demonstrate a DFA with in- and out-degrees of 3 at each state and prove that any DFA with a vertex of lower degree fails to accept that language. (Note that self-loops do not count, nor do two arcs between the same two states.) Degree 6 implies that we must have at least 6 alphabet symbols. Consider the symbols of the alphabet to denote the integer values 1, 2, 3, 4, 5, and 6 and let the language be the set of all strings over that alphabet such that the sum of their digits is one less than a multiple of 7. Note that, from any state of this machine, we must always have 6 distinct outgoing transitions— there cannot be a trap state and using the same transition for two distinct input symbols will irreparably damage our partial sum. Thus the degree of every vertex in the transition state diagram of any DFA for this problem is at least 6 and thus the graph is not planar.

Contrast such a DFA with an NFA for the same problem: the NFA may have vertices of low degree because it may have alternate paths between two states and thus also dead-ends; for instance, we could think of moving from state 3 to state 6 (the one accepting state) in one transition (which needs an input of 3), in two transitions (1 and 2, or 2 and 1, or 4 and 6, or 5 and 5, or 6 and 4), or in three transitions (many choices), using entirely distinct intermediate states (although each such state corresponds to a well-defined partial sum still) and forcing rejection if the second or third transition is not the expected one. The implicit rejection is then the key mechanism in lowering the degree of the nodes.

Exercise 3.15

1. We may want to write separate expressions for strings of length 0, 1, 2, 3, and for strings of length 4 or more: $\epsilon + 0 + 1 + 01 + 10 + 11 + 010 + 011 + 101 + 110 + 111$ will take care of all permissible strings of length less than 4; with length 4 or more we get $(0 + 1)^*(0101 + 0110 + 0111 + 1010 + 1011 + 1101 + 1110 + 1111)$; the sum of the two expressions does the trick, even if without any subtlety!
2. We can start by generating all strings that do not have more than two 0s in a row; this can be done in a number of ways, for instance by $(1 + 01 + 001)^*(\epsilon + 0 + 00)$, where the first part always generates something that ends in 1 (unless it is ϵ) and the last part simply allows a last few 0s. Once we have that, we can handle separately strings that may end with four arbitrary characters, remembering that, if the 6th and 5th characters from the end are both 0s, then the tail of four characters cannot start with a 0. The tail is not needed, so we write an ϵ alternative to it. We get $(1 + 01 + 001)^*((\epsilon + 0)(\epsilon + (0 + 1)^4) + 00(\epsilon + 1(0 + 1)^3))$.
3. We can divide the language into strings that do include the triple and strings that do not. The strings with the triple are of the form $x000y$ where x is either ϵ or a string with no triple of 0s that ends with a 1 and y is either ϵ or a string with no triple of 0s that starts with a 1. The key is thus to come up with a regular expression that describes all strings that do not include a triple of 0s, say expression R; then the language is simply $R + (\epsilon + R1)000(\epsilon + 1R)$. We derived an R in the previous problem, although we can be a bit more clever now, since our R was "oriented" (to end with 1) and could have been "oriented" the other way (to start with 1). Using this extra information, we can write $(1 + 01 + 001)^*(\epsilon + 0 + 00) + (1 + 01 + 001)^*000(1 + 10 + 100)^*$, which simplifies to $(1 + 01 + 001)^*(\epsilon + 0 + 00 + 000(1 + 10 + 100)^*)$.

4. We can write a regular expression for this language by brute force: a 0 can be followed by anything and a 1 can be followed by a 0 and anything. If a 1 is followed by another 1, then 0s cannot appear after that. Thus we can write $(0+10)^*1^*$
5. We assume that the string 000 is considered to contain two pairs of 0s. A simple way to do this problem is to decompose the language into pieces: (i) strings with no pair of consecutive 0s or 1s; (ii) strings with one pair of consecutive 0s and no consecutive 1s; (iii) strings with one pair of consecutive 1s and no consecutive 0s; and (iv) strings with one pair each of consecutive 0s and of consecutive 1s. Part (i) is described by the regular expression

$$(\epsilon+0)(10)^*(\epsilon+1)$$

part (ii) is described by the regular expression

$$((\epsilon+1)(01)^*)00((10)^*(\epsilon+1))$$

part (iii) is described by the regular expression

$$((\epsilon+0)(10)^*)11((01)^*(\epsilon+0))$$

and part (iv) is described by the regular expression

$$((\epsilon+1)(01)^*)00(10)^*11((01)^*(\epsilon+0))+((\epsilon+0)(10)^*)11(01)^*00((10)^*(\epsilon+1))$$

The union (sum) of these four expressions is the desired solution.
6. We begin by expressing the language of all strings with no pair of adjacent 0s, call it L_0, and the language of all strings with no pair of adjacent 1s, call it L_1. L_0 can be written as $(1+01)^*(\epsilon+0)$, while L_1 can be written as $(0+10)^*(\epsilon+1)$. Now, our desired language is just the concatenation L_1L_0, or

$$(0+10)^*(\epsilon+1)(1+01)^*(\epsilon+0)$$

Note that we can simplify this expression: the middle $(\epsilon+1)$ serves no purpose in the concatenation, since the 1 it introduces can also be introduced by the next expression; so we get

$$(0+10)^*(1+01)^*(\epsilon+0)$$

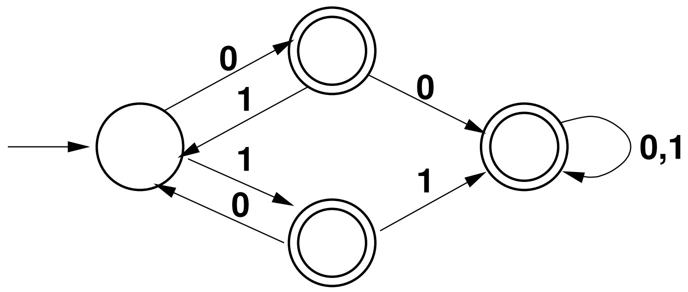
Exercise 3.16

1. $(P^*)^* = P^*$
   True. We prove it by induction. Our base case is to show that $P^*P^* = P^*$; from this base case, the conclusion follows by a trivial induction on the definition of Kleene closure: $(P^*)^* = \bigcup_{i \in \mathfrak{N}} (P^*)^i$ and our induction gives us that $(P^*)^i = P^*$. To prove the base case, we use a separate, nested induction: since $P^* = \bigcup_{i \in \mathfrak{N}} P^i$ and since $P^i P^j = P^{i+j}$, we get
   $$P^*P^* = \bigcup_{i \in \mathfrak{N}} P^i \bigcup_{j \in \mathfrak{N}} P^j = \bigcup_{i,j \in \mathfrak{N}} P^{i+j} = \bigcup_{i \in \mathfrak{N}} P^i = P^*$$
2. $(P + Q)^* = (P^*Q^*)^*$
   True. We prove it by showing inclusion in both directions. To show that $A = (P + Q)^*$ is contained in $B = (P^*Q^*)^*$, we prove by induction that $(P + Q)^i$ is contained in B for each i. Clearly $(P + Q)^0 = \epsilon$ and $(P + Q)$ are both contained in B. If $(P + Q)^i$ is contained in B, then $(P + Q)^{i+1} = (P + Q)^i(P + Q)$ contains concatenations of strings from $(P + Q)^i$, which is contained in B, with strings from $(P + Q)$, which is also contained in B; since B is a Kleene closure, it contains all concatenations of any of its strings with any other of its strings, and so contains $(P + Q)^i(P + Q)$. To show that B is contained in A, we use induction on the definition of Kleene closure and prove that $(P^*Q^*)^i$ is contained in A for each i. Clearly, $(P^*Q^*)^0 = \epsilon$ and (P^*Q^*) are contained in A (the latter would, in all formality, require another nested induction). If $(P^*Q^*)^i$ is contained in A, then $(P^*Q^*)^{i+1} = (P^*Q^*)^i(P^*Q^*)$ contains concatenations of strings from $(P^*Q^*)^i$, which is contained in A, with strings from (P^*Q^*), which is aslo contained in A; since A is a Kleene closure, it contains all concatenations of any of its strings with any other of its strings, and so contains $(P^*Q^*)^i(P^*Q^*)$.
3. $(P + Q)^* = P^* + Q^*$
   False. An easy counterexample is $(0 + 1)^*$, which is Σ^* on the alphabet $\Sigma = \{0, 1\}$; the regular expression $0^* + 1^*$ denotes only strings of 0s and strings of 1s, not strings containing both 0s and 1s.

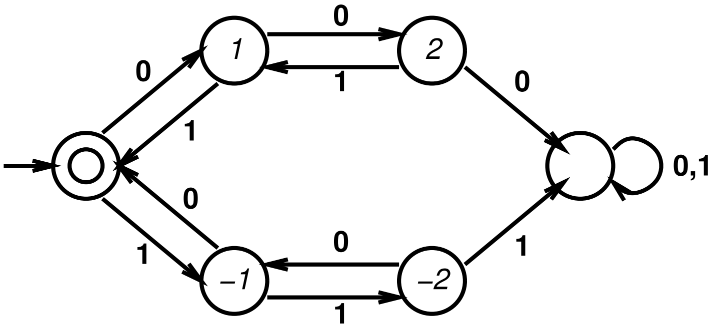
Exercise 3.17

1.
2. Not regular. Let n be the constant of the pumping lemma and pick the string $z = 0^n20^n$, which is clearly in the language. Write $z = uvw$; then v is a string of 0s (at least one and at most n) and lies entirely within the left half of the string. Pumping v just once yields the new string $uvvw = 0^{n+|v|}20^n$, which is not in the language.

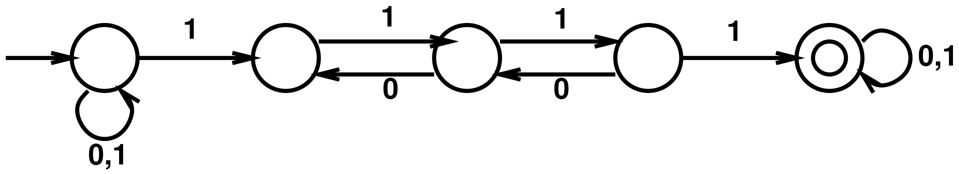
3. Not regular. The crucial point is that w cannot be the empty string (otherwise the language would be Σ^+, since we could always choose $w = \varepsilon$ and let y account for the entire string). Let n be the constant of the pumping lemma and pick the string $z = 10^{2n-1}110^{2n-1}10$, which is clearly in the language and which, moreover, has only one acceptable decomposition in the language, namely $z = (10^{2n-1}1)(10^{2n-1}1)^R0$. Use our extended pumping lemma to write $z = z_1z_2z_3$ with $z_1 = 1$, $z_2 = 0^n$, and $z_3 = 0^{n-1}110^{2n-1}10$. Now write $z_2 = uvw$; thus v is a string of 0s; pick an odd value for i, say $i = 3$; then the string $z_1uv^3wz_3$ should be in the language. But that string is $10^{2n+2|v|-1}110^{2n-1}10$, which is not in the language, because its first part, $10^{2n+2|v|-1}1$, is not reversible (because it has an odd number of 0s) and because its second part, $10^{2n-1}1$, no longer matches the first (because the first part now has more 0s).
4. Regular. This language is the complement of a regular language and regular languages are closed under complementation. More directly, the following four-state DFA recognizes it:



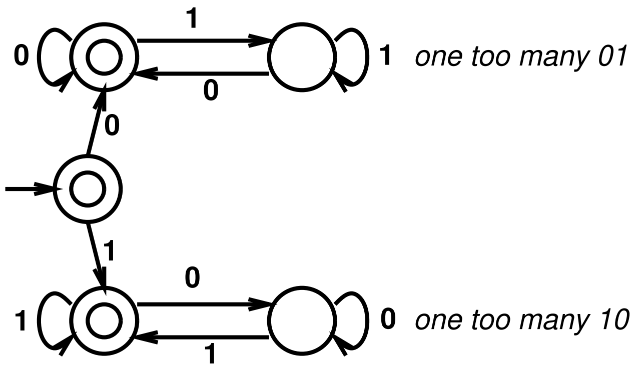
5. Regular. All we need to do is to keep track of the number of 0s and 1s as differences that can take the values 0, 1, -1, 2, -2, and larger. The following FA recognizes it:



6. Not regular. Let n be the constant of the pumping lemma and choose the string $z = 0^n 1^{2n} 0^n$. z is in the language, because the third piece is as long as the first. Now write $z = uvw$, with $|uv| \leq n$; it follows that uv is a string of 0s and thus also v. But then the pumped string $z' = uvvw = 0^{n+|v|} 1^{2n} 0^n$ is such that the first two pieces are each longer than the last and thus $z' \notin L$.
7. This language is something of a trick: it is simply 0^*, since, for every $n \in \mathfrak{N}$, there exists some $y \in \mathfrak{N}$ such that $y = n^2$. Thus the language is regular.
8. Regular. Being able to use just prefixes means that we need not worry about initial deficits in the number of 1s: we can choose a substring where the number of 1s always dominates the number of 0s and need only keep track of dominance by 1, 2, 3 or 4, which an FA can do. Thus the language is in fact regular: an NFA need only guess where the substring of interest starts (always at a 11 block), then move right on another 1, left on a 0, until it has moved three states to the right; if it ever tries to move back to the start state, the guess was bad. The resulting NFA is shown below.



A DFA with exactly the same states, but no loop on input of 1 on the start state and a transition back to the start state from the second state on input of 0 does the same job.
9. Regular. The key insight here is that we need only worry about transitions from 0 to 1 and from 1 to 0 and that these transitions must match. (Any string of 0s or any string of 1s by itself is entirely neutral, since neither one introduces a 01 or a 10.) We need not even count the transitions, since a transition from 1 to 0 can only be followed by one from 0 to 1 and vice versa. Thus we have a very simple machine or regular expression, which simply swallows consecutive 0s and consecutive 1s and only checks for transitions; a possible regular expression is $0^* + 1^* + (0^+1^+)^*0^+ + (1^+0^+)^*1^+$. Another way to view the problem is to realize that matching the number of transitions will cause the string to end with the same character with which it started, so we could also write $\varepsilon + 0(0+1)^*0 + 1(0+1)^*1$. The following DFA recognizes the language:



10. Not regular. Let n be the constant of the pumping lemma. Consider the string $z = 0^{n!+1}1^{n!}$. $n!$ and $n!+1$ are relatively prime (the only prime divisors of $n!$ are the primes between 2 and n and all leave a remainder of 1 when dividing $n!+1$), so z is in the language. By construction, the substring uv of the lemma is composed entirely of 0s, so the string $z(i) = uv^i w$ will have $n!+1+(i-1)|v|$ 0s. If $|v| < n$, we pump up once ($i = 2$) to obtain $z' = uvvw = 0^{n!+1+|v|}1^{n!}$; but $n!+1+|v| = n! + (|v|+1)$ is divisible by $|v|+1$, which is at least 2 and at most n and so also divides $n!$, so that the gcd is at least 2 and z' is not in the language. If $|v| = n$, we pump down once ($i = 0$) to obtain $z'' = uw = 0^{n!+1-|v|}1^{n!}$; but $n!+1-|v| = n! - (|v|-1)$ is divisible by $|v|-1 = n-1$, so that the gcd is at least $n-1$ and z'' is not in the language.

Exercise 3.18

The point of the problem is to use one or the other of our various closure properties, in order to avoid a difficult application of the pumping lemma (because the language is hard to characterize). Denote our language by L; by definition, L equals $\{0, 1\}^* - \{0^n 1^n \mid n \in \mathbb{N}\}$.

1. *Proof through Closure:* Regular languages are closed under complementation. Thus, if L' were regular, then so would be its complement; but its complement is the language $\{0^n 1^n \mid n \in \mathbb{N}\}$, well known to be nonregular. Hence L' itself cannot be regular. (Other closure-based proofs could be derived, but the one we just gave is clearly the simplest.)
2. *Proof with the Pumping Lemma:* In contrast with a proof based on closure, a proof through the pumping lemma is slightly tricky. Let n be the constant of the pumping lemma and let us pick $z = 0^n 1^{n!+n}$. Clearly, z is in L', since it does not have matching numbers of 0s and

1s. Write $z = uvw$; then v is a string of 0s, so that pumping v a total of i times yields the string $z_i = uv^{i+1}w = 0^{n+i|v|}1^{n!+n}$. Whatever the length of v is (some integer between 1 and n), we can always find an integer i such that $n + i|v| = n! + n$—just pick the value $i = n!/|v|$. The resulting string has matched numbers of 0s and 1s and thus does not belong to L', yielding the desired contradiction and proving that L' is not regular.

Exercise 3.19

1. This language is not regular. Let n be the constant of the pumping lemma and choose the string z to be

$$z = \binom{0}{0}^n \binom{1}{1} \binom{0}{0}^n$$

   which is clearly in the language. Write $z = uvw$; since $|uv| \leq n$, uv is a string of $\binom{0}{0}$ before the marker $\binom{1}{1}$; thus pumping it up once yields the new string

$$z' = uvvw = \binom{0}{0}^{n+|v|} \binom{1}{1} \binom{0}{0}^n$$

   But then the top and bottom rows are both the string $0^{n+|v|}10^n$, which is not its own reverse; hence z' is not in the language.
2. This language is regular: basically, we can only use two of the four symbols of the alphabet, namely $\binom{0}{1}$ and $\binom{1}{0}$. Thus the language has regular expression

$$\left(\binom{0}{1} + \binom{1}{0}\right)^*$$

3. This language is not regular. Let n be the constant of the pumping lemma and pick the string $z = \binom{1}{0}^n \binom{0}{1}^n$, which is clearly in the language. Now write $z = uvw$; v is a string of $\binom{1}{0}$, so that pumping it up just once to $uvvw$ creates more 1s on top than on bottom, so that $uvvw$ is not in the language.

Exercise 3.20

1. This language is regular. In processing the symbols, the FA need only store in its finite state whether or not the next symbol encountered should generate or propagate a carry. Since we read the input in

the "wrong" direction (from the most significant bit to the least significant), what really happens is that we detect the use of a carry before we get to the point where it is generated—in effect, our FA will check that every carry used is also generated. The FA has just two states; the start state is accepting, the other rejecting. We stay in the start state when the next vector read involves no carry generation, propagation, or use, for triples $\begin{pmatrix}0\\0\\0\end{pmatrix}$, $\begin{pmatrix}0\\1\\1\end{pmatrix}$, and $\begin{pmatrix}1\\0\\1\end{pmatrix}$; we move to the other state when a carry is used and not propagated, for triple $\begin{pmatrix}0\\0\\1\end{pmatrix}$. Once in the other state, we stay there when a carry is propagated, for triples $\begin{pmatrix}0\\1\\0\end{pmatrix}$, $\begin{pmatrix}1\\0\\0\end{pmatrix}$, and $\begin{pmatrix}1\\1\\1\end{pmatrix}$, but return to the start state when a carry is generated, for triple $\begin{pmatrix}1\\1\\0\end{pmatrix}$.

2. This language is not regular. Basically, this is due to the fact that the number of bits needed to express the product of x and y can grows as fast as (although no faster than) the sum of the number of bits needed to express each of x and y. We can set up a product where xy takes $|x| + |y|$ bits and where we only pump in the area where x and y have leading 0s, thereby increasing the number of bits used in xy without increasing the number of bits used in x or y—or, viewed differently, thereby altering the product without altering the terms.
   Let n be the constant of the pumping lemma. We choose the following string:

$$z = \begin{pmatrix}0\\0\\1\end{pmatrix}\begin{pmatrix}0\\0\\0\end{pmatrix}^n\begin{pmatrix}1\\1\\0\end{pmatrix}\begin{pmatrix}0\\0\\0\end{pmatrix}^{n+1}$$

   which corresponds to the product $2^{n+1} \cdot 2^{n+1} = 2^{2n+2}$ and is sufficiently long. We use the extended pumping lemma and write $z = xuvwy$, where $x = \begin{pmatrix}0\\0\\1\end{pmatrix}$, $y = \begin{pmatrix}1\\1\\0\end{pmatrix}\begin{pmatrix}0\\0\\0\end{pmatrix}^{n+1}$, and $uvw = \begin{pmatrix}0\\0\\0\end{pmatrix}^n$, so that v is a string of one or more triples of 0s. Pumping up or down once does not affect the two product terms, but doubles or halves the result, which is then incorrect for the multiplicands given, so that the new string is not in the language.

Exercise 3.21

This language is regular. It suffices to observe that the strings of the language are all of the form M^*w, where w denotes a number between 0 and 999; the set of such numbers is finite and thus regular, say with regular expression E. Then the regular expression for the entire language is M^*E.

Exercise 3.22

In spite of its rating, this is an easy exercise. The main problem is to define what is a legal expression in that context. Since we have banished parentheses and unary operators, we are left with two binary operators. Precedence and order of evaluation do not affect the legality of an expression, although they certainly affect the language the expression represents. Thus it is enough to ensure that each operator has two operands, which simply means that we have a three-state DFA: it starts in a rejecting state (the expression cannot be empty) q_0 and has a rejecting trap q_r and an accepting state q_F. From the start state, the machine moves to the trap on reading an operator, but to the accepting state on reading 0 or 1. From the accepting state, the machine moves back to the start state on reading an operator; it stays in the accepting state on reading a 0 or 1. In the trap, of course, the machine stays trapped for any input.

Exercise 3.23

Intuitively, verifying membership in that language appears to require that we check all $n+1$ decompositions of the input string z into substrings x and y with $z = xy$ and evaluate each substring according to the procedure described in Exercise 3.12 before testing two results for equality. The equality testing is not really a problem, since each expression must assume a value from the set $\{a, b, c\}$—we can just maintain in the finite-state control the values of the two expressions by turning the states into triples, with the first part of the triple corresponding to a state of the machine described in Exercise 3.12 and the next two parts corresponding to the two values. We can use nondeterminism to guess the correct breakdown of z into x and y, proceeding on x just as in Exercise 3.12, storing the computed value into the finite-state control, then proceeding on y again as in Exercise 3.12 (but now with states that store the value computer for x), and accepting if the value computed for y matches that computed for x.

Exercise 3.24

Our proof has two parts; one part is easy, while the converse if a bit harder.

- We prove that, if L is unitary, then, whenever we have $u, uv, w \in L$, then we must also have $wv \in L$. Because L is unitary, it is accepted by a DFA with a single accepting state; let this machine be M and its accepting state be q_F. In accepting uv, machine M must first process u; since u is in the language and since M is deterministic, M ends up in state q_F after processing u. But it also ends up in state q_F after processing uv, so that there is a loop from q_F back to q_F that is followed under input v. Consider then the processing of string wv; since M accepts w, it finds itself in state q_F after processing w and then follows the loop from q_F back to q_F under input v; hence it ends in the one accepting state, so that it accepts wv, so that wv is in L.
- Now we need to prove the converse: namely, that, given some regular language L, if having $u, uv, w \in L$ always implies $wv \in L$, then L must be unitary. We can thus start with some DFA for L and must proceed to show how to build some equivalent DFA that has a single accepting state. Let M be some DFA for L and let us assume, without loss of generality, that every state of M is accessible—that is, that, for each state q of M, there exists some string x_q that causes M to move from its start state to q. (If M had unreachable states, we could simply eliminate them to obtain the required machine; detecting unreachable states is exactly the same problem as graph reachability and is easily solved in linear time through graph traversal.) If M has a single accepting state, we are done. Otherwise we construct a new DFA M' that accepts L and has strictly fewer accepting states than M.

  If M has more than one accepting state and none of them is reachable from any other (including itself), then L is prefix-free and we can replace the accepting states of M by a single accepting state (any transition in M leading to one of its accepting states is replaced by a transition that leads to the new single accepting state in M'), from which any transition leads to a rejecting trap. We have the desired construction.

  If M has accepting states q' and q'' such that q'' is reachable from q', let v be a string that leads M from q' to q'' and let u be a string that leads M from its start state to q'. Note that M must accept, not just u and uv, but also uv^i for any $i \in \mathbb{N}$, because we can set $w = uv$ in the statement of the hypothesis and thus obtain that $wv = uvv$ is also in L—and then repeat with one more suffix of v, etc. Similarly, if w is a string that leads M from its start state to q'', then M must accept wv^i for any $i \in \mathbb{N}$. Moreover, if M accepts some string wt (with $t \neq v$)

by reaching q'' and then some third accepting state q''', then it must also accept string ut (because it accepts w, wt, and u). Thus we can eliminate state q'' by producing a new DFA in which each transition into q'' in the original automaton is replaced by a transition into q'. From our arguments above the new machine accepts exactly what the old machine did and has one fewer accepting state. Our inductive step is thus complete.

Exercise 3.25

1. False. Any language that includes as strings every character of Σ will, when we take its Kleene closure, generate Σ^*. Thus it is enough to identify a nonregular language that includes every symbol of the alphabet. One example is the complement of our standard nonregular language $\{0^i 1^i \mid i \in \mathbb{N}\}$ (over the alphabet $\Sigma = \{0, 1\}$); another over the same alphabet would be $\{0^i 10^i \mid i \in \mathbb{N}\} \cup \{1^i 01^i \mid i \in \mathbb{N}\}$. Both languages contain as strings both 0 and 1 and thus their Kleene closure is $\{0, 1\}^*$, which is of course regular.
2. False. The problems is that, given L and L_2, there typically exist many choices of language L' such that $L = L'L_2$; if L is regular and L_2 is finite, at least one of these choices *must* be regular, but not all of them need be.
   To get a regular L', we can run the machine for L until the input runs out, then start guessing a suffix and run it both through M—continuing the computation—and through a machine for L_2—starting from the start state. If both machines accept together at any stage, we accept the input string. Since we cannot really run a machine until the input runs out, we use ε transitions instead. Given a DFA M for L, we can build an NFA M' for L_1 as follows: M' will have a copy of M (with all states made non-accepting) and an ε transition from each state q of M to the start of a new submachine, call it M'_q; this new submachine makes *only* ε transitions and combines M (started in state q) and the machine for L_2, started in its start state, running them effectively in parallel. Each M'_q can only make a finite number of ε transitions because the language L_2 is finite; any state of M'_q in which both the copy of M and the copy of the machine for L_2 accept is an accepting state of M'. Thus, in order to accept an input string, M' must make use of one of its ε transitions (otherwise it stays within its copy of M, which has no accepting state), but it can only make such a transition when it has run out of input, because it will enter a submachine that will not process any input on the way to any accepting state—any futher input will cause rejection. Moreover, from this point, both the

"continuing copy" of M and the machine for L_2 must run together and accept at the same time, i.e., M' must guess a suffix for its input that belongs to L_2 and that, when concatenated to its input, belongs to L.

As a counterexample (on a one-symbol alphabet!), let $L_1 = \{0^k \mid k > 2$ and not a prime$\}$ and let $L_2 = \{\varepsilon, 0\}$. Thus L_1 is not regular and L_2 is finite. Note that $L = L_1 L_2$ now consists of all strings of 0s whose length is either a non-prime or a non-prime plus 1. But that is *all* strings of 0s (of at least 4 characters), since any prime beyond 3 is just some non-prime plus 1. Thus $L = \{0^k \mid k > 3\}$ is regular and we have a counterexample. But note that we could have chosen $L' = L$ and still have had $L = L'L_2$.

3. True. If the union were disjoint, there would be no problem at all: if L and L_2 are regular, then so is $L - L_2$ (we can imagine running two machines in parallel and accepting only if the first machine accepts and the second rejects). We do not know the extent of the intersection of L_2 with L_1, but L_2 is finite and any of its subsets is also finite and thus regular. For each subset L_2' of L_2, the language $L - L_2'$ is thus regular; our language L_1 is one of these differences and thus is regular. (Note that we made use of the finiteness of L_2 when concluding that any subset of L_2 must be regular.)
4. False. A simple example has $L = L_2 = \{0, 1\}^*$ and L_1 be the language of all strings over $\{0, 1\}$, the length of which is a perfect square (our language L_2 on page. 73). The language L_1 is not regular, yet L and L_2 are clearly regular. That the quotient works as advertised is easily seen: for any string, there is always another string such that the sum of the lengths of the two strings is a perfect square—and our choice on either side ranges over all possible integers and any possible choice of characters within that length.

Exercise 3.26

The language is regular. Given a DFA $M = (\Sigma, Q, S, F, \delta)$ for L, we construct an NFA $M' = (\Sigma, Q', S', F', \delta')$ for SUB(L) as follows. The idea is to allow M' to make any transition that M can make (corresponding to reading a character of the subsequence) and also to allow it to make an ε-transition instead—to the same state as the real transition in M. Thus we set $Q' = Q$, $S' = S$, and $F' = F$; if we have $\delta(q_i, a) = q_j$ in machine M, then we set $\delta'(q_i, a) = q_j$ and $q_j \in \delta'(q_i, \varepsilon)$. Our new machine accepts a string through a series of transitions that are a combination of ε-transitions and transitions of M; each ε-transition amounts to guessing that the complete string would have some character at that place that is not present in the subsequence and guessing which character that is (by moving to the proper guessed next state).

Exercise 3.27

The language is regular. Given a DFA $M = (\Sigma, Q, S, F, \delta)$ for L, we construct an NFA $M' = (\Sigma, Q', S', F', \delta')$ for L' as follows. The idea is to have M', on input $W = xy$ guess the decomposition of W (i.e., guess where x stops and y starts) and also guess which state M would be in after reading y; then we simulate M on x; when we guess the end of x has come, M must be in a final state, and we now start M from the its start state on y, making sure that by the end it reaches the state which we had initially guessed. We set $Q' = S' \cup Q \times Q \times Q$; other than the new start state, every state in the new machine is a triple, (q_i, q_j, q_k), where q_j records the initial guess of the state M would be in after reading the second part of w, q_j is the state of M when reading the second part (which comes first for M'), and q_k is the state of M when reading the first part (which comes second for M'). Thus the final states are of the form (q_f, q_j, q_j) for $q_f \in F$ and $q_j \in Q$. Our initial transitions are

$$\delta'(S', \epsilon) = \{(q_i, q_i, S) \mid \forall q_i \in Q\}$$

Then we have transitions to mimic M on the first part (x): if $\delta(q_i, a) = q_l$, then

$$(q_l, q_j, S) \in \delta'((q_i, q_j, S), a)$$

and transitions to mimic M on the second part (y): if $\delta(q_i, a) = q_l$, then

$$(q_f, q_j, q_l) \in \delta'((q_f, q_j, q_i), a)$$

where $q_f \in F$. The only problem with this construction is that we can cycle through it more than once: if we return to the start state in the second part (on y), instead of using transitions for the second part, we could instead choose to use a transition for the first part, which would cause errors. We can take care of that detail by using S' rather than S in transitions for the first part, then adding transitions

$$(q_f, q_j, S) \in \delta'((q_f, q_j, S'), \epsilon)$$

(where $q_f \in F$), which guess when x ends and switch from the first to the second part by simply renaming S' to S.

Exercise 3.28

Since L is regular, it is accepted by some deterministic FA with fully specified transition, say $M = (\Sigma, Q, S, F, \delta)$. Given M, we construct a new FA M' that accepts exactly $NP(L)$. M' has the same alphabet, same same start state, and same accepting states as M; its state set is $Q \cup \{T\}$; and its transition function δ' is constructed from that of M as follows:

$$\delta'(q, a) = \begin{cases} \delta(q, a) & \text{if } q \in Q - F \\ T & \text{if } q \in F \text{ or } q = T \end{cases}$$

That is, M' behaves exactly like M on any string until it first reaches an accepting state of M; any further input at that point moves M' to the rejecting trap state T. Thus any string accepted by M' is also accepted by M; moreover, any string x accepted by M' cannot be the prefix of any other string accepted by M', since any input beyond the last character of x will cause M' to enter the rejecting trap. Note that M' as constructed may have many inaccessible states, including inaccessible accepting states.

Exercise 3.29

The language is regular. Let M be a DFA for L, with starting state q_0, final state set F, and transition function δ. We construct a new machine, M', which recognizes our language. M' basically runs one copy of M forward on x, the other (nondeterministically) backwards on x; if the two copies end in the same state, then the string is accepted.

Formally, we can write a state of M' as a pair $[q_i, q_j]$, where q_i is the state of the copy of M that processes x forwards and q_j is the state of the copy of M that processes x backwards. In addition, M' has an initial state q' from which there are ϵ transitions to each state of the form $[q_0, q_f]$ for $q_f \in F$. The final states of M' are all states of the form $[q_i, q_i]$. If M has transition $\delta(q_i, a) = q'_i$, then M' can move from state $[q_i, q_j]$ to any state $[q'_i, q'_j]$ such that M has a transition $\delta(q'_j, a) = q_j$.

Exercise 3.30

This language need not be regular. The problem is that this operation enables us to match exactly in length two otherwise independent substrings, something that an FA normally cannot do. Consider the regular language $L = \{0^i 1 2^j \mid i, j \geq 1\}$ and the regular language $L_1 = \{0^i 2^j \mid i, j \geq 1\}$. Now, if $FL(L)$ is regular, the new language $L' = FL(L) \cap L_1$ must also be regular, since regular languages are closed under intersection. But it is easy to see that L' is our old non-regular acquaintance $L' = \{0^i 2^i \mid i \geq 1\}$. (By intersecting with L_1, we retain only strings from $FL(L)$ that have no 1 in them; since every string in $FL(L)$ begins with a 0 and ends with a 2, the first third selected must contain only 0s and the last third selected must contain only 2s, and the two must have the same length.) So, even though our chosen L is regular, $FL(L)$ is not.

Exercise 3.31

There is no particular difficulty in solving this problem other perhaps than notation. The solution follows the approach given in the text: we build a machine with states denoted by tuples that encode information about various guesses and states of simulated runs of the original machine. If we want to accept the ith out of k pieces of equal length from strings of L (the original language), we shall have to reconstitute (guess) the first $i-1$ pieces as well as the last $k-i$ pieces and process them concurrently—even though the machine for L would have processed them and the piece given in the input in serial order. Thus we need to guess, not just the $k-1$ missing pieces, but also the state in which the machine for L would have found itself as it started processing each piece. We then need to verify that the pieces can be "stitched together" properly, i.e., that we can end the processing of each piece (guessed or not) in the state used to start the processing of the next piece. Thus, given a DFA M for the original language L, we construct an NFA M' for our language as follows: except for a new start state, each state of M' will be represented as a $(2k-1)$-tuple, encoding k current states (one for each piece) and $k-1$ guessed initial processing states (one for every piece except the first, since that piece is processed from the start state); denote such a state

$$(q_1, g_2, q_2, g_3, q_3, \ldots, g_k, q_k)$$

where q_i and g_j are states of M and q_i denotes the current state in the processing of the i piece and g_i denotes the guessed initial processing state for the ith piece. The new start state, call it S', has ε-transitions to all states of the form

$$(S, g_2, g_2, g_3, g_3, \ldots, g_k, g_k)$$

where S is the start state of M; and accepting states are any states of the form

$$(g_2, g_2, g_3, g_3, g_4, \ldots, g_k, f)$$

where f is any accepting state of M. Under input a, if machine M in state q_i has transition $\delta(q_i, a) = q_i'$, then our new machine can make a transition from state

$$(q_1, g_2, q_2, , \ldots , g_i, q_i, \ldots , g_k, q_k)$$

to any state of the form

$$(q_1', g_2, q_2', , \ldots , g_i, q_i', \ldots , g_k, q_k')$$

where for each state q_j $(j \neq i)$ in the tuple, there exists some alphabet character c_j such that $\delta(q_j, c_j) = q_j'$.

Thus machine M' guesses all missing $k-1$ pieces (all of length equal to the input piece), guesses the state which M would have reached when beginning to process the jth piece for each j, and verifies at the end that its guesses are all consistent—that the k pieces fit together and that each simulation of M ends in the same state that was guessed as the starting point for the simulation of the next piece. The checking prevents M' from accepting strings not in our target language; the guessing allows it to accept any string in our target language.

Exercise 3.32

The trick here is mostly in figuring out a way to pace the submachine that will handle the substring z, since that substring is half the length of the input. The submachine for z must make only one transition for every two transitions made by the submachine for substring x, which dictates the pace (since it is fed the real input). One solution is to use a toggle: on even positions in x, the submachine for z makes a transition, but on odd ones it does not. Otherwise, the machine is constructed exactly along the lines discussed in class for this class of languages: a state of the NFA we are building will be a 7-tuple, augmented into an 8-tuple by the odd/even toggle. The 7-tuple members include 4 current states for the 4 submachines handling the 4 substrings and 3 markers that store the initial guesses for the starting positions of the 2nd, 3rd, and 4th submachines. (Note that the 1st and 3rd submachines must be kept separate: they have exactly the same transition function, but need not start in the same state and thus need separate current states.)

So, given a DFA $M = (\Sigma, Q, S, F, \delta)$ for L, we construct an NFA $M' = (\Sigma, Q', S', F', \delta')$ for L' as follows.

$Q' = \{S'\} \cup \{(q_i, q_j, q_k, q_l, q_m, q_n, q_o, t) \mid q_i, q_j, q_k, q_l, q_m, q_n, q_o \in$
Q and $t \in \{\text{odd}, \text{even}\}\}$

$\delta'(S', \varepsilon) = \{(S, q_i, q_i, q_j, q_j, q_k, q_k, \text{even}) \mid q_i, q_j, q_k \in Q\}$

$$\delta'((q_i, q_j, q_k, q_l, q_m, q_n, q_o, \text{even}), a) = \{(q_i', q_j, q_k', q_l, q_m', q_n, q_o, \text{odd}) \mid \delta(q_i, a) = q_i',\ \delta(q_m, a) = q_m',\ \exists\alpha, \beta,\ \delta(\delta(q_k, \alpha), \beta) = q_k'\}$$

$$\delta'((q_i, q_j, q_k, q_l, q_m, q_n, q_o, \text{odd}), a) = \{(q_i', q_j, q_k', q_l, q_m', q_n, q_o', \text{even}) \mid$$

$$\delta(q_i, a) = q_i',\ \delta(q_m, a) = q_m',\ \exists\alpha, \beta,\ \delta(\delta(q_k, \alpha), \beta) = q_k',\ \exists\gamma,\ \delta(q_o, \gamma) = q_o'\}$$

$$F' = \{(q_i, q_i, q_j, q_j, q_k, q_k, q, \text{even}) \mid q_i, q_j, q_k \in Q,\ q \in F\}$$

In order to accept, we must be in an "even" state, i.e., we must have seen an even number of input characters; the transition function keeps toggling between odd and even state labels and the submachine for z only makes transitions when we read an even-numbered input character.

Exercise 3.33

Let L be our language and SUB(L) the set of subsequences of strings in L; denote by L' the complement of SUB(L). Note that, by definition of minimal elements, no minimal element may be a subsequence of any other minimal element. We want to show that the set of minimal elements for a language is always finite.

Since this is the hard part of the proof, let us start with the relatively simple case where the alphabet has two symbols (the case of a one-symbol alphabet is of course trivial). Let the alphabet be $\Sigma = \{0, 1\}$. Let x be any string in Σ^* of length $n = |x|$. It is easily seen that x must be a subsequence of $(01)^n$ (or $(10)^n$), since the latter has n copies of each alphabet symbol. Let y be a string over Σ^* such that x is not a subsequence of y; then it follows immediately that neither $(01)^n$ nor $(10)^n$ can be a subsequence of y (because the relation "is a subsequence of" is transitive). Now define the family of languages $\{L_i\}$, $i \in \mathbb{N}$, with $L_{2n} = (0^*1^*)^n$ and $L_{2n+1} = L_{2n}0^*$. Thus L_i is defined on the basis of the largest possible number of alternations between 0 and 1 in any string ($i - 1$ as per our definition). Note that any string of length n must belong to L_{2n}—again, we can choose any symbol for each position.

We claim that, if z belongs to L_k for some k and S is a subset of L_k in which any two strings are incomparable, then S contains only a finite number of strings y such that z is not a subsequence of y. We prove our claim by induction on k. The basis ($k = 0$) is trivially true. Assume that the result holds for all strings z in L_i, $0 \le i < k$, and consider a string z in L_k; in particular, z has at most $k - 1$ alternations. Note that our inductive hypothesis implies that S contains only a finite number of strings in L_{k-1} (as can be seen by choosing $z = \varepsilon$). Thus we need only prove that S contains finitely many strings y with exactly $k-1$ alternations such that z (a string with at most $k-1$ alternations) is not a subsequence of y. But S cannot contain more than a finite number of strings y with exactly $k-1$ alternations in any case—regardless of z: in fact, it cannot contain more than k such strings, one for each run in the series of alternations (you can imagine each such string as having $k-1$ short runs and one long run; each has its long run in a different place in the series of alternations, so that none can be a subsequence of the other; but any additional string with $k-1$ alternations is either a subsequence or a supersequence of one of the existing k strings.)

Now we are almost done, at least for languages over a two-symbol alphabet: we simply need to conclude that the set of minimal elements for any language over the alphabet $\Sigma = \{0, 1\}$ is finite. Our previous result indicates that any set of S of minimal elements can only have a finite number of members with at most k alternations, for any value of k (because we can pick z to be the empty string). Moreover, our earlier results indicate that any string with at least $2n$ alternations includes as a subsequence any string of length n. Thus suppose that our set S includes at least one string of length n; then it cannot have any strings in it with more than $2n-1$ alternations, since such strings would not be minimal. Hence the maximum number of alternations is bounded by some (unknown) constant, which is at most twice the length of the shortest string in S; such being the case, the number of strings in S must be finite.

Now we need to extend this result to alphabets of more than two elements. The details are simple and mechanical (we can base our reasoning on alternations again, but this time have to consider a larger number of alternations), but rather tedious, so we omit them here. With this result in mind we have proved that the set of minimal elements for L', the complement of SUB(L), is finite. Using this result, we can easily recognize L' with an NFA: the NFA will guess a subsequence of the input string and verify that it is one of the finitely many minimal elements of L'—because a string belongs to L' if and only if it contains a minimal element as a subsequence. (The "only if" part is trivial. The "if" part follows from the definition of SUB(L): if the string were in fact in SUB(L), then all of its subsequences would be too, yet one of these subsequences is a minimal element of L', a contradiction.) Since the complement of SUB(L) is regular, so is SUB(L): we have thus obtained the desired result.

Solution Manual
Chapter 3, Additional Problems

Additional Problem 1

The key observation (a very simple one) is that, if x is a suitable string—let's call it a *distinguishing* string, then, for every prefix y of x, we must have $\delta(p, y) \neq \delta(q, y)$. So, until the end of the string is reached, the two sequences of moves go through pairs of different states; moreover, both states in each pair are accepting or both are rejecting—otherwise we would have found a shorter distinguishing string. There are $(n - |F|)(n - |F| - 1)$ pairs of different rejecting states and $|F|(|F| - 1)$ pairs of different accepting states, or $f(n, F) = n^2 - n + 2|F|(|F| - n)$ pairs in all. The shortest distinguishing string never goes through the same pair twice—if it did, that loop could simply be cut out. So the shortest distinguishing string takes the machine through at most $f(n, F)$ transitions and thus has at most $f(n, F)$ symbols. This is not very long: even when $|F| = 1$, the value is at most $(n-1)(n-2)$.

On the other hand, it is unlikely that this bound is tight, because we have approached the two copies of the machine as if they were completely independent; since they are, in fact, copies of each other, certain transitions may not be possible in the paired machine. For instance, if we are in combined state (q_i, q_j) and, under input a, have a transition to combined state (q_j, q_k), then, from the latter state, under input a, the transition must be to a combined state of the form $(q_k, \ )$—not just to any combined state.

Can we then do better? That is, can we derive a tighter bound through a different approach? Suppose we want to divide all of the n states of the DFA into groups, such that states within any group are indistinguishable from each other. We can do this constructively as follows: begin with two groups, one being F, the other $Q - F$—any accepting state is immediately distinguishable from any rejecting state. At any step of the process, we will have a certain number of groups, call them $G_1, G_2, \ldots$. Now look at the states reachable in one transition on symbol a from any of the states within a group G_i; if all of these states are in the same group, then the states in group G_i remain indistinguishable under additional symbol a; if that is the case for every input symbol a, then the group of states in G_i remains indistinguishable under any input sequence longer by one symbol and thus G_i remains unchanged to the next step. On the other hand, if there is some input symbol a such that the successors of some states in G_i are in some group G_j and the successors of some other states in G_i are in some other group G_k ($j \neq k$), then the states in the first group and the states in the second group can be distinguished and thus group G_i gets split into two or more groups for the next step. If all groups remain unchanged from one step to the next, the final answer has been found: no more distinguishing of any type can take place. Thus the process stops as soon as two consecutive steps return the same groups; viewed the other way, the process continues as long as at least one group is split, thereby increasing the number of groups. But the number of groups cannot exceed n, the number of states of the machine—all states are then distinguishable—and there were two groups initially, so that the number of steps is at most $n - 2$.

We claim that this process creates distinguishing sequences automatically, so that no distinguishing sequence need be longer than $n - 2$ symbols. If we have two distinguishable states p and q, we find in the process the first step, say i, at which the two are no longer in the same group; then a sequence of $i - 1$ symbols will serve as a distinguishing sequence and can be reconstructed by following the sequence of groups splits that led to the last group in which p and q remain together, followed by the symbol that led to the split which sundered p from q.

Additional Problem 2

Basically, we want to run the two machines in parallel on a variety of input strings, checking whether or not the two accept the same strings. The key question is how long the strings need to be before we are convinced that the two machines accept the same language. Since running the two machines together can be simulated by running a "product" machine, the states of which correspond to ordered pairs of states from the two machines, we need only ask how many states the product machine has. Given a DFA of m states and one of n states, the product machine has mn states, so that the longest string that we need to run through it need not exceed $mn - 1$ symbols (after that, we must have a loop). Now each different string may cause the machine to follow a different path among its states; we need to explore as many paths as we can in order to reach every paired state that can be reached, so we must be prepared to test every string of length up to $mn - 1$, or order $|\Sigma|^{mn}$ strings.

An important problem that would arise in practice in such a test is whether, after running a string through a DFA, there is any way to force the DFA back to its start state. In our answer above, we have assumed such to be the case—automagically (we have some sort of reset button). In practice while testing a sequential machine (such as a CPU), no such assumption can be made (even if there is a reset line, it too must be tested). What one then needs is an input sequence that will take the machine (if it functions properly) from any state back to the start state in some guaranteed number of moves. Such sequences are known as homing or synchronizing sequences, but not every DFA has one.