Bismillah hir Rehman nir Raheem ----------^^^^^^^^^^----------Assalat o Wasalam o Allika Ya RasoolALLAH

Solution Manual to Introduction to Languages and the Theory of Computation

(3rd Ed)

John C. Martin

Full book

Published By: Muhammad Hassan Riaz Yousufi

To Read Online & Download: WWW.ISSUU.COM/SHEIKHUHASSAN

Chapter 1

Basic Mathematical Objects

1.1. (a) $\{(-1)^n n \mid n \in \mathcal{N}\}$

(b) The denominator of an element can be any positive power of 2, and the numerator can be any positive integer less than the denominator. One formula is therefore

$$\{\frac{m}{2^j} \mid j > 0 \text{ and } 0 < m < 2^j\}$$

(c) $\{1^n 0^n \mid n > 0\}$
(d) $\{\{n\} \mid n \in \mathcal{N}\}$
(e) $\{\{j \in \mathcal{N} \mid 0 \leq j \leq n\} \mid n \in \mathcal{N}\}$
(f) $\{\{j \in \mathcal{N} \mid 0 \leq j < 2^n\} \mid n \in \mathcal{N}\}$

1.4. (a)

$$\begin{aligned} A - (A - B) &= A \cap (A \cap B')' \\ &= A \cap (A' \cup B) \\ &= (A \cap A') \cup (A \cap B) \\ &= \emptyset \cup (A \cap B) \\ &= A \cap B \end{aligned}$$

(b)

$$\begin{aligned} A - (A \cap B) &= A \cap (A \cap B)' \\ &= A \cap (A' \cup B') \\ &= (A \cap A') \cup (A \cap B') \\ &= \emptyset \cup (A \cap B') \\ &= A \cap B' \\ &= A - B \end{aligned}$$

(c)

$$\begin{aligned} (A \cup B) - A &= (A \cup B) \cap A' \\ &= (A \cap A') \cup (B \cap A') \\ &= \emptyset \cup (B \cap A') \\ &= B \cap A' = B - A \end{aligned}$$

(d)

$$\begin{aligned} (A - B) \cup (B - A) \cup (A \cap B) &= (A - B) \cup (A \cap B) \cup (B - A) \\ &= (A \cap B') \cup (A \cap B) \cup (B - A) \end{aligned}$$

$$\begin{aligned} &= (A \cap (B' \cup B)) \cup (B - A) \\ &= (A \cap U) \cup (B \cap A') \\ &= A \cup (B \cap A') \\ &= (A \cup B) \cap (A \cup A') \\ &= (A \cup B) \cap U \\ &= A \cup B \end{aligned}$$

(e) $(A' \cap B')' = (A')' \cup (B')' = A \cup B$
(f) $(A' \cup B')' = (A')' \cap (B')' = A \cap B$
(g)

$$\begin{aligned} A \cup (B \cap (A - (B - A))) &= A \cup (B \cap (A \cap (B \cap A')')) \\ &= A \cup (B \cap (A \cap (B' \cup A))) \\ &= A \cup (B \cap ((A \cap B') \cup (A \cap A))) \\ &= A \cup (B \cap ((A \cap B') \cup A)) \\ &= A \cup (B \cap (A \cup (A \cap B'))) \\ &= A \cup (B \cap A) \\ &= A \cup (A \cap B) \\ &= A \end{aligned}$$

(h)

$$\begin{aligned} A' \cup (B - (A \cup (B' - A))) &= A' \cup (B - (A \cup (B' \cap A'))) \\ &= A' \cup (B - ((A \cup B') \cap (A \cup A'))) \\ &= A' \cup (B - (A \cup B')) \\ &= A' \cup (B \cap (A \cup B')') \\ &= A' \cup (B \cap (A' \cap (B')') \\ &= A' \cup (B \cap (A' \cap B)) \\ &= A' \cup (A' \cap B) \\ &= A' \end{aligned}$$

1.6. In a 2-set Venn diagram, the four disjoint regions are $A - B$, $A \cap B$, $B - A$, and $A' \cap B'$. Let us denote these R_1, R_2, R_3, and R_4, respectively. Then in each part of the exercise, the given equation can be rewritten so that each side is the union of one or more of the R_i's. One way to simplify the equation is just to observe that because any two of the R_i's are disjoint, any R_i that appears on one side of the equation and not the other must be empty, because any $x \in R_i$ would be an element of one side of the equation but not the other. Furthermore, the statement that certain R_i's are empty is the same as the statement that their union is empty (again using the fact that they are all disjoint). In several of the parts, the statement that the union is empty can be simplified.

(a) This equation says $R_1 \cup R_3 = R_1 \cup R_2$. It is therefore equivalent to $R_2 \cup R_3 = \emptyset$, or $B = \emptyset$.
(b) $R_1 \cup R_3 = R_1$, or $R_3 = \emptyset$. This means $B - A = \emptyset$, which is the same as $B \subseteq A$.
(c) $R_1 \cup R_3 = R_1 \cup R_2 \cup R_3$, or $R_2 = A \cap B = \emptyset$.
(d) $R_1 \cup R_3 = R_2$, or $R_1 \cup R_2 \cup R_3 = \emptyset$, or $A \cup B = \emptyset$, or $A = B = \emptyset$.
(e) $R_1 \cup R_3 = R_3 \cup R_4$, or $R_1 \cup R_4 = \emptyset$, or $B' = \emptyset$, or $B = U$.

1.7. (a) Two correct formulas are

$$(A \cup B) \cap (A \cap B)' \text{ and } (A \cap B') \cup (A' \cap B)$$

(b) Two correct formulas are

$$(A \cap B' \cap C') \cup (A' \cap B \cap C') \cup (A' \cap B' \cap C) \text{ and } (A \cup B \cup C) \cap ((A \cap B) \cup (A \cap C) \cup (B \cap C))'$$

(c) Two correct formulas are

$$(A \cap B' \cap C') \cup (A' \cap B \cap C') \cup (A' \cap B' \cap C) \cup (A' \cap B' \cap C') \text{ and } (A' \cap B') \cup (A' \cap C') \cup (B' \cap C')$$

(d) Two correct formulas are

$$(A \cap B) \cup (A \cap C) \cup (B \cap C) \cap (A \cap B \cap C)' \text{ and } (A \cap B \cap C') \cup (A \cap B' \cap C) \cup (A' \cap B \cap C)$$

(e) $(A \cup B \cup C) \cap (A \cap B \cap C)'$

1.8. (a) C_{10}
(b) D_1 (or C_1)
(c) C_1 (or D_1)
(d) D_{10}
(e) $\mathcal{R}$
(f) D_1 (or C_1)
(g) C_1 (or D_1)
(h) $x < 1/n$ for every $n \geq 1$ if and only if $x \leq 0$. Therefore, the answer is $\{x \mid x \leq 0\}$.
(i) $\mathcal{R}$
(j) $\emptyset$ (because no real number is less than n for every integer n)

1.10. The elements of $2^{2^{\{0,1\}}}$, or $2^{\{\emptyset,\{0\},\{1\},\{0,1\}\}}$, are (one per line)
$\emptyset$,
$\{\emptyset\}$,
$\{\{0\}\}$,
$\{\{1\}\}$,
$\{\{0,1\}\}$,
$\{\emptyset,\{0\}\}$,
$\{\emptyset,\{1\}\}$,
$\{\emptyset,\{0,1\}\}$,
$\{\{0\},\{1\}\}$,

$\{\{0\},\{0,1\}\}$,
$\{\{1\},\{0,1\}\}$,
$\{\emptyset,\{0\},\{1\}\}$,
$\{\emptyset,\{0\},\{0,1\}\}$,
$\{\emptyset,\{1\},\{0,1\}\}$,
$\{\{0\},\{1\},\{0,1\}\}$,
and $\{\emptyset,\{0\},\{1\},\{0,1\}\}$ (16 elements in all)

1.12. All are true.

1.13. In each case, a simpler statement is given.
(a) $\neg p$
(b) *true* (i.e., the statement is a tautology)
(c) $p \wedge q$
(d) q
(e) q
(f) q

1.14. In the case where p is false and q is true, $(p \wedge (p \rightarrow q))$ is false, so the conditional $r \rightarrow s$ must be true when r is false and s is true. In the case where p is true and q is false, $(p \wedge (p \rightarrow q))$ is false, so the conditional $r \rightarrow s$ must be true when r and s are false.

1.15. (a) $(m_1 < m_2) \vee ((m_1 = m_2) \wedge (d_1 < d_2))$
(b) $(m_1 \leq m_2) \wedge ((m_1 < m_2) \vee (d_1 < d_2))$. The second clause in this statement could also be written $((m_1 = m_2) \rightarrow (d_1 < d_2))$.

1.17. (a) neither
(b) contradiction
(c) neither
(d) tautology
(e) contradiction

1.18. $\forall x(\exists y(\exists t(L(x,y,t))))$ This is the most likely interpretation. However, if we interpret the statement to mean that there is a single time at which everybody loves somebody, the correct statement is $\exists t(\forall x(\exists y(L(x,y,t))))$.

1.19. "You can fool all the people some of the time" might be translated either $\exists t(\forall x(F(x,t)))$ or $\forall x(\exists t(F(x,t)))$, and similarly, there are two nonequivalent translations of "you can fool some of the people all the time." Therefore, there are at least four interpretations of the statement.

1.20. (a) True for $(0,1)$, false for $[0,1]$.
(b) True for any domain that is a subset of $\mathcal{R}$.

(c) False for any domain that is a subset of $\mathcal{R}$.
(d) False for $(0, 1)$, true for $[0, 1]$.

1.21. (a) B has at least n elements.
(b) B has at most n elements.

1.22. (a) f_a is one-to-one. The range of f_a is $\{x \in \mathcal{R}^+ \mid x \geq a\}$.
(b) d is one-to-one and onto.
(c) t is one-to-one. The range of t is $\{n \in \mathcal{N} \mid n \text{ is even}\}$.
(d) g is not one-to-one, but it is onto.
(e) p is one-to-one. The range of p is the set of all numbers x such that for some nonnegative integer n, $2n \leq x < 2n + 1$. In other words, the range is

$$\bigcup_{n=0}^{\infty} [2n, 2n + 1) = [0, 1) \cup [2, 3) \cup [4, 5) \cup \ldots$$

where $[a, b) = \{x \mid a \leq x < b\}$.
(f) i is not one-to-one. The range of i is $2^{[0,1]}$, the set of subsets of $[0, 1]$.
(g) u is not one-to-one. The range of u is $\{S \subseteq \mathcal{R} \mid [0, 1] \subseteq S\}$, the set of supersets of $[0, 1]$.
(h) m is not one-to-one. The range of m is the interval $[0, 2]$.
(i) M is not one-to-one. The range of M is the set $\{x \mid x \geq 2\}$.
(j) s is one-to-one. The range of s is $\{x \mid x \geq 2\}$. (Note that $s(x) = x + 2$ for every x.)

1.23. f is onto and g is one-to-one.

1.24(a) For $y \in B$, if there are several x's in A with $f(x) = y$, the formula $f(g(y)) = y$ will hold as long as $g(y)$ is defined to be any of these x's. So, for example, the formulas $g(7) = 2$, $g(8) = 4$, $g(9) = 3$, and $g(10) = 6$ define a function g that works, but $g(7)$ could also be 18, $g(9)$ could be 12, etc. In general, if f is onto, f will have at least one such "right inverse" g; if f is onto and not one-to-one, there will be more than one.
(b) The given information determines the values f must have on the elements 2, 6, 7, and 18. ($f(2)$ must be 9, since $f(g(9))$ is supposed to be 9; etc.) The values of f at the other three elements of A are arbitrary, and so there are many functions that would work.

1.25. (a) $g \circ d(x) = \lfloor 2x \rfloor$
(b) $t \circ g(x) = 2\lfloor x \rfloor$
(c) $t \circ t(x) = 4x$
(d) $d \circ f_a(x) = 2(x + a)$
(e) $f_a \circ d(x) = 2x + a$
(f) $g \circ f_a(x) = \lfloor x + a \rfloor$
(g) $u \circ i(A) = (A \cap [0, 1]) \cup [0, 1] = [0, 1]$
(h) $i \circ u(A) = (A \cup [0, 1]) \cap [0, 1] = [0, 1]$

1.26. (a) $f^{-1}(x) = x$

(b) Since every y in the codomain is positive, the equation $1/(1+x) = y$ can be rewritten $1+x = 1/y$, and this is the same as $x = 1/y - 1$. Clearly, for each y in the codomain, there is at most one x in the domain for which this equation holds, and thus f is one-to-one. Moreover, any y in the codomain is ≤ 1, so $1/y \geq 1$, so $1/y - 1 \geq 0$; therefore, the solution x is in fact in the domain of f, so that f is onto. The solution x to this equation is $f^{-1}(y)$, so we obtain $f^{-1}(y) = 1/y - 1$.

(c) For any two real numbers a and b, the equations $x + y = a$, $x - y = b$ have the unique solution $x = (a + b)/2$, $y = (a - b)/2$. Therefore, f is one-to-one and onto, and $f^{-1}(a, b) = ((a + b)/2, (a - b)/2)$.

1.28. (a) not reflexive, symmetric, not transitive.

(b) reflexive, not symmetric, transitive.

(c) not reflexive, but both symmetric and transitive.

1.29. In each case, a relation with as few pairs as possible is given.

reflexive	symmetric	transitive	
true	true	true	$\{(1,1),(2,2),(3,3)\}$
true	true	false	$\{(1,1),(2,2),(3,3),(1,2),(2,1),(2,3),(3,2)\}$
true	false	true	$\{(1,1),(2,2),(3,3),(1,2)\}$
true	false	false	$\{(1,1),(2,2),(3,3),(1,2),(2,3)\}$
false	true	true	$\emptyset$
false	true	false	$\{(1,2),(2,1)\}$
false	false	true	$\{(1,2)\}$
false	false	false	$\{(1,2),(2,3)\}$

1.30. (a) reflexive, not symmetric, transitive.

(b) reflexive (since we are considering a relation on the set of *nonempty* subsets of $\mathcal{N}$), symmetric, not transitive.

(c) not reflexive, symmetric, transitive.

1.31. The relation in (b) would no longer be reflexive.

1.32. not reflexive: $\exists x(\neg xRx)$
not symmetric: $\exists x(\exists y(xRy \wedge \neg yRx))$
not transitive: $\exists x(\exists y(\exists z(xRy \wedge yRz \wedge \neg xRz)))$

1.33(e) It's easy to see that R is reflexive and symmetric. If xRy and yRz, then there are numbers j and k so that $x_i = y_i$ for every $i \geq j$ and $y_i = z_i$ for every $i \geq k$. Therefore, if m is the larger of the two numbers j and k, $x_i = z_i$ for every $i \geq m$.

1.34. We observe first that two elements of A (i.e., two subsets of S) are related if and only if they are the same size. (Any two-element set is related to any other two-element set, and

it is not related to any three-element set.) An equivalence class must contain precisely all k-element subsets of S, for some k with $0 \leq k \leq 10$. There are eleven equivalence classes, one for each value of k. The equivalence class containing $\{a, b\}$ is the set of all two-element subsets of S, of which there are 45.

1.35. $\{0\}$ is an equivalence class, and for each nonzero real number x, $\{x, -x\}$ is an equivalence class.

1.36. (a) n (b) m

1.37. Two statements involving the propositional variables p, q, and r are equivalent if and only if they have the same truth table—in other words, they have the same truth values for each of the eight possible ways of assigning truth values to p, q, and r. Since a truth table has eight rows, there are 2^8 distinct truth tables, and since distinct truth tables correspond to distinct equivalence classes, there are $2^8 = 256$ equivalence classes.

1.38. $L^+ = L^*$ if and only if $\Lambda \in L$.

1.39. (a) $\{a\}^*$ and $\{x \in \{a, b\}^* \mid |x| \text{ is even}\}$ are two examples.
(b) $\{a\}^+$ and $\{x \in \{a, b\}^* \mid |x| \text{ is odd}\}$ are two examples..

1.40. (a) $L_1 = \{a\}$, $L_2 = \{aa\}$.
(b) $L_1 = \{a\}$, $L_2 = \{\Lambda, a\}$.

1.41. (a) $L_1^* \cup L_2^*$ is always a subset of $(L_1 \cup L_2)^*$. This is true because $L_1 \subseteq L_1 \cup L_2$, so that $L_1^* \subseteq (L_1 \cup L_2)^*$, and similarly $L_2^* \subseteq (L_1 \cup L_2)^*$. $L_1 = \{a\}$, $L_2 = \{b\}$ is an example where the opposite inclusion does not hold. For example, $ab \in (L_1 \cup L_2)^*$ and $ab \notin L_1^* \cup L_2^*$.
(b) $L_1 = \{aa, aaaaa\}$, $L_2 = \{aaa, aaaaa\}$. Then L_1^* contains a^i for every i except $i = 1$ and $i = 3$, and L_2^* contains a_i for every i except $i = 1$, $i = 2$, $i = 4$, and $i = 7$. Therefore, $L_1^* \cup L_2^* = (L_1 \cup L_2)^* = \{\Lambda\} \cup \{a^i \mid i \geq 2\}$.

1.43. For any language L, we have $L \subseteq L^+ \subseteq L^*$. From Exercise 1.42, we obtain $L^* \subseteq (L^+)^* \subseteq (L^*)^*$. We also have $L^* \subseteq (L^*)^+ \subseteq (L^*)^*$. Therefore, in order to show that all four languages are equal, it is sufficient to show that $(L^*)^* \subseteq L^*$. But an element of $(L^*)^*$ is formed by concatenating zero or more elements of L^*, each of which is the concatenation of zero or more elements of L. The result is a concatenation of zero or more elements of L.

1.44. Clearly, $|AB| \leq |A||B|$, since an element of AB is specified by first choosing an element of A and then choosing an element of B, and there are $|A||B|$ ways to do this. However, several of these choices might produce the same result. Let $A = B = \{a, aa\}$, for example. $|A||B| = 4$, but $|AB| = |\{aa, aaa, aaaa\}| = 3$.

1.45. One description of $\{a, ab\}^*$ is $\{x \in \{a, b\}^* \mid x \text{ does not start with } b \text{ and does not contain the substring } bb\}$.

1.46. (a) $L = \{a, ba\}^*$

(b) If there were such an S, then b would be an element of S^* (since it does not contain the substring bb), and therefore bb would be (since the concatenation of two elements of S^* is an element of S^*). But this is impossible.

1.47. (a) The two languages are equal.

(b) $(L_1 \cap L_2)^* \subseteq L_1^* \cap L_2^*$, and they are not always equal. For example, let $L_1 = \{a\}$ and $L_2 = \{aa\}$. Then $L_1^* \cap L_2^*$ contains the string aa, and $(L_1 \cap L_2)^* = \emptyset$.

(c) Neither is necessarily a subset of the other. For example, if $L_1 = \{a\}$ and $L_2 = \{b\}$, $aabb \in L_1^* L_2^* - (L_1 L_2)^*$ and $abab \in (L_1 L_2)^* - L_1^* L_2^*$.

(d) The two languages are equal.

1.48. One approach is to use the logic underlying Venn diagrams, without actually relying on the diagrams, by considering the eight cases separately. For example, in the case $x \in A \cap B' \cap C$, then $x \in A$ and $x \in B \oplus C$, so $x \notin A \oplus (B \oplus C)$; in addition, $x \in A \oplus B$ and $x \in C$, so $x \notin (A \oplus B) \oplus C$. In the case $x \in A' \cap B' \cap C'$, then $x \notin A$ and $x \notin B \oplus C$, so $x \notin A \oplus (B \oplus C)$; also, $x \notin (A \oplus B)$ and $x \notin C$, so $x \notin (A \oplus B) \oplus C$. The remainder of the proof is to show that in each of the other six cases also, $x \in A \oplus (B \oplus C)$ if and only if $x \in (A \oplus B) \oplus C$.

1.49. (c)

$$\begin{aligned} |A \cup B \cup C \cup D| \;=\; & |A| + |B| + |C| + |D| \\ & - |A \cap B| - |A \cap C| - |A \cap D| - |B \cap C| - |B \cap D| - |C \cap D| \\ & + |A \cap B \cap C| + |A \cap B \cap D| + |A \cap C \cap D| + |B \cap C \cap D| \\ & - |A \cap B \cap C \cap D| \end{aligned}$$

1.50. (a) 4 (b) One algorithm is described by the following pseudocode, which processes the symbols left-to-right, starting with the leftmost bracket.

```
read ch, the next nonblank character;
e = 0; (element count)
u = 1; (current number of unmatched left brackets)
while there are nonblank characters remaining
{ read ch, the next nonblank character;
  if ch is a left bracket
    u = u + 1;
  else if ch is a right bracket
    u = u - 1;
  if u = 1 and ch is either 0 or a right bracket
    e = e + 1;
}
```

1.51. (a) $\{a\}$ (b) $\{x \mid |x-a| < 1\}$

1.52. No. If A and B are distinct and both are nonempty, then either there is an $a \in A - B$ or there is a $b \in B - A$. In the first case, for any $x \in B$, the pair (a, x) is in $A \times B$ but not in $B \times A$; in the second case, for any $x \in A$, the pair (x, b) is in $A \times B$ but not in $B \times A$.

1.53(a) If either A or B is a subset of the other, then the two sets are equal. If not, then $2^A \cup 2^B \subseteq 2^{A \cup B}$; however, if $a \in A - B$ and $b \in B - A$, then the subset $\{a, b\}$ is an element of $2^{A \cup B}$ but is not in $2^A \cup 2^B$.

(b) $2^A \cap 2^B = 2^{A \cap B}$, because a set is a subset of $A \cap B$ if and only if it is both a subset of A and a subset of B.

(c) $2^{A \oplus B}$ can never be a subset of $2^A \oplus 2^B$, because $\emptyset$ is an element of the first but not the second. In the case where $A \cap B = \emptyset$, we have $2^A \oplus 2^B \subseteq 2^{A \oplus B}$. If $A \cap B \neq \emptyset$, however, and $A \neq B$, then a set containing an element of $A \cap B$ as well as an element of $A \oplus B$ is an element of $2^A \oplus 2^B$ that is not in $2^{A \oplus B}$.

(d) Again, the first set can never be a subset of the second, because $\emptyset \in 2^{A'}$ and $\emptyset \notin (2^A)'$. Every element of $2^{A'}$ that is not empty is an element of $(2^A)'$. $(2^A)' \subseteq 2^{A'}$ if and only if either A or A' is empty.

1.54. $(p \wedge q) \vee (\neg p \wedge \neg q)$

1.55. (a) $\exists x(P(x)) \wedge \forall x(\forall y((P(x) \wedge P(y) \rightarrow x = y)$ (The first part of the statement says that there is at least one such x; the second says that there is at most one.) Another possibility is $\exists x(P(x) \wedge \forall y(P(y) \rightarrow y = x))$.

(b) $\exists x(\exists y(P(x) \wedge P(y) \wedge x \neq y))$

1.56. (a) The second logically implies the first, but not vice-versa. For example, let $p(x)$ be the statement "x is even" and $q(x)$ the statement "x is odd". Then the first statement is true and the second is false.

(b) Each logically implies the other.

(c) Each logically implies the other.

(d) The first logically implies the second, but not vice-versa, as illustrated by the statements $p(x)$ and $q(x)$ in part (a).

1.57. (a) Yes. If $y \in f(S \cup T)$, then $y = f(x)$ for some $x \in S \cup T$, and in either case ($x \in S$ or $x \in T$), $y \in f(S) \cup f(T)$.

(b) Yes. Clearly $f(S) \subset f(S \cup T)$ and $f(T) \subset f(S \cup T)$; therefore, the union $f(S) \cup f(T)$ is also a subset of $f(S \cup T)$.

(c) Yes. If $y \in f(S \cap T)$, then $y = f(x)$ for some $x \in S \cap T$. Therefore, $y \in f(S)$, because $x \in S$, and $y \in f(T)$, because $x \in T$.

(d) No. If $y \in f(S) \cap f(T)$, then $y = f(x_1)$ for some $x_1 \in S$, and $y = f(x_2)$ for some $x_2 \in T$. However, x_1 and x_2 may be different, and there is no reason to assume that there is a single $x \in S \cap T$ with $y = f(x)$. For an example, let $A = \{a, b\}$, $B = \{c\}$, $S = \{a\}$, and $T = \{b\}$, and let $f : A \rightarrow B$ be the only possible function from A to B, the one with

$f(a) = f(b) = c$. Then $c \in f(S) \cap f(T)$, but $c \notin f(S \cap T)$ because $S \cap T = \emptyset$.

(e) In part (d), if f is one-to-one, the answer is yes.

1.58. (a) We can define f by saying, for each $S \in 2^X$, that $f(S)$ is the function $F_S : X \to \{0,1\}$ such that $F_S(x) = 1$ if $x \in S$ and $F_S(x) = 0$ if $x \notin S$. f is one-to-one because if $f(S) = F_S = f(T) = F_T$, then the set $\{x \in X \mid F_S(x) = 1\}$, which is just S, is the same as $\{x \in X \mid F_T(x) = 1\}$, which is T. f is onto, because for any function $F : X \to \{0,1\}$, if $S = \{x \in X \mid F(x) = 1\}$, F is just the function $F_S = f(S)$.

(b) We can define g by saying, for each function $t : X \to \{0,1\}$, that $g(t)$ is the n-tuple $(t(1), t(2), \ldots, t(n))$. It's easy to see that g is a bijection.

(c) We can define h by saying, for each n-tuple $N = (i_1, i_2, \ldots, i_n) \in \{0,1\}^n$, that $h(N)$ is the subset $\{x \in X \mid i_x = 1\}$. It is easy to see that h is a bijection.

1.59. (a) No, because the two subsets A and $\mathcal{N} - A$ are the same as the two subsets $\mathcal{N} - A$ and A. (That is, a partition of this type is an *unordered* pair of subsets, and if we start with one unordered pair, switching the two subsets produces the same unordered pair.) This means that f is not one-to-one.

(b) Yes. g is one-to-one, because if $g(A) = g(B)$, then the set $\{A, \mathcal{N} - A\}$ is the same as the set $\{B, \mathcal{N} - B\}$. Because A and B are both subsets of E, and $\mathcal{N} - A$ must therefore contain odd integers, A cannot be $\mathcal{N} - B$, and so A and B must be equal.

1.60. (a) For each $x \in I_0$, $x \in S$ for every $S \in \mathcal{S}$, and so $x \in \cap_{S \in \mathcal{S}} S$. If $x, y \in T$, then $x, y \in S$ for every $S \in \mathcal{S}$; therefore, since each $S \in \mathcal{S}$ is closed under $\circ$, $x \circ y \in S$ for each $S \in \mathcal{S}$; therefore, $x \circ y \in T$.

(b) Suppose $I_0 \subseteq T_1$ and T_1 is closed under $\circ$. Then $T_1 \in \mathcal{S}$, by definition of $\mathcal{S}$. For any $x \in T$, $x \in S$ for every $S \in \mathcal{S}$, and so $x \in T_1$.

1.61. The error is in the statement "choose some $b \in A$ so that aRb"—there may be no such b. The argument does prove that if R is symmetric and transitive, and for any x there is a y with xRy, then R is reflexive.

1.62(a) The same as the number of subsets of $A \times A$, or $2^{(n^2)}$.

(b) Of the n^2 elements in $A \times A$, n are of the form (i, i). Specifying a reflexive relation on A the same as specifying the subset of the remaining $n^2 - n$ pairs that are in the relation in addition to these n. The answer is therefore 2^{n^2-n}.

(c) To make the solution easier to describe, take the set A to be $\{1, 2, \ldots, n\}$. If R is symmetric, specifying R is accomplished by specifying the pairs (i, j) in R for which $i \leq j$. Therefore, the number of symmetric relations on A is the number of subsets of the set $\{(i, j) \mid 1 \leq i \leq j \leq n\}$. Since this set has $n(n+1)/2$ elements, the number of symmetric relations is $2^{n(n+1)/2}$.

(d) The number of subsets of $\{(i, j) \mid 1 <= i < j <= n\}$, which is $2^{n(n-1)/2}$.

1.63. (a) Another way to describe R^s is $\{(x, y) \mid xRy \text{ or } yRx\}$; now it is easy to see that R^s is symmetric. If R_1 is a symmetric relation with $R \subseteq R_1$, then for every $x, y \in A$, if

xRy, then $(x, y) \in R_1$ (because $R \subseteq R_1$) and $(y, x) \in R_1$ (because R_1 is symmetric); therefore, $R^s \subseteq R_1$.

(b) Suppose $xR^t y$ and $yR^t z$, and let R_1 be any transitive relation containing R. Then (x, y) and (y, z) are in R_1, since they are in R, and so $(x, z) \in R_1$ because R_1 is transitive. Since this is true for any transitive R_1 containing R, $(x, z) \in R^t$. Furthermore, if R_1 is any transitive relation containing R, then R_1 is one of the relations of which R^t is the intersection, so that $R^t \subseteq R_1$.

(c) R^u and R^t are not always the same. For an example, let $R = \{(n, n+1) \mid n \in \mathcal{N}\}$. Then $R^u = \{(n, m) \mid n \in \mathcal{N}$ and $(m = n + 1$ or $m = n + 2)\}$, whereas R^t contains every pair $(n, n + k)$ for which $k > 0$.

1.64. The function $f : 2^S \to \mathcal{N}$ defined by $f(A) = |A|$ (the number of elements of A) is such a function.

1.65. The function f defined by $f(i) = i \bmod n$ is such a function. (Two integers are congruent mod n if and only if they have the same remainder mod n.)

1.66. We may take B to be the set of equivalence classes with respect to the relation R, and f to be the function defined by $f(x) = [x]$ (the equivalence class containing x).

1.67. Suppose first that S is a maximal pairwise inequivalent set. Let C be any equivalence class. Since S is pairwise inequivalent, S cannot contain more than one element of C, and since S is maximal S must contain at least one (if it didn't, we could add one without destroying the pairwise inequivalent property). On the other hand, if S is any set containing exactly one element from each equivalence class, then S is pairwise inequivalent, because an element of one equivalence class cannot be related to an element from any other equivalence class; and S is maximal, because for any $x \in A$, S contains an element of the equivalence class $[x]$.

1.68. Suppose that $R_1 \subset R_2$. Consider a subset S in the partition P_2 and any element $x \in S$. Let $[x]_1$ be the equivalence class containing x with respect to R_1. ($[x]_1$ is the subset in the partition P_1 containing x.) Then for any $y \in [x]_1$, since xR_1y and $R_1 \subseteq R_2$, $y \in S$. In other words, $[x]_1 \subseteq S$. It follows that S is the union of all the sets $[x]_1$, for the elements $x \in S$. On the other hand, if every subset S in the partition P_2 is the union of subsets in the partition P_1, then for any x and y with xR_1y, x and y are in the same equivalence class relative to R_1, so they are in the same subset in the partition P_2, so xR_2y.

1.69. Obviously, $xy = yx$ if for some string α, $x = \alpha^j$ and $y = \alpha^k$. In fact, this is the only way xy and yx can be equal, although the proof is a little complicated.

Suppose that $xy = yx$. If we can find a string α so that $x = \alpha^j$ and $y = \alpha^k$ for some k and some j, then obviously $|\alpha|$ is a divisor of both $|x|$ and $|y|$. With this in mind, let d be the greatest common divisor of $|x|$ and $|y|$. Then we can write $x = x_1x_2 \ldots x_p$ and $y = y_1y_2 \ldots y_q$, where all the x_i's and y_i's are of length d and p and q have no common factors. (What we'd like to show is that these strings are all identical.)

Since $xy = yx$, it must be true that $x^q y^p = y^p x^q$, because starting with one of these strings, we can obtain the other by repeated transpositions of x and y. $x^q y^p$ and $y^p x^q$ are both strings of length $2pqd$, and the prefixes of length pqd are x^q and y^p, respectively; therefore, these two strings are equal. Now $x^q = (x_1 x_2 \dots x_p)^q$. This means that in the string x^q, the substring x_1 occurs starting at positions 1, $pd+1$, $2pd+1$, $\dots$ $(q-1)pd+1$. In the string y^p, the substring of length d starting at position $ipd+1$ is y_{r_i}, where $r_i = ip(\text{mod } q) + 1$. Since p and q have no common factors, it is easy to check that all the numbers $r_0, r_1, \dots, r_{q-1}$ are distinct. This means, however, that all the strings $y_1, y_2, \dots, y_q$ are the same, say α, and this makes it clear that all the x_i's are equal to α as well.

1.70. If there were such an L, L^* would have to contain both the strings ab and aa, since these are both elements of $\{aa, bb\}^*\{ab, ba\}^*$. However, L^* would then contain $abaa$, which is not an element of $\{aa, bb\}^*\{ab, ba\}^*$.

1.71. No. Suppose $L = L_1 L_2$ and neither L_1 nor L_2 is $\{\Lambda\}$. Since any even-length string of 0's is in L, there are arbitrarily long strings of 0's that must be in either L_1 or L_2. Similarly, there are arbitrarily long strings of 1's that are in L_1 or L_2. It is not possible for L_1 to have a nonnull string of 0's and L_2 to have a nonnull string of 1's, since the concatenation could not be in L; similarly, there can't be a string of 1's in L_1 and a string of 0's in L_2. The only possibilities, therefore, are for all the strings of 0's and all the strings of 1's to be in L_1 or for all these strings to be in L_2. Suppose, however, that they are all in L_1. Let y be a nonnull string in L_2. Then y contains both 0's and 1's. L_1 contains a string x of 0's with $|x| \geq |y|$. But then xy, which is in $L_1 L_2$, has 1's in the second half and not in the first, so that it can't be in L. A similar argument shows that L_2 can't contain all the strings of 0's and the strings of 1's. This argument shows that our original assumption, that $L = L_1 L_2$ and neither L_1 nor L_2 is $\{\Lambda\}$, must not be true.

Chapter 2

Mathematical Induction and Recursive Definitions

2.5. The proof is identical except that the proof of the induction step now goes like this. Since $|x| = k+1$ and $x = 0y1$, $|0y| = k$. If y ends with 0, then x ends with the substring 01. If y ends with 1, then $0y$ begins with 0 and ends with 1; by the induction hypothesis, $0y$ contains the substring 01, and therefore x does also.

2.6. To show that for any $n \geq 1$, if x contains n 0's and $x = 0y1$, then x contains the substring 01.
Basis step. If x contains only one 0 and $x = 0y1$, then x begins with 01.
Induction hypothesis. $k \geq 1$, and if x contains k 0's and $x = 0y1$, then x contains the substring 01.
Statement to be shown in induction step. If x contains $k+1$ 0's and $x = 0y1$, then x contains the substring 01.
Proof. Consider the string $y1$, which has k 0's. If y starts with 1, then x begins with 01. Otherwise, $y1$ starts with 0 and ends with 1. Therefore, by the induction hypothesis, $y1$ contains the substring 01, and so x does also.

2.7. Proving the basis step would be as difficult as proving the original statement.

2.8. **Induction hypothesis.** $k \geq 0$ and $\sum_{i=1}^{k} i^2 = k(k+1)(2k+1)/6$.
Statement to be shown in induction step. $\sum_{i=1}^{k+1} i^2 = (k+1)((k+1)+1)(2(k+1)+1)/6$
Proof of induction step.

$$\begin{aligned}
\sum_{i=1}^{k+1} i^2 &= \sum_{i=1}^{k} i^2 + (k+1)^2 \\
&= k(k+1)(2k+1)/6 + (k+1)^2 \\
&= (k+1)(k(2k+1)/6 + (k+1)) \\
&= (k+1)(2k^2+7k+6)/6 \\
&= (k+1)(k+2)(2k+3)/6 \\
&= (k+1)((k+1)+1)(2(k+1)+1)/6
\end{aligned}$$

(The second equality is because of the induction hypothesis.)

2.9. **Proof of induction step** (using ordinary induction).

$$\begin{aligned}
\sum_{i=1}^{k+1}(a_i - a_{i-1}) &= \sum_{i=1}^{k}(a_i - a_{i-1}) + (a_{k+1} - a_k) \\
&= (a_k - a_0) + (a_{k+1} - a_k) = a_{k+1} - a_0
\end{aligned}$$

2.10. **Proof of induction step** (using ordinary induction).

$$\begin{aligned} 7+13+19+ \ \ldots \ +(6k+1)+(6(k+1)+1) &= (7+13+\ldots+6k+1)+(6(k+1)+1) \\ &= k(3k+4)+(6k+7) \text{ (using the induction hypothesis)} \\ &= 3k^2+10k+7=(k+1)(3k+7)=(k+1)(3(k+1)+4) \end{aligned}$$

2.11. **Proof of induction step** (using ordinary induction).

$$\begin{aligned} \sum_{i=1}^{k+1}\frac{1}{i(i+1)} &= \sum_{i=1}^{k}\frac{1}{i(i+1)}+\frac{1}{(k+1)(k+1+1)} \\ &= \frac{k}{k+1}+\frac{1}{(k+1)(k+2)} \\ &= \frac{k(k+2)+1}{(k+1)(k+2)} \\ &= \frac{(k+1)^2}{(k+1)(k+2)} \\ &= \frac{k+1}{k+2} \end{aligned}$$

2.12. (a)

$$\begin{aligned} C(n-1,i-1)+C(n-1,i) &= \frac{(n-1)!}{(i-1)!(n-1-(i-1))!}+\frac{(n-1)!}{i!(n-1-i)!} \\ &= \frac{(n-1)!}{(i-1)!(n-i)!}+\frac{(n-1)!}{i!(n-1-i)!} \\ &= \frac{(n-1)!i}{i(i-1)!(n-i)!}+\frac{(n-1)!(n-i)}{i!(n-i)(n-i-1)!} \\ &= \frac{(n-1)!i}{i!(n-i)!}+\frac{(n-1)!(n-i)}{i!(n-i)!} \\ &= \frac{(n-1)!(i+n-i)}{i!(n-i)!} \\ &= \frac{n(n-1)!}{i!(n-i)!} \\ &= \frac{n!}{i!(n-i)!}=C(n,i) \end{aligned}$$

(b) **Proof of induction step** (using ordinary induction).

$$\sum_{i=0}^{k+1}C(k+1,i) = C(k+1,0)+\sum_{i=1}^{k}C(k+1,i)+C(k+1,k+1)$$

$$\begin{aligned}
&= 1+\sum_{i=1}^{k}(C(k,i-1)+C(k,i))+1\\
&= 1+\sum_{i=0}^{k-1}C(k,i)+\sum_{i=1}^{k}C(k,i)+1\\
&= \sum_{i=0}^{k}C(k,i)+\sum_{i=0}^{k}C(k,i)\\
&= 2\sum_{i=0}^{k}C(k,i)\\
&= 2(2^k)=2^{k+1}
\end{aligned}$$

2.13. **Proof of induction step** (using ordinary induction).

$$\begin{aligned}
\sum_{i=0}^{k+1}r^i &= \sum_{i=0}^{k}r^i+r^{k+1}\\
&= \frac{1-r^{k+1}}{1-r}+r^{k+1}\\
&= \frac{1-r^{k+1}+r^{k+1}-r^{k+2}}{1-r}\\
&= \frac{1-r^{(k+1)+1}}{1-r}
\end{aligned}$$

2.14. **Proof of induction step** (using ordinary induction).

$$\begin{aligned}
1+\sum_{i=1}^{k+1}i*i! &= 1+\sum_{i=1}^{k}i*i!+(k+1)*(k+1)!\\
&= (k+1)!+(k+1)(k+1)!\\
&= (k+1)!(1+k+1)=(k+2)!
\end{aligned}$$

2.15. **Proof of induction step** (using ordinary induction).

$$(k+1)!=(k+1)*k!\geq 2*k!>2*2^k=2^{k+1}$$

The first inequality uses the fact that $k\geq 1$, and the second inequality uses the induction hypothesis.

2.16. We can consider any $m\geq 0$, and then prove, by induction on n, the statement: for every $n\geq m$, $a_m\leq a_n$.
Statement to be proved in induction step. $a_m\leq a_{k+1}$

Proof of induction step. By the induction hypothesis, $a_m \leq a_k$; by the original assumption, $a_k \leq a_{k+1}$. Therefore, by the transitivity of the $\leq$ relation, $a_m \leq a_{k+1}$.

2.17. **Basis step.** The statement to be proved is $(1+x)^0 \geq 1+0x$, which is clear.
Induction hypothesis. $k \geq 0$ and $(1+x)^k \geq 1+kx$.
Statement to be proved in induction step. $(1+x)^{k+1} \geq 1+(k+1)x$.
Proof.

$$\begin{aligned}(1+x)^{k+1} &= (1+x)^k * (1+x) \\ &\geq (1+kx) * (1+x) \\ &= 1+kx+x+kx^2 \\ &= 1+(k+1)x+kx^2 \\ &\geq 1+(k+1)x\end{aligned}$$

The first inequality uses the induction hypothesis, the fact that $x+1 > 0$, and the algebraic fact that if $A \geq B$ and $C > 0$, then $AC \geq BC$ (where $A = (1+x)^k$, $B = 1+kx$, and $C = 1+x$). The second inequality uses the fact that for any x, $x^2 \geq 0$.

2.18. **Basis step.** If we let $k_1 = 2$, then $\sum_{i=1}^{k_1} 1/i = 1/1 + 1/2 > 1$.
Induction hypothesis. $j \geq 1$, and there is an integer k_j so that $\sum_{i=1}^{k_j} 1/i > j$.
Statement to be proved in induction step. There is an integer k_{j+1} so that $\sum_{i=1}^{k_{j+1}} 1/i > j+1$.
Proof. First we write

$$\sum_{i=1}^{k_{j+1}} 1/i = \sum_{i=1}^{k_j} 1/i + 1/(k_j+1) + \ldots + 1/k_{j+1}$$

According to the induction hypothesis, the first sum on the right side is greater than j. Therefore, all we need to show is that we can choose k_{j+1} so that the sum of the remaining terms is at least 1. The hint suggests that we consider the sum $1/(k_j+1) + 1/(k_j+2) + \ldots + 1/(2k_j)$. This is the sum of k_j terms, each of which is at least $1/(2k_j)$; therefore, this sum is greater than $1/2$. Now we may repeat this step, starting with $1/(2k_j+1)$: the sum $1/(2k_j+1) + 1/(2k_j+2) + \ldots + 1/(2(2k_j))$ is also greater than $1/2$. Now we choose $k_{j+1} = 4k_j$, and we have

$$\sum_{i=1}^{k_{j+1}} 1/i = \sum_{i=1}^{k_j} 1/i + \sum_{i=k_j+1}^{2k_j} 1/i + \sum_{i=2k_j+1}^{4k_j} 1/i > j + 1/2 + 1/2 = j+1$$

2.19. **Proof of induction step** (using ordinary induction).

$$\sum_{i=1}^{k+1} i * 2^i = \sum_{i=1}^{k} i * 2^i + (k+1) * 2^{k+1}$$

$$\begin{aligned} &= (k-1)*2^{k+1}+2+(k+1)*2^{k+1} \\ &= (k-1+k+1)*2^{k+1}+2 \\ &= k*2^{k+2}+2 \end{aligned}$$

2.20. **Proof of induction step** (using ordinary induction).

$$\begin{aligned} 1+\sum_{i=2}^{k+1}\frac{1}{\sqrt{i}} &= 1+\sum_{i=2}^{k}\frac{1}{\sqrt{i}}+\frac{1}{\sqrt{k+1}} \\ &> \sqrt{k}+\frac{1}{\sqrt{k+1}} \\ &= \frac{\sqrt{k}\sqrt{k+1}+1}{\sqrt{k+1}} \\ &> \frac{\sqrt{k}\sqrt{k}+1}{\sqrt{k+1}} \\ &= \frac{k+1}{\sqrt{k+1}}=\frac{1}{\sqrt{k+1}} \end{aligned}$$

2.21. **Basis step.** 0 is even, and therefore 0 is either even or odd.
Induction hypothesis. $k \geq 0$, and k is either even or odd.
Statement to be shown in induction step. $k+1$ is either even or odd.
Proof. We consider two cases, depending on whether k is even or odd. If k is even, then there is an integer i with $k = 2i$. In this case $k+1 = 2i+1$. It follows by the definition that $k+1$ is odd, and therefore that $k+1$ is either even or odd. If k is odd, then there is an integer i with $k = 2i+1$. Then $k+1 = 2i+2 = 2(i+1)$. Since there is an integer j (namely, $j = i+1$) for which $k+1 = 2j$, $k+1$ is even, and therefore $k+1$ is either even or odd.

2.22. We formulate the statement as follows: for every $n \geq 1$, if $L^2 \subseteq L$, then $L^n \subseteq L$. (It will then follow that if $L^2 \subseteq L$, then $\cup_{n=1}^{\infty} L^n = L^+ \subseteq L$.)
Basis step. $L^1 = L$, and so $L^1 \subseteq L$. (Here we don't even need the assumption that $L^2 \subseteq L$.
Induction hypothesis. $k \geq 1$, and if $L^2 \subseteq L$, then $L^k \subseteq L$.
Statement to be shown in induction step. If $L^2 \subseteq L$, then $L^{k+1} \subseteq L$.
Proof. If $L^2 \subseteq L$, then $L^k \subseteq L$ because of the induction hypothesis. Therefore, $L^{k+1} = L^k L \subseteq LL$. Therefore, if $LL \subset L$, it follows that $L^{k+1} \subseteq L$.

2.23. We show this statement by structural induction, using the usual recursive definition of Σ^*.
Basis step. To show $f(\Lambda) = \Lambda$. Using the assumption on f, we know that $f(\Lambda) = f(\Lambda\Lambda) = f(\Lambda)f(\Lambda)$. The only string z satisfying the equation $z = zz$ is the string Λ.
Induction hypothesis. x is a string for which $f(x) = x$.
Statement to be shown in induction step. For any $a \in \Sigma$, $f(xa) = xa$.

Proof. For any $a \in \Sigma$, $f(xa) = f(x)f(a)$ by assumption. By the induction hypothesis, $f(x) = x$, and by the assumption on f, $f(a) = a$. Therefore, $f(xa) = xa$.

2.24. **Proof of induction step** (using ordinary induction).
$(k+1)((k+1)^2+5) = (k+1)^3+5(k+1)+5 = k^3+3k^2+3k+1+5(k+1) = k^3+3k^2+8k+6$. This can be rewritten as $k^3 + 5k + (3k^2 + 3k + 6)$, or $k(k^2 + 5) + (3k(k + 1) + 6)$. The induction hypothesis tells us that the first term is divisible by 6. For any k, either k or $k + 1$ is even, so that $k(k + 1)$ is even, and therefore $3k(k + 1)$ is divisible by 6. Therefore, the entire expression is.

2.25. **Basis step.** To show that for every a and b with $0 \leq a < b$, $b^1 - a^1$ is divisible by $b - a$. This is obvious.
Induction hypothesis. $k \geq 1$, and for every a and b with $0 \leq a < b$, $b^k - a^k$ is divisible by $b - a$.
Statement to be shown in induction step. For every a and b with $0 \leq a < b$, $b^{k+1} - a^{k+1}$ is divisible by $b - a$.
Proof. We can write

$$b^{k+1} - a^{k+1} = b * b^k - a * a^k = b * b^k - a * b^k + a * b^k - a * a^k = (b - a) * b^k + a * (b^k - a^k)$$

The first term is obviously divisible by $b - a$, and the induction hypothesis says that the second is also. Therefore, the sum is divisible by $b - a$.

2.26. **Basis step.** $1 = 2^0 * 1$, which is a power of 2 times an odd integer.
Induction hypothesis. $k \geq 1$, and any integer n with $1 \leq n \leq k$ is the product of a power of 2 and an odd integer.
Statement to be shown in induction step. $k + 1$ is the product of a power of 2 and an odd integer.
Proof. We consider two cases. If $k + 1$ is odd, then $k + 1$ is the product of a power of 2 (namely, 2^0) and an odd integer (namely, $k + 1$). If $k + 1$ is even, then $k + 1 = 2 * j$ for some $j \geq 1$. By the induction hypothesis, $j = 2^p * m$ for some nonnegative integer p and some odd integer m. Therefore, $k + 1 = 2 * j = 2^{p+1}m$.

2.27. **Proof of induction step** (using ordinary induction).

$$(\bigcap_{i=1}^{k+1} A_i)' = (\bigcap_{i=1}^{k} A_i \cap A_{k+1})' = (\bigcap_{i=1}^{k} A_i)' \cup A'_{k+1} = \bigcup_{i=1}^{k} A'_i \cup A'_{k+1} = \bigcup_{i=1}^{k+1} A'_i$$

The second equality follows from De Morgan's law, formula (1.12), and the third follows from the induction hypothesis.

2.28. Basis step. There are two subsets of $\{1\}$, namely, $\emptyset$ and $\{1\}$.
Induction hypothesis. $k \geq 1$, and there are 2^k subsets of $\{1, 2, \ldots, k\}$.
Statement to be shown in induction step. There are 2^{k+1} subsets of $\{1, 2, \ldots, k+1\}$.
Proof. By the induction hypothesis, there are 2^k subsets of $\{1, 2, \ldots, k\}$. Each such subset

A determines exactly two subsets of $\{1, 2, \ldots, k+1\}$ (namely, A and $A \cup \{k+1\}$), and all of the 2^{k+1} subsets obtained this way are distinct. Furthermore, every subset B of $\{1, 2, \ldots, k+1\}$ arises this way, since if $k+1 \notin B$ then B is a subset of $\{1, 2, \ldots, k\}$ and otherwise $B = (B - \{k+1\}) \cup \{k+1\}$. Therefore, the number of subsets of $\{1, 2, \ldots, k+1\}$ is 2^{k+1}.

2.29. This is a generalization of 2.28, since a subset of $\{1, 2, \ldots, n\}$ can be viewed as a function from $\{1, 2, \ldots, n\}$ to a two-element set, say {true, false}. Of the two integers m and n, n is the more appropriate one on which to base the induction. A function from $\{1, 2, \ldots, k+1\}$ to $\{1, 2, \ldots, m\}$ is specified by first saying how it is defined on the elements of $\{1, 2, \ldots, k\}$ and then saying how it is defined at $k+1$. By the induction hypothesis, there are m^k ways to do the first, and there are m ways to do the second, since the value can be any one of the m possible values. Therefore, there are $m^k * m = m^{k+1}$ ways to specify the function.

2.30. **Proof of the induction step** (using ordinary induction).

$$\begin{aligned}
\frac{d}{dx}(x^{k+1}) &= \frac{d}{dx}(x^k * x) \\
&= x^k * \frac{d}{dx}x + \frac{d}{dx}(x^k) * x \\
&= x^k * 1 + kx^{k-1} * x = x^k + kx^k = (k+1)x^k
\end{aligned}$$

2.31 **Basis step.** $a_0 = -2 = 2 * 1 - 4 * 1 = 2 * 3^0 - 4 * 2^0$.
Induction hypothesis. $k \geq 0$, and for every n with $0 \leq n \leq k$, $a_n = 2 * 3^n - 4 * 2^n$.
Statement to be shown in induction step. $a_{k+1} = 2 * 3^{k+1} - 4 * 2^{k+1}$.
Proof. If $k = 0$, $a_{k+1} = -2$, which is $2 * 3^1 - 4 * 2^1$. Otherwise,

$$\begin{aligned}
a_{k+1} &= 5a_k - 6a_{k-1} \\
&= 5(2 * 3^k - 4 * 2^k) - 6(2 * 3^{k-1} - 4 * 2^{k-1}) \\
&= 10 * 3^k - 12 * 3^{k-1} - 20 * 2^k + 24 * 2^{k-1} \\
&= 10 * 3^k - 4 * 3^k - 20 * 2^k + 12 * 2^k \\
&= 6 * 3^k - 8 * 2^k = 2 * 3^{k+1} - 4 * 2^{k+1}
\end{aligned}$$

2.32. (a) **Induction Hypothesis.** $k \geq 0$, and for every n with $0 \leq n \leq k$, $f_n < C(13/8)^n$.
Statement to be shown in induction step. $f_{k+1} < C(13/8)^{k+1}$.
Proof. If $k+1 = 1$, then $f_{k+1} = 1 < C(13/8)$, because of the assumption that $C < 8/13$. If $k+1 > 1$, then by definition $f_{k+1} = f_k + f_{k-1}$. According to the induction hypothesis, $f_k < C(13/8)^k$ and $f_{k-1} < C(13/8)^{k-1}$. Therefore,

$$f_{k+1} < C((13/8)^k + (13/8)^{k-1}) = C(13/8)^{k-1}(13/8+1) = C(13/8)^{k-1}(168/64)$$
$$< C(13/8)^{k-1}(169/64) = C(13/8)^{k+1}$$

(b) **Proof of induction step** (using ordinary induction).

$$\begin{aligned}\sum_{i=0}^{k+1} f_i^2 &= \sum_{i=0}^{k} f_i^2 + f_{k+1}^2 \\ &= f_k f_{k+1} + f_{k+1}^2 = f_{k+1}(f_k + f_{k+1}) = f_{k+1}f_{k+2}\end{aligned}$$

(c) **Proof of induction step** (using ordinary induction).

$$\begin{aligned}\sum_{i=0}^{k+1} f_i &= \sum_{i=0}^{k} f_i + f_{k+1} \\ &= f_{k+2} - 1 + f_{k+1} = f_{k+3} - 1\end{aligned}$$

2.33. (a) **Proof of induction step** (using ordinary induction).

$$f(k+1) = \sqrt{1 + f(k)} < \sqrt{1+2} < \sqrt{2+2} = 2$$

(The first inequality follows from the induction hypothesis, and the fact that if $0 \le a < b$, then $\sqrt{a} < \sqrt{(b)}$.)

(b) **Proof of induction step** (using ordinary induction).

$$f(k+1) = \sqrt{1 + f(k)} < \sqrt{1 + f(k+1)} = f(k+2)$$

2.34. Assuming that our recursive definition of Σ^* is still the first one in Example 2.15, it would appear that induction on x doesn't work, because in the induction step, we have the expression $|(xa)y|$, which we have no way to simplify. If we used a different recursive definition of Σ^*, in which symbols are added at the left end, and the corresponding recursive definition of $|x|$ (i.e., using the formula $|ax| = |x| + 1$), then it would be possible.

2.35. In the induction step of a proof using structural induction on x, we would have $|(xa)^r| = |ax^r|$. Using Example 2.24, this is $|a| + |x^r|$, which is $|a| + |x|$ by the induction hypothesis. So to complete the proof we only need to show that $|a| = 1$, which is easy: $|a| = |\Lambda a| = |\Lambda| + 1 = 1$.

2.36. We use structural induction, based on the definition of AE. **Basis step.** Every prefix of i contains at least as many ('s as)'s. This is obvious.
Induction hypothesis. For some strings x and y in AE, every prefix of x or y contains

at least as many ('s as)'s.
Statement to be shown in induction step. Every prefix of $(x + y)$ contains at least as many ('s as)'s, and similarly for $(x - y)$.
Proof. We show the first statement, and the proof of the second is virtually identical. Let $z = (x + y)$. A prefix of z must be either Λ, or $(x_1$ for some prefix x_1 of x, or $(x + y_1$, for some prefix y_1 of y, or z. The first of these four cases is obvious. In the second case, since by the induction hypothesis x_1 has at least as many ('s as)'s, the string $(x_1$ actually has *more* left parentheses than right. In the third case, the induction hypothesis implies that both x and y_1 have at least as many ('s as)'s (note: x is a prefix of itself), and so $(x + y_1$ does also. Finally, in the fourth case the induction hypothesis implies that x and y both have at least as many ('s as)'s, from which it follows that $(x + y)$ does also.

2.37. (a) The basis step is easy, since i is in both languages by definition.
Induction Hypothesis. x and y are in *GAE*.
Statement to be shown in induction step. $(x + y)$ and $(x - y)$ are in *GAE*.
Proof. The strings $x + y$ and $x - y$ are in *GAE*, by part (ii) of the definition. Therefore, both $(x + y)$ and $(x - y)$ are also, by part (iii).

The proof of (b) is similar to that in Exercise 2.36. In (c), the number $N(x)$ is the maximum depth of parenthesis nesting in the expression x—i.e., if $N(x) = k$, there are k nested pairs of parentheses in the expression, and nowhere in x are there more than k.

2.38. Suppose we know that for any $k \geq n_0$, if $P(n)$ is true for every n with $n_0 \leq n < k$, then $P(k)$ is true. We can show that $P(n)$ is true for every $n \geq n_0$ using the strong principle of induction. The basis step is to show that $P(n_0)$ is true. However, our assumption for the value $k = n_0$ is that if $P(n)$ is true for every n with $n_0 \leq n < n_0$, then $P(n_0)$ is true, and $P(n)$ *is* true for every n in this range, because there are no n's in this range.

The induction step is to show that if $k \geq n_0$ and $P(n)$ is true for every n with $n_0 \leq n \leq k$, then $P(k+1)$ is true. This follows from the assumption, using $k+1$ instead of k: we know that if $P(n)$ is true for every n with $n_0 \leq n < k+1$, then $P(k+1)$ is true.

2.39. (a) The set of all strings beginning with a.
(b) The set of all strings containing exactly one a.
(c) The set of all strings containing at least one a and not containing the substring ba. It can also be described as the set of all strings containing at least one a in which all the a's precede all the b's.
(d) The set of all strings containing at least one a.
(e) The set of all strings containing at least one a.
(f) The set of all strings starting with a that do not contain the substring aa.

2.40. (Each of the definitions is assumed to contain a final statement that nothing else is in the language.)
(a) $0 \in \mathcal{N}$; for every $n \in \mathcal{N}$, $n + 1 \in \mathcal{N}$.
(b) $0 \in S$; for every $n \in S$, both $n - 7$ and $n + 7$ are in S.
(c) $2 \in T$; $7 \in T$; for every $n \in T$ and every positive integer i, $in \in T$.

(d) $00 \in U$; for every $x \in U$, all the strings $0x$, $1x$, $x0$, and $x1$ are in U.
(e) $\Lambda \in V$; for every $x \in V$, both $0x1$ and $00x1$ are in V.
(f) $\Lambda \in W$; for every $x \in W$, both $0x$ and $00x1$ are in W.

2.41. **Proof of the induction step** (using strong induction).
Suppose $x \in L$ and $|x| = k+1$. Then $x \neq \Lambda$, and so there are two possibilities for x: either $x = 0y$ for some $y \in L$, or $x = 0y1$ for some $y \in L$. In either case, $y = 0^i1^j$ for some i and j with $i \geq j \geq 0$, and since $|y| \leq k$, $y \in A$ by the induction hypothesis. In the first case, $x = 0^{i+1}1^j$ and $i+1 \geq j \geq 0$, so that $x \in A$; in the second case, $x = 0^{i+1}1^{j+1}$ and $i+1 \geq j+1 \geq 0$, so that $x \in A$.

2.42 (b) First, to show $L_2 \subseteq L$. This is an easy proof using structural induction.
Basis step. Both the strings Λ and a are clearly in L, because neither contains the substring aa.
Induction hypothesis. x is a string in L_2 that is an element of L.
Statement to be proved in induction step. bx and abx are in L.
Proof. These are both obvious. If x does not contain the substring aa, neither does bx or abx.

Now to show $L \subseteq L_2$, using strong induction on the length of the string.
Basis step. If $x \in L$ and $|x| = 0$, then $x = \Lambda$, which is in L_2 according to the definition of L_2.
Induction hypothesis. $k \geq 0$, and for any $x \in L$ with $|x| \leq k$, $x \in L_2$.
Statement to be proved in induction step. If $x \in L$ and $|x| = k+1$, then $x \in L_2$.
Proof. If $x = by$, then since x does not contain the substring aa, neither does y. Therefore, $y \in L$, and so by the induction hypothesis, $y \in L_2$. Therefore, according to the first portion of the recursive part of the definition of L_2, $by \in L_2$. Otherwise $x = ay$ for some y (because $x \neq \Lambda$), and since we know that $x \in L$, y cannot start with a. If $y = \Lambda$, then $x = a$, and $x \in L_2$ by the definition of L_2. If $y \neq \Lambda$, then $y = bz$, so that $x = abz$. The string z cannot contain the substring aa, and so $z \in L_2$ by the induction hypothesis. Therefore, by the second portion of the recursive part of the definition of L_2, $abz \in L_2$.

2.43.(c) The strings a_0 and b_0 are clearly palindromes. For $n > 0$,

$$a_{2n} = a_{2n-2}b_{2n-2}b_{2n-2}a_{2n-2}$$

This makes it easy to prove by induction that $a_{2n}^r = a_{2n}$, and so a_{2n} is a palindrome (see Exercise 2.60). A similar argument holds for b_{2n}.

(d) It is easy to see that for every $n \geq 2$, a_n starts with 01 and ends with 10, and b_n starts with 10 and ends with 01. Using this fact, it is easy to show by induction that neither a_n nor b_n contains 000 or 111.

2.44. **Proof of induction step** (using ordinary induction). Suppose $f : A \to B$, A has $k+2$ elements, and B has $k+1$. Let a be any specific element of A. If there is some other

element $a_1 \in A$ with $f(a) = f(a_1)$, then obviously f is not one-to-one. If not, then we may consider the function $f_a : A - \{a\} \to B - \{f(a)\}$ defined by the formula $f_a(x) = f(x)$. By the induction hypothesis, f_a is not 1-1, and therefore f is not.

2.45. The first incorrect statement is "Now observe that C_{k-1} is an element of both T and R." If $k = 1$, there is no element called C_{k-1}. (The only thing wrong with the proof, in other words, is that it doesn't work when $k = 1$.)

2.46. Suppose p and q are distinct primes and n is divisible by both p and q. Then $n = pm$ for some m. Therefore, pm is divisible by q. Therefore, by the generally accepted fact, either p is divisible by q or m is divisible by q. However, p cannot be, because p, being prime, is divisible only by itself and 1, and q is not either of these. Therefore, m is divisible by q, and so pm is also.

2.47. This follows almost immediately from the previous exercise.

2.48. (The idea of the proof follows that of the proof in Example 2.3, except that we have to be a little more careful.) Suppose for the sake of contradiction that $\sqrt{n} = a/b$ for positive integers a and b. Then $b^2n = a^2$. We know n is not a perfect square; for reasons that will be clear in a minute, we want the number n not to be divisible by any perfect square bigger than 1. Of course, it may be, so let c be the biggest integer whose square divides n, and let $n = c^2m$. Then $b^2c^2m = a^2$. Rewrite this as $d^2m = a^2$. (This is the same as what we started with—i.e., $\sqrt{m} = a/d$—except we have a little more information about m.) Just as in the proof in Example 2.3, we can divide out factors common to the integers a and d, so that we may assume they have no common factors.

Now let p be any prime factor of m, so that $m = pm'$. Then p is a factor of a^2. Using the generally accepted fact in Exercise 2.46, p is a factor of a. Let $a = ep$. Then $a^2 = e^2p^2 = d^2m = d^2pm'$. Canceling p, we get $e^2p = d^2m'$. We know that p is not a factor of m', because $m = pm'$ is not divisible by p^2. Therefore, using the generally accepted fact again, p is a factor of d^2. Using it once more, p is a factor of d. Now we have a contradiction, however, because p is a factor of both a and d, and we assumed a and d had no common factors.

2.49. The induction step (using ordinary induction) follows from the inequality

$$\frac{2k\sqrt{k}}{3} + \sqrt{k+1} > \frac{2(k+1)\sqrt{k+1}}{3}$$

By subtracting $\frac{2\sqrt{k+1}}{3}$ from both sides and multiplying both sides by 3, we see that this is equivalent to

$$2k\sqrt{k} + \sqrt{k+1} > 2k\sqrt{k+1}$$

and the truth of this inequality can be seen easily by squaring both sides.

2.50. **Basis step.** Since $18 = 1 * 4 + 2 * 7$, we may choose $i_{18} = 1$ and $j_{18} = 2$.

Induction hypothesis. $k \geq 18$, and there exist integers i_k and j_k so that $k = i_k * 4 + j_k * 7$.
Statement to be shown in induction step. There exist integers i_{k+1} and j_{k+1} so that $k+1 = i_{k+1} * 4 + j_{k+1} * 7$.
Proof. We consider first the case in which $j_k \geq 1$. In this case we may write $k+1 = i_k * 4 + j_k * 7 + 1 = (i_k + 2) * 4 + (j_k - 1) * 7$, and we may therefore take $i_{k+1} = i_k + 2$ and $j_{k+1} = j_k - 1$. Otherwise, $k = i_k * 4$. We know $k \geq 18$; if $k \geq 20$, then $i_k \geq 5$, and this allows us to take $i_{k+1} = i_k - 5$ and $j_{k+1} = 3$, since $i_k * 4 + 1 = (i_k - 5) * 4 + 3 * 7$. We may therefore complete the proof by considering $k = 18$ and $k = 19$ separately. $18 + 1 = 3 * 4 + 1 * 7$, and $19 + 1 = 5 * 4 + 0 * 7$.

2.51. **Proof of induction step** (using ordinary induction). By the induction hypothesis, the number of subsets of $\{1, 2, \ldots, k\}$ having an even number of elements is 2^{k-1}. Since the total number of subsets is 2^k, the number having an odd number of elements is also 2^{k-1}. If $A \subseteq \{1, 2, \ldots, k+1\}$ and $|A|$ is even, then either $A = A_1 \cup \{k+1\}$, where A_1 is a subset of $\{1, 2, \ldots, k\}$ having an odd number of elements, or A is itself a subset of $\{1, 2, \ldots, k\}$ having an even number of elements. There are 2^{k-1} subsets of the first type and 2^{k-1} of the second type, and so there are 2^k altogether.

2.52. Suppose that the three given statements are true. For each $n \geq N$, let us consider the statement

$$Q(n) = P(N) \wedge P(N+1) \wedge P(N+2) \wedge \ldots \wedge P(n)$$

and try to show that $Q(n)$ is true for every $n \geq N$, using the ordinary principle of induction. It will then follow that $P(n)$ is true for every $n \geq N$.

The statement in the basis step is one of the three statements we are assuming is true. The induction hypothesis is the statement

$$k \geq N \text{ and } P(N) \wedge P(N+1) \wedge \ldots \wedge P(k) \text{ is true.}$$

We wish to show that

$$P(N) \wedge P(N+1) \wedge \ldots \wedge P(k+1) \text{ is true.}$$

But the statement that the first formula implies the second is also one of the three statements we are assuming.

2.53. (a) We use the strong principle of induction. In the induction step, we consider the case $k+1 = 1$ separately. If $k+1 > 1$, $f_{k+1} = f_k + f_{k-1} = c(a^k - b^k + a^{k-1} - b^{k-1}) = c(a^{k-1}(a+1) - b^{k-1}(b+1))$. It is easy to verify that $a + 1 = a^2$ and $b + 1 = b^2$, and this implies the result.

(b) In the induction step, we have

$$\begin{aligned} f_{k+2}^2 &= f_{k+1}f_{k+2} + f_k f_{k+2} \\ &= f_{k+1}(f_k + f_{k+1}) + f_k f_{k+2} \\ &= f_{k+1}^2 + f_k f_{k+2} + f_k f_{k+1} \end{aligned}$$

$$\begin{aligned} &= f_{k+1}^2 + (f_{k+1}^2 - (-1)^{k+1}) + f_k f_{k+1} \\ &= 2f_{k+1}^2 - (-1)^{k+1} + f_k f_{k+1} \\ &= f_{k+1}(2f_{k+1} + f_k) + (-1)^{k+2} \\ &= f_{k+1}(f_{k+1} + (f_k + f_{k+1})) + (-1)^{k+2} \\ &= f_{k+1}(f_{k+1} + f_{k+2}) + (-1)^{k+2} \\ &= f_{k+1}f_{k+3} + (-1)^{k+2} \end{aligned}$$

The fourth equality uses the induction hypothesis.

2.54. We first show that for any $n \geq 1$, n can be expressed as a sum of distinct powers of 2. This is true for $n = 1$, since $1 = 2^0$.
Induction hypothesis. $k \geq 1$, and any positive integer $\leq k$ is the sum of distinct powers of 2.
Statement to be shown in induction step. $k + 1$ is the sum of distinct powers of 2.
Proof. We know that $2^1 \leq k + 1$. Let i be the largest integer such that $2^i \leq k + 1$. If $2^i = k + 1$, then we have the result we want. Otherwise, the induction hypothesis applied to the positive integer $j = k + 1 - 2^i$ tells us that j is the sum of distinct powers of 2. Since $j < 2^i$ (because otherwise 2^{i+1} would be $\leq k+1$), none of these powers is as large as i, and therefore, $k + 1 = j + 2^i$ is the sum of distinct powers of 2.

For the second part, rather than using induction on n, we show that for any $m \geq 0$, if $\sum_{i=0}^{m} a_i * 2^i = \sum_{i=0}^{m} b^i * 2^i$ and each a_i is 0 or 1 and each b_i is 0 or 1, then $a_i = b_i$ for each i. The basis step is straightforward.
Induction hypothesis. $k \geq 0$, and if $\sum_{i=0}^{k} a_i * 2^i = \sum_{i=0}^{k} b^i * 2^i$ and each a_i is 0 or 1 and each b_i is 0 or 1, then $a_i = b_i$ for each i.
Statement to be shown in induction step. if $\sum_{i=0}^{k+1} a_i * 2^i = \sum_{i=0}^{k+1} b^i * 2^i$ and each a_i is 0 or 1 and each b_i is 0 or 1, then $a_i = b_i$ for each i.
Proof. We first show that $a_{k+1} = b_{k+1}$. If this is not true, assume for the sake of concreteness that $a_{k+1} = 0$ and $b_{k+1} = 1$. Then $\sum_{i=0}^{k} a_i 2^i \geq 2^{k+1}$. But this is impossible, because $\sum_{i=0}^{k} a_i 2^i \leq \sum_{i=0}^{k} 2^i = 2^{k+1} - 1$. A similar argument works if $a_{k+1} = 1$ and $b_{k+1} = 0$.

Now, since $a_{k+1} = b_{k+1}$ and $\sum_{i=0}^{k+1} a_i * 2^i = \sum_{i=0}^{k+1} b^i * 2^i$, it follows that $\sum_{i=0}^{k} a_i * 2^i = \sum_{i=0}^{k} b^i * 2^i$. By the induction hypothesis, $a_i = b_i$ for each i with $0 \leq i \leq k$. Therefore, $a_i = b_i$ for each i with $0 \leq i \leq k + 1$.

2.55. The basis step is obvious. Suppose the statement is true for $n = k$, and consider a sequence $a_1, a_2, \ldots, a_{k+1}$, and a permutation $b_1, b_2, \ldots, b_{k+1}$. We suppose, with no loss of generality, that a_{k+1} is the largest of the a_i's. If $b_{k+1} = a_{k+1}$, then the result we want follows easily from the induction hypothesis, since $b_1, \ldots, b_k$ is a permutation of $a_1, \ldots, a_k$. Otherwise, let $b_{k+1} = a_p$ and $a_{k+1} = b_q$.

$$(*) \quad \sum_{i=1}^{k+1} \frac{a_i}{b_i} - 1 = \left(\sum_{i=1}^{q-1} \frac{a_i}{b_i} + \frac{a_q}{a_p} + \sum_{i=q+1}^{k} \frac{a_i}{b_i}\right) + \left(\frac{a_{k+1}}{a_p} - \frac{a_q}{a_p} + \frac{a_q}{b_q} - 1\right)$$

In the right-hand expression of $(*)$, the denominators of the terms within the first pair of parentheses form a permutation of the numerators, because we have replaced the denominator b_q (i.e., a_{k+1}) by the one that appeared with a_{k+1} (i.e., a_p). The terms in the second pair of parentheses can be rewritten

$$\frac{a_{k+1} - a_q}{a_p} + \frac{a_q - a_{k+1}}{b_q} = (a_{k+1} - a_q)(\frac{1}{a_p} - \frac{1}{a_{k+1}})$$

By the induction hypothesis, the first parenthetical expression in $(*)$ is at least k, and since a_{k+1} is the largest a_i, the second is nonnegative. Therefore, the sum $\sum_{i=1}^{k+1}(a_i/b_i)$ is at least $k+1$. Furthermore, it equals $k+1$ only if the first parenthetical expression in $(*)$ is k and the second is 0, and by the induction hypothesis, this is true only if each of the a_i's is equal to the corresponding b_i.

2.56. (a) We can prove that for any $n \geq 0$, if the loop is iterated n times, then P is still true after the nth iteration. In the induction step, if the loop is iterated $k+1$ times, then B must still have been true after the kth iteration (because otherwise the loop would have terminated before iteration $k+1$). By the induction hypothesis, P is true after the kth iteration. Therefore, using the fact that P is a loop invariant, P is still true after iteration $k+1$.

(b) Let P be the condition $(r \geq 0) \wedge (x = q * y + r)$. Suppose this is true when $r = r_0$ and $q = q_0$, and that B is true (i.e., $r_0 \geq y$). Then the loop is iterated one more time. The new value of q is $q_0 + 1$, and the new value of r is $r_0 - y$. Since $r_0 \geq y$, $r = r_0 - y \geq 0$, and $x = q_0 * y + r_0 = (q_0 + 1) * y + (r_0 - y)$. Therefore, P is still true, and P is a loop invariant. Since the value of r is decreased by y in each iteration, and $y > 0$, the condition $r \geq y$ will eventually fail, so that the loop terminates. It follows from part (a) that P will be true, and the failure of the condition $r \geq y$ means that $0 \leq r < y$ will be true.

2.57. **Induction hypothesis.** $k \geq 1$, and for every n with $0 \leq n \leq k$, $f(n)$ is the largest power of 2 less than or equal to n.
Statement to be proved in induction step. $f(k+1)$ is the largest power of 2 less than or equal to $k+1$.
Proof. If $k+1$ is either $2j$ or $2j+1$, then $f(k+1) = 2f(j)$. By the induction hypothesis, $f(j)$ is the largest power of 2 less than or equal to j, say 2^p, which implies that $f(j) = 2^p \leq j < 2^{p+1}$. Therefore, $f(k+1) = 2 * 2^p = 2^{p+1}$. Since $j < 2^{p+1}$, both $2j$ and $2j+1$ are less than 2^{p+2}, so that no matter whether $k+1$ is even or odd, $f(k+1)$ is the largest power of 2 less than or equal to $k+1$.

2.58. **Proof of induction step** (using ordinary induction on k).

$$\begin{aligned} T(2^{k+1}) &\leq C(2^{k+1}) + 2T(2^{k+1}/2) \\ &\leq C(2^{k+1}) + 2(2^k * (Ck+1)) \end{aligned}$$

$$= 2^{k+1}(C + Ck + 1)$$
$$= 2^{k+1}(C(k+1) + 1)$$

2.59. (a) We use structural induction on the string y.
Basis step. For any string x, $(x\Lambda)^r = x^r = \Lambda x^r = \Lambda^r x^r$. Thus the statement holds when $y = \Lambda$.
Induction hypothesis. y is a string, and for any string x, $(xy)^r = y^r x^r$.
Statement to be shown in induction step. For any string x and any $a \in \Sigma$, $(x(ya))^r = (ya)^r x^r$.
Proof. $(x(ya))^r = ((xy)a)^r = a(xy)^r = a(y^r x^r) = (ay^r)x^r = (ya)^r x^r$. The first equality is because concatenation is associative, the second is the definition of the reverse function, the third is the induction hypothesis, the fourth is also the associativity of concatenation, and the fifth is the definition of the reverse function.

(b) **Proof of induction step** (using structural induction). $((xa)^r)^r = (ax^r)^r = (x^r)^r a^r = xa^r = x(\Lambda a)^r = x(a\Lambda^r) = x(a\Lambda) = xa$. The first equality uses the definition of the reverse function. The second uses the result in part (a).

(c) **Proof of induction step** (using ordinary induction on n). $(x^{k+1})^r = (x^k x)^r = x^r (x^k)^r = x^r (x^r)^k = (x^r)^{k+1}$. The second inequality uses the result in part (a), the third the induction hypothesis.

2.60. First, to show that if $x \in pal$, then $x^r = x$. This is by structural induction. Both Λ and a have this property, for any $a \in \Sigma$.

If x is an element of *pal* for which $x^r = x$, and $a \in \Sigma$, then $(axa)^r = a(ax)^r = a(x^r a^r) = axa$. The first equality uses the definition of reverse, the second uses the result in part (a) of Exercise 2.59.

Now we must show that if $x^r = x$, then $x \in pal$. In this proof, there appears to be no real advantage in using structural induction, and we use strong induction on $|x|$.
Basis step. $\Lambda \in pal$. This is true by definition of *pal*.
Induction hypothesis. $k \geq 0$, and for any x with $|x| \leq k$ satisfying $x^r = x$, $x \in pal$.
Statement to be shown in induction step. For any x with $|x| = k + 1$ satisfying $x^r = x$, $x \in pal$.
Proof. If $k + 1 = 1$, $x \in pal$ because of the definition. Assume, therefore, that $k + 1 \geq 2$. Then $x = ayb$ for some $a, b \in \Sigma$ and some y with $|y| = k - 1$. Because $x^r = x$, $x^r = (ayb)^r = ayb$. But $(ayb)^r = by^r a$, and so we may conclude that $a = b$ and $y^r = y$. It follows then from the induction hypothesis that $y \in pal$, and so $x = aya \in pal$ because of the definition of *pal*.

2.61. The proof that no element of L contains the substring aab is a straightforward structural induction proof based on the recursive definition of L. We show the converse, that every string not containing aab is an element of L, using strong induction on $|x|$. In the basis step, $\Lambda \in L$, by definition of L.
Induction hypothesis. $k \geq 0$, and any string x that satisfies $|x| \leq k$ and does not contain the substring aab is in L.

Statement to be shown in induction step. If $|x| = k + 1$ and x does not contain aab, then $x \in L$.
Proof. In the case where $x = ya$ for some y, since x does not contain aab, y cannot. By the induction hypothesis, $y \in L$. Therefore, according to the definition of L, $x = ya \in L$. Similarly in the cases where $x = by$ and $x = aby$. To complete the proof, therefore, it is sufficient to show that one of these cases must hold. If not, then $x = aayb$ for some y. In this case, however, the first b in the string x must be preceded by aa, which is impossible.

2.62. (a) The set of strings not containing the substring aaa.
(b) The set of strings not containing the substring baa.

2.63. (a) The proof is by structural induction, using the recursive definition of L_1. The string Λ is in L_2. If x is any string in L_1 that is known to be in L_2, and $y \in L$, then $y \in L_2$ by the second statement in the definition of L_2, and so $xy \in L_2$ by the third statement in the definition of L_2.

(b) We show the following statement $P(y)$ for every $y \in L_1$: for every $x \in L_1$, $xy \in L_1$. The proof is by structural induction using the definition of L_1. $P(\Lambda)$ is clearly true. If $P(y)$ is true for some $y \in L_1$, then we must show that $P(yz)$ is true for every $z \in L$. $P(yz)$ is the statement: for every $x \in L_1$, $x(yz) \in L_1$. However, $x(yz) = (xy)z$; $xy \in L_1$ by the induction hypothesis, and so $(xy)z \in L_1$ according to the definition of L_1.

(c) The proof is by structural induction using the definition of L_2.
Basis step. To show that $\Lambda \in L_1$ and any element of L is in L_1. The first statement is true by definition of L_1, and the second follows from the second statement in the definition of L_1, using $x = \Lambda$.
Induction hypothesis. x and y are elements of L_2 that are in L_1.
Induction step. To show that $xy \in L_1$. Since x and y are in L_1 by the induction hypothesis, this follows from part (b).

2.64. **Basis step.** If $|x| = 1$ and x has more a's than b's, then $x \in L$. This is true because x must be the string a, which is in L by definition.
Induction hypothesis. $k \geq 1$, and if x is any string with more a's than b's for which $|x| \leq k$, then $x \in L$.
Statement to be shown in induction step. If $|x| = k + 1$ and x has more a's than b's, then $x \in L$.
Proof. If x contains no b's, then $x = ay$, where $|y| = k$ and y also contains more a's than b's. By the induction hypothesis, $y \in L$, and so by part 2 of the definition, $x = ay \in L$. The three remaining cases we consider are those in which x starts with b, x ends with b, and x starts and ends with a but contains at least one b.

Let $d(z) = n_a(z) - n_b(z)$, the difference between the number of a's and the number of b's in the string z. If x starts with b, then $x = by$ for some y satisfying $d(y) \geq 2$. Consider prefixes of y: the shortest, Λ, satisfies $d(\Lambda) = 0$; the longest, y, satisfies $d(y) \geq 2$; since adding 1 to the length of the prefix changes the value of d by 1, there must be some prefix y_1 for which $d(y_1) = 1$. Therefore, if $y = y_1y_2$, we have $d(y_2) \geq 1$. By the induction hypothesis, both y_1 and y_2 are in L. Therefore, the string $x = by_1y_2$ is also in L.

The proof in the case when x ends with b is similar. Suppose $x = aya$, where y contains m b's and $m \geq 1$. If x_1 is the portion of x preceding the last b, and $d(x_1) > 0$, then $x = x_1 b x_2$ where $d(x_1)$ and $d(x_2)$ are both positive; the induction hypothesis implies that x_1 and x_2 are in L, and it follows from the definition of L that x is. Otherwise, $d(x_1) \leq 0$. In this case, for each i with $1 \leq i \leq m$, let w_i be the prefix of x preceding the ith b and z_i the suffix following the ith b. Then $d(w_m) \leq 0$; let j be the *smallest* i for which $d(w_i) \leq 0$. Since x starts with a, j must be at least 2. We know, therefore, that $d(w_{j-1}) > 0$ and $d(w_j) \leq 0$. However, w_j has only one more b than w_{j-1}, which implies that $d(w_{j-1}) = 1$. Now we have $x = w_{j-1} b z_{j-1}$, where $d(x) > 0$ and $d(w_{j-1}) = 1$. It follows that $d(z_{j-1})$ is also positive. Now we can apply the induction hypothesis to w_{j-1} and z_{j-1}; they are both in L, and so the string x is also, by the definition of L.

2.65. The proof that every element of L has equal numbers of a's and b's is a straightforward structural induction proof. The converse is proved by strong induction on $|x|$.
Basis step. $\Lambda \in L$. This is true by definition of L.
Induction hypothesis. $k \geq 0$, and any string x with $|x| \leq k$ having equal numbers of a's and b's is in L.
Statement to be shown in induction step. If x has equal numbers of a's and b's, and $|x| = k + 1$, then $x \in L$.
Proof. If x is either $01y$ or $10y$, for some y with $|y| = k - 1$, then y has equal numbers of a's and b's, and is in L by the induction hypothesis. Therefore, x, either $0\Lambda 1y$ or $1\Lambda 0y$, is in L by definition of L.

Otherwise x starts with either 00 or 11. We complete the proof in the first case, and the second case is similar. Let $d(z) = n_0(z) - n_1(z)$, and consider $d(z)$ for the prefixes z of x. $d(00) = 2$ and $d(x) = 0$. Since adding a single symbol to a prefix changes the d value by 1, there must exist a prefix z longer than 00 for which $d(z) = 1$. Let z_1 be the *longest* such prefix. Then x must be $z_1 1 z_2$ for some string z_2—otherwise, since $d(z_1 0) = 2$, there would be a longer prefix for which $d = 1$. We now know that $x = 00y_1 1z_2$, and $d(0y_1) = d(00y_1 1) = 0$. Therefore, $d(z_2) = 0$ as well. By the induction hypothesis, both $0y_1$ and z_2 are in L. Therefore, $x = 0(0y_1)1z_2$ can be also, because of the definition of L.

2.66. We show that for $n \geq 0$ and elements $x_1, \ldots, x_n$ of S, if $e^*(x_1 \ldots x_n) = e^*(y)$ for some $y \in S^*$, then $y = x_1 x_2 \ldots x_n$. (Note: although the problem doesn't say explicitly that $e^*(\Lambda) = \Lambda$, this is the correct interpretation of what $x_1 x_2 \ldots x_n$ and $e(x_1) \ldots e(x_n)$ mean when $n = 0$.)
Basis step. We must show that if $e^*(\Lambda) = \Lambda = e^*(y)$ for some $y \in S^*$, then $y = \Lambda$. This is true because $\Lambda \notin T$.
Induction hypothesis. $k \geq 0$, and for any $x_1, \ldots, x_k$ in S, if $e^*(x_1 \ldots x_k) = e^*(y)$ for some $y \in S^*$, then $x_1 \ldots x_k = y$.
Statement to be shown in induction step. For any $x_1, \ldots, x_{k+1}$ in S, if $e^*(x_1 x_2 \ldots x_{k+1}) = e^*(y)$ for some $y \in S^*$, then $x_1 x_2 \ldots x_{k+1} = y$.
Proof. Suppose $e(x_1)e(x_2) \ldots e(x_{k+1}) = e(y_1)e(y_2) \ldots e(y_m)$. Then $m \geq 1$. Now if the two strings $e(x_1)$ and $e(y_1)$ are not equal, one must be a prefix of the other, and this is

impossible; therefore, $e(x_1) = e(y_1)$. Now we have $e(x_2) \ldots e(x_{k+1}) = e(y_2) \ldots e(y_m)$. The induction hypothesis now implies that $x_2 \ldots x_{k+1} = y_2 \ldots y_m$, and this implies the result.

2.67. Nothing happens until the nth day; on that day all n wives of unfaithful husbands realize that their husbands are unfaithful, and proceed to kill them. To show this, we may let $P(n)$ be the statement that if there are at least n unfaithful husbands, no one has been killed by the end of the $n-1$th day, and if there are exactly n, then all n are killed on the nth day. Suppose that this is true when $n = k$, and now suppose $n = k+1$. Consider the wife of one of one of the unfaithful husbands. She knows that there are either k unfaithful husbands (if her husband is faithful) or $k+1$ (if he is not). Furthermore, she realizes that if her husband is faithful, the k wives of unfaithful husbands will all kill their husbands on the kth day, and if he is not, they won't. At midnight of the kth day, however, no one has been killed, and she concludes that her husband must be unfaithful.

2.68. Yes. Let $S(n)$ be the statement: for every $m \geq 0$, $P(m, n)$ is true. We can prove that $S(n)$ is true for every $n \geq 0$, by induction.
Basis step. We must show that for every $m \geq 0$, $P(m, 0)$ is true. This is itself an induction proof. The basis step uses the fact that $P(0, 0)$ is true, and the induction step uses the fact that if $P(k, 0)$ is true, then $P(k+1, 0)$ is true.
Induction hypothesis. $k \geq 0$, and for every $m \geq 0$, $P(m, k)$ is true.
Statement to be shown in induction step. For every $m \geq 0$, $P(m, k+1)$ is true.
Proof. This follows immediately from the assumption on P: if $P(m, k)$ is true, $P(m, k+1)$ is true.

2.69. Let us say that x satisfies property p if $x = 2$ or x is divisible by 5. Then $\{x \in S \mid x$ satisfies property $P\}$ is the set of all multiples of 5. However, $\{x \in \mathcal{N} \mid x$ has property $P\}$ is the set containing 2 and all multiples of 5. This second set is not closed under the operation of adding 5. Therefore, the proof using structural induction that every element of S satisfies property P could not be simplified as in the two examples.

2.70. Assume for the sake of this discussion that $\circ$ is a binary operation. Consider the algorithm described by this pseudocode.

```
A = I;
Repeat
   B = A;
   for each element x of A
      for each element y of A
         B = B ∪ {x ∘ y}
until B = A;
Determine whether the desired element is in A
```

The algorithm terminates because U is finite: there can be only a finite number of iterations of the Repeat loop in which one or more elements are added to A. The final value of A is S, because if T is any subset of U that contains all the elements of I and is closed under

o, any element added to A in any of the iterations must belong to T.

Chapter 3

Regular Expressions and Finite Automata

3.1. (a) 001 or 011 (b) 0101 or 1010 (c) 110 (d) 0110

3.2. (a) 00 (b) 01 (c) 0 (d) The shortest one is 010.

3.3. (a) $(r+s)^*$ (b) $r(r+s)^*$ (c) r^* (d) r^* (e) $(r+s)^*$

3.4. First observe that the only difference between $(111^*)^*$ and 111^* is that the first allows the string Λ. In other words, strings corresponding to $(111^*)^*$ are Λ and all strings of two or more 1's. Therefore, the formula follows from the fact that any string of two or more 1's can be formed by concatenating copies of 11 and/or 111—i.e., that any integer $n \geq 2$ can be expressed as $2i + 3j$ for some $i, j \geq 0$. The proof of this fact is similar to the argument in Exercise 2.50.

3.5. The string Λ corresponds to both expressions, and it is easy to see that any nonnull string corresponding to the first one must start with a and end in b. Therefore, it is sufficient to show that every string of the form $x = ayb$ corresponds to the first regular expression. Let us show this by induction on the number of a's in x that are not immediately preceded by a. If there is only one such a, then x does not contain the substring ba, and it must therefore match the regular expression aa^*bb^*. Suppose $k \geq 1$, and every string of the form ayb having k a's not immediately preceded by a matches the first regular expression. Now let $x = ayb$, and suppose x contains $k+1$ a's that are not immediately preceded by a. The first is the one at the beginning of x; consider the second such a, and let x_1 be the suffix of x beginning with this a. Then the prefix of x preceding x_1 must be of the form aa^*bb^*. The induction hypothesis tells us that x_1 matches the regular expression $(aa^*bb^*)^*$, and so the entire string x must match it also.

3.6. $\emptyset^* = \{\Lambda\}$.

3.7. (a) The set of languages that are subsets of $\Sigma \cup \{\Lambda\}$. This can also be described as the set of languages L over Σ for which every element of L is a string of length 0 or 1. If $|\Sigma| = k$, there are 2^{k+1} such languages.
(b) The set of all languages over Σ which either are empty or contain exactly one string. There are infinitely many such languages.
(c) The set of all languages that are $\emptyset$ or $\{\Lambda\}$ or of the form $\{a\}^*$, where $a \in \Sigma$. There are exactly $|\Sigma| + 2$ such languages.
(d) The set of all finite languages over Σ, of which there are infinitely many.
(e) The (finite) set of all languages of the form

$$\Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_k \cup \Sigma_{k+1}^* \cup \Sigma_{k+2}^* \cup \ldots \Sigma_{k+j}^*$$

where each Σ_i is a subset of $\Sigma \cup \{\Lambda\}$. (k and/or j may be 0; if both are 0, the language is $\emptyset$.) For $\Sigma = \{a, b\}$, this includes the languages $\emptyset$, $\{\Lambda\}$, $\{a\}$, $\{b\}$, $\{a, b\}$, $\{a, \Lambda\}$, $\{b, \Lambda\}$, $\{a, b, \Lambda\}$, $\{a\}^*$, $\{b\}^*$, $\{a\}^* \cup \{b\}^*$, $\{a, b\}^*$, $\{a\}^* \cup \{b\}$, and $\{b\}^* \cup \{a\}$.

3.8. (a) $(001)^*(11)^*$ (b) $(001)^*0(001 + 11)^*$ (c) $(001 + 11)^*(0 + \Lambda)$

3.9. (a) $1^*01^*01^*$
(b) The most obvious solutions are those of the form $A0B0C$, where each of A, B, C is either 1^* or $(0 + 1)^*$, and at least one of the three is $(0 + 1)^*$.
(c) $\Lambda + 1 + (0 + 1)^*0 + (0 + 1)^*11$
(d) $(00 + 11)(0 + 1)^* + (0 + 1)^*(00 + 11)$
(e) Two answers are $(1 + 01)^*(\Lambda + 0)$ and $(\Lambda + 0)(1 + 10)^*$.
(f) $1^*(01^*01^*)^*$
(g) The regular expression $r = (1 + 01)^*$ corresponds to the set of strings that don't end with 0 and don't contain 00, and $s = (1 + 10)^*$ to the set of strings that don't begin with 0 and don't contain 00. In a string with exactly one occurrence of 00, then the strings before and after 00 correspond to r and s, respectively. Therefore, one answer is $(1+01)^*(0+\Lambda)+(1+01)^*00(1+10)^*$. A more concise answer is $(1+01)^*(\Lambda+0+00)(1+10)^*$.
(h) $1^*(011^+)^*$
(i) $(0 + 1)^*(11(0 + 1)^*010 + 010(0 + 1)^*11)(0 + 1)^*$

3.10. (a) All strings with an odd number of 1's.
(b) All strings whose length is a multiple of 3, or 1 plus a multiple of 3.
(c) All strings not containing any substring of the form $00x11$.
(d) All strings containing both 10 and 01.

3.13. (b) We may define *rrev* recursively on the set of regular expressions as follows:

$$rrev(\emptyset) = \emptyset; \qquad rrev(\Lambda) = \Lambda; \qquad \text{for } a \in \Sigma,\ rrev(a) = a$$

and for arbitrary regular expressions r and s,

$$rrev((r+s)) = (rrev(r)+rrev(s)); \quad rrev((rs)) = (rrev(s)rrev(r)); \quad rrev((r^*)) = ((rrev(r))^*)$$

Now we can show by structural induction that for any regular expression r, if $L(r)$ is the corresponding language, $rrev(r)$ is the regular expression corresponding to $rev(L(r))$. This is clearly true for the regular expressions $\emptyset$, Λ, and a, for $a \in \Sigma$. Suppose it is true for the regular expressions r and s. We show it is true for (rs), and the argument is similar for the other two cases. $rrev((rs)) = (rrev(s)rrev(r))$, by definition of *rrev*, and by Exercise 2.59a, $rev(L(r)L(s)) = rev(L(s))rev(L(r))$. According to the induction hypothesis, $rrev(s)$ corresponds to the language $rev(L(s))$, and similarly for r. Since the language corresponding to the concatenation of two regular expressions is the concatenation of the two languages, we have the desired result in this case.

3.14. One expression is

$$(\Lambda + a + m)(d^+ + d^*pd^+ + d^+pd^*)(\Lambda + (E + e)(\Lambda + a + m)d^+)$$

3.15. (a) 2 (b) 3

3.16. (a) $\Lambda + aaa^*$ (b) $\Lambda + aaaa^*$

3.17. (a) The strings corresponding to each state are as follows. (It's a little easier to start from the end.)

V. All strings containing $aaba$.
IV. All strings ending in aab but not containing $aaba$.
III. All strings ending in aa but not containing $aaba$.
II. All strings ending in a but not ending in aa and not containing $aaba$.
I. All strings not ending in a or aab and not containing $aaba$.

(b)

V. All strings ending with $aaba$.
IV. All strings ending with aab.
III. All strings ending with aa.
II. All strings ending with a but not with either aa or $aaba$.
I. All strings ending with neither a nor aab.

(c)

V. All strings beginning with $aaba$.
IV. Only the string aab.
III. Only the string aa.
II. Only the string a.
I. Only the string Λ.
VI. All strings that don't begin with $aaba$, except for the strings Λ, a, aa, and aab.

(d)

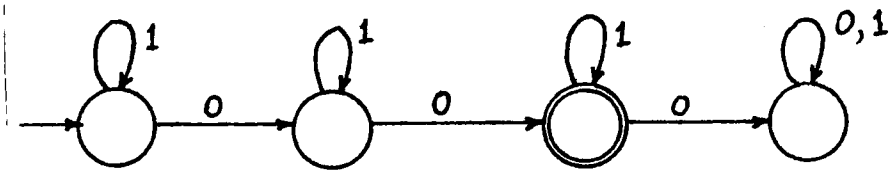
I. Λ only.
II. All strings that start and end with a.
III. All strings that start with a and end with b.
IV. All strings that start with b.

(d)

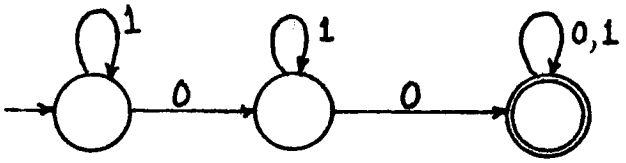
I. $(ab + ba)^*$
II. $(ab + ba)^*a$
III. $(ab + ba)^*b$
IV. $(ab + ba)^*(aa + bb)(a + b)^*$

3.18. If $|x| = n$, there is an FA with $|x| + 2$ states accepting $\{x\}$. It has one state for each of the $n + 1$ prefixes of x, and one state N representing all the strings that are nonprefixes. For each state representing a prefix of x other than x itself, there is one transition to the next longer prefix and one to N; from the states x and N, both transitions go to N.

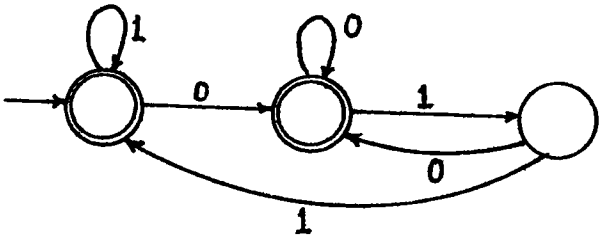
3.19 (a)



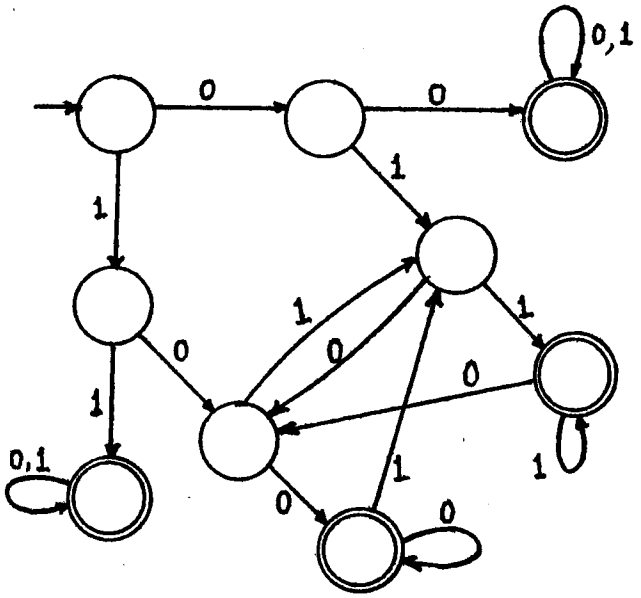
(b)



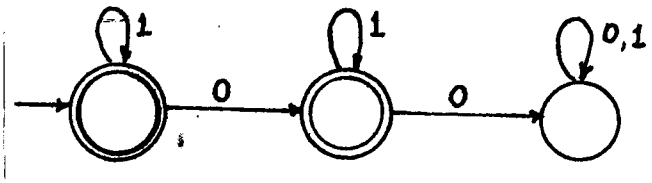
(c)



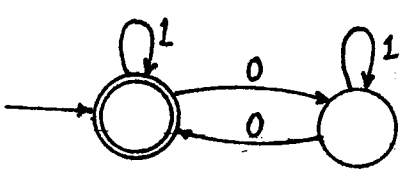
(d)



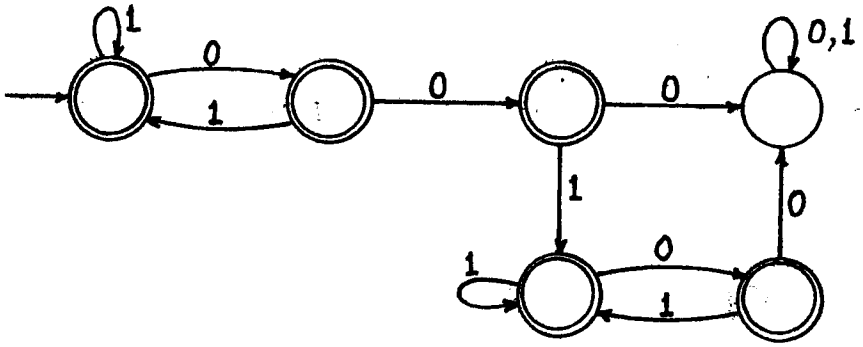
3.19 (e)



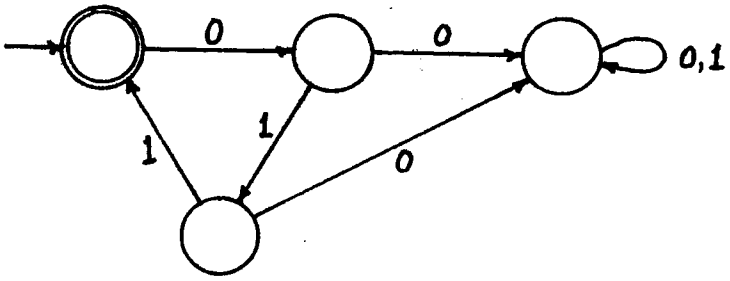
(f)



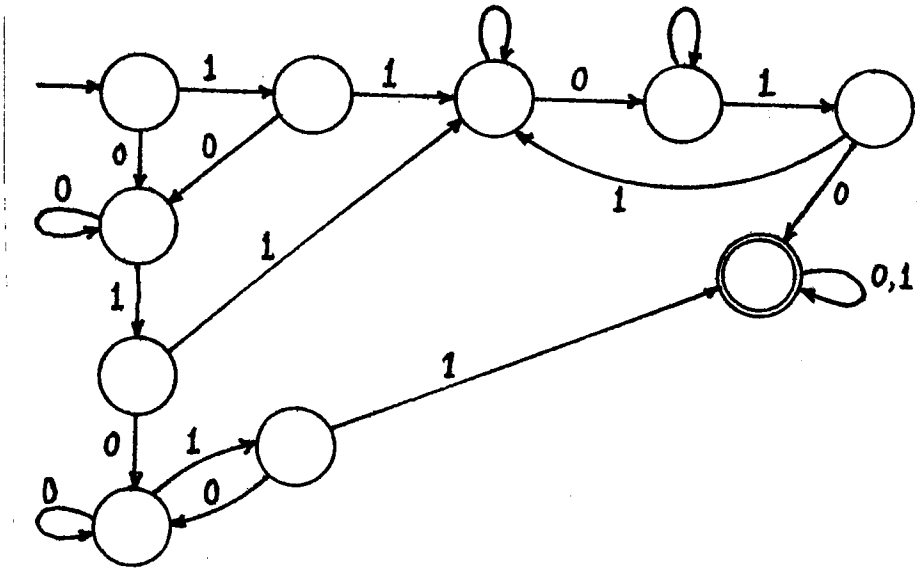
(g)



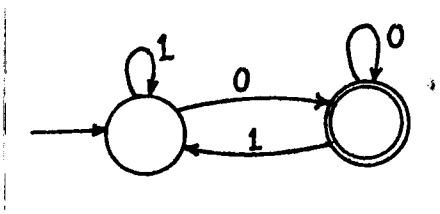
(h)



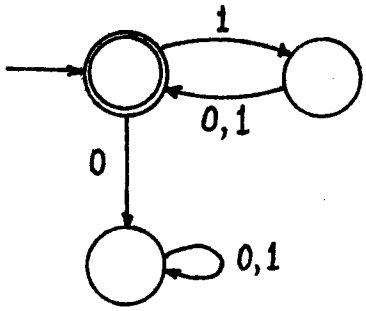
(i)



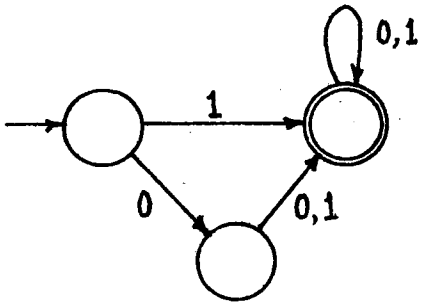
3.20 (a)



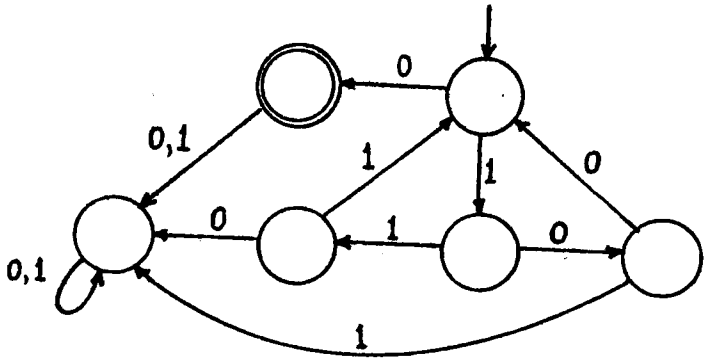
(b)



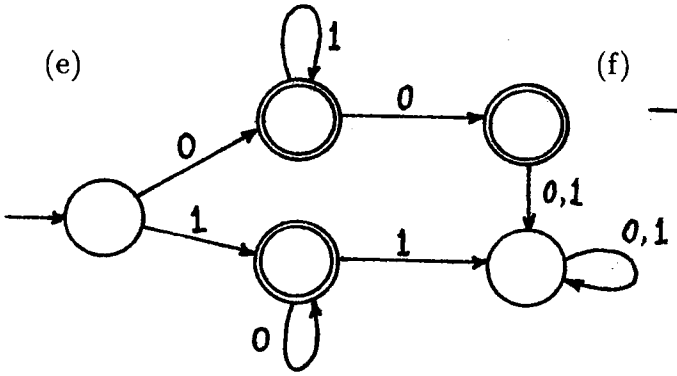
(c)



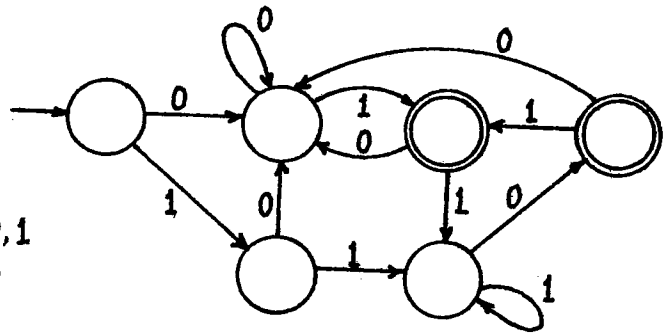
(d)



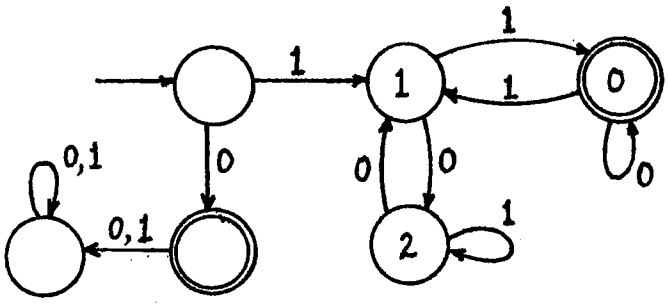
(e)



(f)



3.21. The numbered states correspond to remainders mod 3. Appending 0 or 1 to the string representing n yields $2n$ or $2n + 1$, respectively. The unnumbered accepting state represents the integer 0.



3.22. (a) We use structural induction on y. The basis step is to show that for any x and any q, $\delta^*(q, x\Lambda) = \delta^*(\delta^*(q, x), \Lambda)$. This is true because $x\Lambda = x$ and $\delta^*(p, \Lambda) = p$ for any p (in particular, for $p = \delta^*(q, x)$). Suppose y has the property that for any x and any q, $\delta^*(q, xy) = \delta^*(\delta^*(q, x), y)$. Now consider ya, for any $a \in \Sigma$.

$$\delta^*(q, x(ya)) = \delta^*(q, (xy)a) = \delta(\delta^*(q, xy), a) = \delta(\delta^*(\delta^*(q, x), y), a) = \delta^*(\delta^*(q, x), ya)$$

The second equality uses the definition of δ^*; the next one uses the induction hypothesis; and the last one uses the definition of $\delta^*(p, ya)$, where $p = \delta^*(q, x)$.

(b) We use structural induction on x. In the basis step, $\delta^*(q, \Lambda) = q$ by definition of δ^*. In the induction step, suppose $\delta^*(q, x) = q$, and let $a \in \Sigma$. Then $\delta^*(q, xa) = \delta(\delta^*(q, x), a) = \delta(q, a) = q$.

(c) In the induction step, we consider $\delta^*(q, x^{k+1}) = \delta^*(q, x^k x) = \delta^*(\delta^*(q, x^k), x) = \delta^*(q, x) = q$. The second equality uses the formula in part (a), and the next equality uses the induction hypothesis.

3.23. For any FA, a trivial way to get another FA with one more state accepting the same language is to add a state, allow no transitions to that state, and make all the transitions from that state go to one of the states in the original FA.

3.24. The language $L = \{\Lambda, 0\} \subseteq \{0, 1\}^*$ has this property, as does any language for which there are two strings in the language that are distinguishable with respect to the language. (A special case is that in which some strings in L are prefixes of other strings in L and some strings in L are not; however, more than one accepting state may be required even if this condition does not hold.)

3.25. The language accepted by the FA is the set L of strings that don't contain the substring 00 and don't end in 01. The simplest strings corresponding to the four states are Λ, 0, 01, and 00. It is possible to show that any two of these strings are distinguishable with respect to L. For example, Λ and 0 are distinguished by the string 0; Λ and 01 are distinguished by the string Λ, as are 0 and 01, and 0 and 00, and Λ and 00; and 00 and 01 are distinguished by the string 0.

3.26. $n + 1$ states are required. There are $n + 1$ prefixes of z, whose lengths vary from 0 to n, and any two of these can be distinguished with respect to $L = \{0, 1\}^*\{z\}$. (If $x_1y_1 = x_2y_2 = z$ and $|x_1| < |x_2|$, then $x_1y_2 \notin L$ and $x_2y_2 \in L$.) No more states are necessary, because for any string x, if $x = x_1y$ and y is a prefix of z and x does not end with any longer prefix of z, then x is indistinguishable from y with respect to L. (Note: every x ends with a prefix of z, if only the prefix Λ.)

3.27. No. A simple example is provided by the language L of all even-length strings of 0's and the sequence $x_0, x_1, \ldots$, where $x_i = 0^i$. For any n, x_n and x_{n_1} are distinguished with respect to L by the string Λ. (One of the two is in L and the other is not.)

3.28. If L_n is any nonempty language such that every $x \in L_n$ has length n, then an FA accepting L_n must have at least $|n|+2$ states, because for any $x \in L_n$, a set of $n+2$ strings containing each of the $n+1$ prefixes of x as well as a string y of length $n+1$ is pairwise distinguishable with respect to L_n. (If $x_1y_1 = x_2y_2 = x$, and $|x_1| < |x_2|$, then x_1 and x_2 are distinguished by the string y_1. Furthermore, any of the the prefixes is distinguishable from y with respect to L_n.) The language $L = \{0,1\}^*$ is one example. Any infinite language that does not contain two strings of the same length is another.

3.29. If $x \in L(M)$, then for some accepting state q, $\delta^*(q_0, x) = q$. For each prefix x_1 of x, the state $\delta^*(q_0, x_1)$ is obviously reachable from q_0, and therefore still present in M_1. Therefore, $\delta_1^*(q_0, x)$ is still q, and $x \in L(M_1)$. If $x \in L(M_1)$, since all the transitions of M_1 are present in M, x is also in $L(M)$.

3.30. Suppose $\delta^*(q_0, x) = q \in R$. By definition of R, there is a string y so that $\delta^*(q, y) \in A$. Therefore, $\delta^*(q_0, xy) = \delta^*(\delta^*(q_0, x), y) \in A$, and x is a prefix of an element of $L(M)$. On the other hand, if x is a prefix of an element of $L(M)$, then $\delta^*(q_0, xy) \in A$, for some string y, so that $\delta^*(q_0, x) \in R$. The conclusion is that M_1 accepts the language of all strings that are prefixes of elements of $L(M)$.

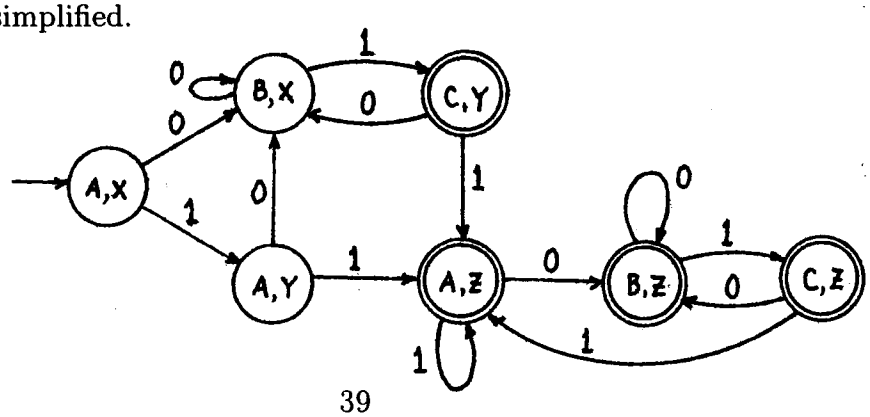
3.31. Every string is a prefix of an element of $L(M)$. (See the preceding exercise.)

3.32. The proof is by structural induction on x. $\delta^*((p,q), \Lambda) = (p,q) = (\delta_1^*(p, \Lambda), \delta_2^*(q, \Lambda))$, from the definitions of δ^*, δ_1^*, and δ_2^*. Suppose x is a string for which $\delta^*((p,q), x) = (\delta_1^*(p,x), \delta_2^*(q,x))$, and let $a \in \Sigma$. Then

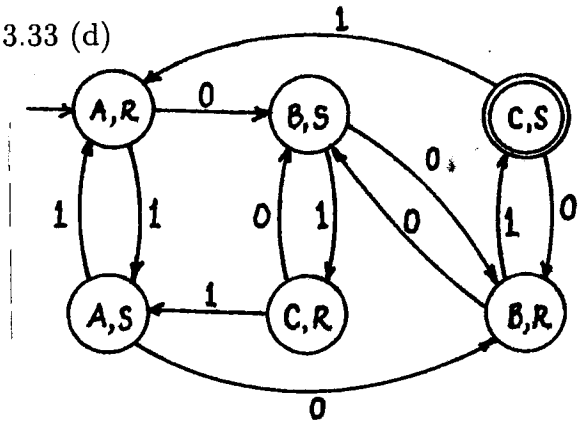
$$\begin{aligned}
\delta^*((p,q), xa) &= \delta(\delta^*((p,q), x), a) \\
&= \delta((\delta_1^*(p,x), \delta_2^*(q,x)), a) \\
&= (\delta_1(\delta_1^*(p,x), a), \delta_2(\delta_2^*(q,x), a)) \\
&= (\delta_1^*(p, xa), \delta_2^*(q, xa))
\end{aligned}$$

The first inequality is true by definition of δ^*; the second is true by the induction hypothesis; the third is true by the definitions of δ; and the last is true by the definitions of δ_1^* and δ_2^*.

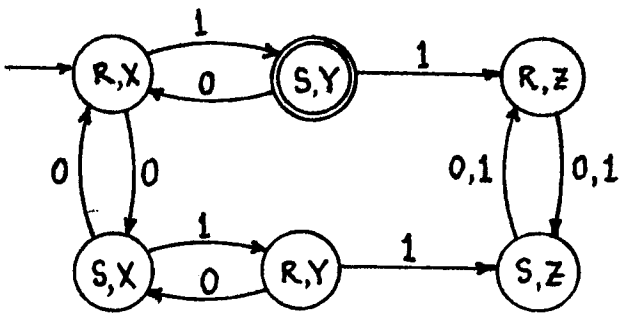
3.33 (a) The picture below is also the one for parts (b) and (c), except that in (b) the only accepting state is (C, Z) and in (c) the only accepting state is (C, Y). In both (a) and (c), the picture can be simplified.



3.33 (d)



(e)



3.34. On the one hand, any string matching $r + s$ matches r^*s^*, and therefore any string matching $(r + s)^*$ matches $(r^*s^*)^*$. On the other hand, if x is a string matching r^*s^*, then $x = x_1x_2$, where x_1 matches r^* and x_2 matches s^*. Then x_1 matches $(r + s)^*$ and so does x_2. Therefore, $x = x_1x_2$ also matches $(r + s)^*$.

3.35. We first show that for any $n \geq 0$, if x corresponds to the regular expression $(00^*1)^n1$, then x corresponds to the regular expression $1 + 0(0 + 10)^*11$. The proof for $n = 0$ is easy. Suppose that $k \geq 0$ and any string corresponding to $(00^*1)^k1$ corresponds to $1+0(0+10)^*11$. We must show that if x corresponds to $(00^*1)^{k+1}1$, then x corresponds to $1 + 0(0 + 10)^*11$. We know that $x = 00^j1y$, where y corresponds to $(00^*1)^k1$. By the induction hypothesis, y corresponds to $1 + 0(0 + 10)^*11$. If $y = 1$ the result is clear. Otherwise $y = 0z11$, where z corresponds to $(0 + 10)^*$. In this case $x = 00^j10z11$. Since 0^j10 corresponds to $(0 + 10)^*$, 0^j10z does also, and this implies the result.

Now we show the opposite direction, that if x corresponds to $1 + 0(0 + 10)^n11$, then x corresponds to $(00^*1)^*1$. Again the statement is clear when $n = 0$. Suppose $k \geq 0$ and any string corresponding to $1 + 0(0 + 10)^k11$ corresponds to $(00^*1)^*1$. Let x be a string corresponding to $1 + 0(0 + 10)^{k+1}11$. If $x = 1$ then x clearly corresponds to $(00^*1)^*1$. Otherwise, $x = 0z11$, where z corresponds to $(0 + 10)^{k+1}$. This implies that for some z' corresponding to $(0+10)^k$, either $x = 00z'11$ or $x = 010z'11$. By the induction hypothesis, $0z'11$ corresponds to $(00^*1)^*1$, and therefore to $(00^*1)^j1$ for some j. If $j = 1$, then $z' = 0^i$ for some i, and it is easy to see in this case that both $00z'11$ and $010z'11$ correspond to $(00^*1)^*1$, so that x is of the desired form. If $j \geq 2$ then $z' = 0^p1z''00^q$, where z'' corresponds to $(00^*1)^*$; in this case $00z'11 = 0^{p+2}1z''0^{q+1}11$, and $010z'11 = 010^{p+1}1z''0^{q+1}11$, so that again x has the right form.

3.36. (a) $(\Lambda + 0 + 00)(1 + 10 + 100)^*$

(b) Saying that a string doesn't contain 110 means that if 11 appears, then 0 cannot appear anywhere later. Therefore, a string not containing 110 consists of an initial portion not containing 11, possibly followed by a string of 1's. By including in the string of 1's *all* the trailing 1's, we may require that the initial portion doesn't end with 1. Since $(0 + 10)^*$ corresponds to strings not containing 11 and not ending with 1, one answer is $(0 + 10)^*1^*$.

(c) $(0 + 1)^*(101(0 + 1)^*010 + 010(0 + 1)^*101 + 1010 + 0101)(0 + 1)^*$

3.36 (d) $(00 + 11 + (01 + 10)(00 + 11)^*(01 + 10))^*$
(e) $e(01 + 10)e$, where e is the expression in (g).

3.37. We say a regular expression r over Σ has property P if, when each alphabet symbol of r is replaced by another regular expression over Σ, the result is also a regular expression over Σ. We use structural induction to show that every regular expression over Σ has property P.

The basis step is to show that the regular expressions $\emptyset$, Λ, and a have property P, for every $a \in \Sigma$. This is clear, because the only one of these that is changed by substituting regular expressions for elements of Σ is a, and in that case the result is a regular expression.
Induction hypothesis. The regular expressions r_1 and r_2 over Σ have property P.
Statement to be shown in induction step. The regular expressions $(r_1 + r_2)$, $(r_1 r_2)$, and (r_1^*) all have property P.
Proof. We take care of the first case, and the argument in the other two is similar. When a regular expression is substituted for each element of Σ in $(r_1 + r_2)$, the result is $(s_1 + s_2)$, where each s_i is obtained from the corresponding r_i by substituting regular expressions for alphabet symbols. By the induction hypothesis, each s_i is a regular expression. It follows by definition that $(s_1 + s_2)$ is also.

3.38. (b) Suppose r satisfies $r = c + rd$ and Λ does not correspond to d. Let x be a string corresponding to r. To show that x corresponds to cd^*, assume for the sake of contradiction that x does not correspond to cd^j for any j. We can then show using mathematical induction that for *every* $n \geq 0$, x corresponds to the regular expression rd^n.

The basis step $n = 0$ is clear, since x corresponds to r. Suppose that $k \geq 0$ and x corresponds to rd^k. $rd^k = (c + rd)d^k = cd^k + rd^{k+1}$; since x does not correspond to cd^k, it must correspond to rd^{k+1}.

Now it remains to derive a contradiction. Since Λ does not correspond to d, every string corresponding to d has length at least 1. It follows that every string corresponding to d^n must have length at least n, and therefore that x must have length at least n for every n. This is clearly impossible. We conclude that x must correspond to cd^j for some j.

3.39. Make a sequence of passes through the expression. In each pass, replace any regular expression of the form $(\emptyset + r)$ or $(r + \emptyset)$ by r (where r is any regular expression), any regular expression of the form $(\emptyset r)$ or $(r\emptyset)$ by $\emptyset$, and any occurrence of $(\emptyset^*)$ by Λ. Stop after any pass in which no changes are made. If $\emptyset$ remains in the expression, then the expression must actually be $\emptyset$, in which case the corresponding language is empty.

3.40. Make a sequence of passes through the expression. In each pass, replace any regular expression of the form (Λr) or $(r\Lambda)$ by r (where r is any regular expression); replace any occurrence of Λ^* by Λ; replace any regular expression of the form $(r + \Lambda)^*$ by (r^*) (where r is any regular expression); and replace any regular expression of the form $(\Lambda + r)s$ by $s + rs$ (where r and s are any regular expressions). Stop after any pass in which no changes are made. If Λ remains in the expression, then the expression corresponds to the language $\{\Lambda\}$.

3.41. (a) It is clear that if $L^k = L^*$, then $L^k = L^{k+1}$. On the other hand, suppose that $L^k = L^{k+1}$. Let m be the length of the shortest element of L. Then the shortest elements in L^k and L^{k+1} have lengths km and $(k+1)m$, respectively, which implies that m must be 0. Therefore, $\Lambda \in L$. It follows that $L^i \subseteq L^{i+1}$ for every i, and therefore that $L^* = \cup_{i=0}^{\infty} L^i \subseteq \cup_{i=k}^{\infty} L^i$. But in addition, $L^{k+i} \subseteq L^k$ for every i, so that $\cup_{i=k}^{\infty} L^i \subset L^k$. We conclude that $L^k = L^{k+1}$ if and only if $L^k = L^*$.
(b) The order is 3, because L^2 contains no string of length 6, and L^3 contains strings of all lengths ≥ 4.
(c) ∞, because the language does not contain Λ.
(d) It is not hard to see that this language contains every string in which the number of a's is either a multiple of 3, or 1 plus a multiple of 3. It follows from this that the order is 2.

3.42. (a) The language $\{aba\}^*$ can be described as the union of $\{\Lambda\}$ and the set of strings that start with ab and end with ba and contain none of the substrings bb, bab, and aaa. So one generalized regular expression describing this language is

$$\Lambda + ab\emptyset' \cap \emptyset' ba \cap (\emptyset' bb\emptyset')' \cap (\emptyset' bab\emptyset')' \cap (\emptyset' aaa\emptyset')'$$

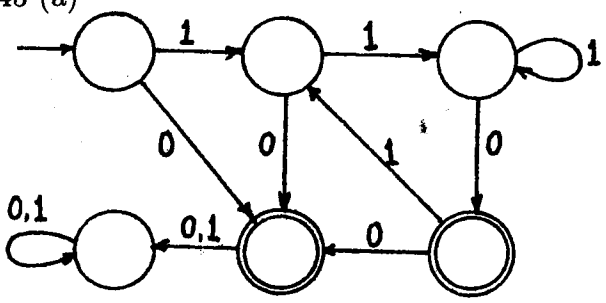
(b) $\{aa\}^*$ cannot be described this way. The intuitive reason is that without using *, there is no way to generate all even-length strings without also including some odd-length strings. A more rigorous proof follows.

Assume that the alphabet is $\{a\}$. (It's not hard to see that if we can show the result for this case, then it will still be impossible to describe $\{aa\}^*$ this way, even if larger alphabets are allowed.) For a language $L \subseteq \{a\}^*$, we let $e(L) = \{n \geq 0 \mid n \text{ is even and } a^n \in L\}$ and $e'(L) = \{n \geq 0 \mid n \text{ is even and } a^n \notin L\}$; similarly, $o(L) = \{n \geq 0 \mid n \text{ is odd and } a^n \in L\}$ and $o'(L) = \{n \geq 0 \mid n \text{ is odd and } a^n \notin L\}$. Obviously, $e(L) \cup e'(L)$ is the set of even integers, and $o(L) \cup o'(L)$ is the set of odd integers. We say L is *finitary* if either $e(L)$ or $e'(L)$ is finite and either $o(L)$ or $o'(L)$ is finite. The language $\{aa\}^*$ is nonfinitary, since both $e(\{aa\}^*)$ and $e'(\{aa\}^*)$ are infinite.

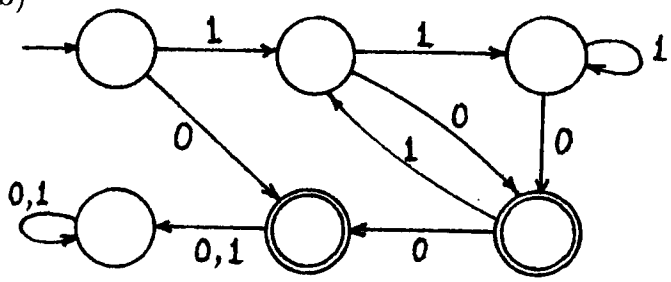
The fact that $\{aa\}^*$ cannot be described by a generalized regular expression not involving * follows from a more general result: no nonfinitary language can be described this way. This result follows from the fact that for languages $L_1, L_2 \subseteq \{a\}^*$, if L_1 and L_2 are finitary, then so are $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 L_2$, and L_1'. We will not verify all these statements, but we check one representative case.

Suppose, for example, that $e(L_1)$ is finite, $o(L_1)$ is finite, $e(L_2)$ is finite, and $o'(L_2)$ is finite. (Assume also that both L_1 and L_2 are nonempty.) Then $e(L_1 \cup L_2)$ is finite, and $o'(L_1 \cup L_2)$ is finite. Both $e(L_1 \cap L_2)$ and $o(L_1 \cap L_2)$ are finite. Both $e'(L_1')$ and $o'(L_1')$ are finite. Consider the number $e(L_1 L_2)$. If L_1 has no strings of odd length, then the strings in $L_1 L_2$ of even length are obtained by concatenating even-length strings in L_1 with even-length strings in L_2, and therefore $e(L_1 L_2)$ is finite. If there is an odd-length string in L_1, however, then since L_2 contains strings of almost all odd lengths, $L_1 L_2$ contains strings of almost all even lengths—i.e., $e'(L_1 L_2)$ is finite. Similarly, either $o(L_1 L_2)$ is finite (which will happen if L_1 has no strings of even length), or $o'(L_1 L_2)$ is finite (which is true if L_1 has a string of even length).

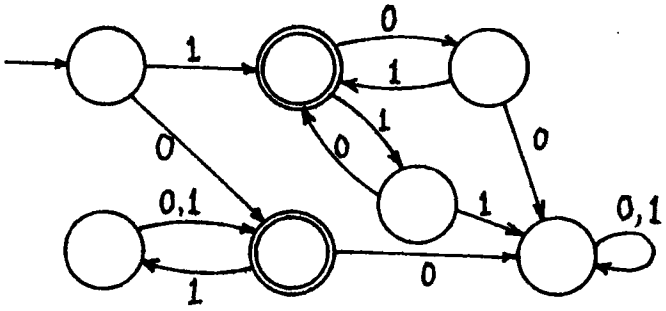
3.43 (a)



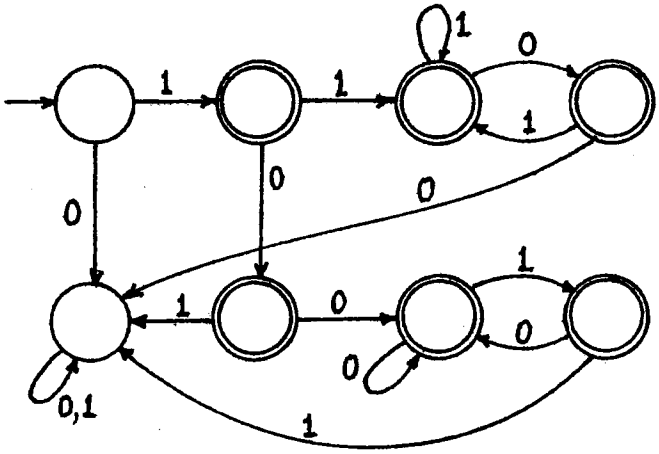
(b)



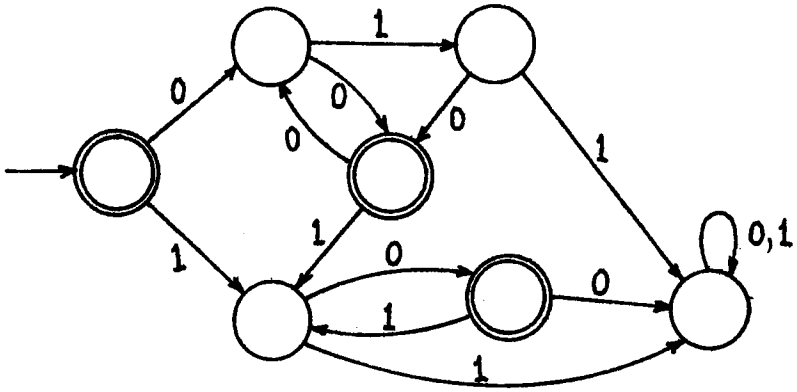
(c)



(d)



(e)



3.44. (a) Not valid. There is no way to use the definition to determine what $\delta^*(q, a)$ is, for $a \in \Sigma$.

(b) This is a valid definition, since $\delta^*(q, \Lambda)$ is defined, and for each x with $|x| \geq 1$, $\delta^*(q, x)$ is defined in terms of $\delta^*(p, y)$ for some state p and some string y of length $|x| - 1$.

Let the function being defined here be called δ_1. Then we must show that $\delta_1(q, x) = \delta^*(q, x)$ for every q and every x. In the basis step, we check that $\delta_1(q, \Lambda) = \delta^*(q, \Lambda)$. This is clear, since both are defined to be q. Suppose that for some y, $\delta_1(q, y) = \delta^*(q, y)$ for every state q. Then for $a \in \Sigma$ and $q \in Q$,

$$\delta_1(q, ay) = \delta_1(\delta(q, a), y) = \delta^*(\delta(q, a), y) = \delta^*(\delta(\delta^*(q, \Lambda), a), y) = \delta^*(\delta^*(q, a), y) = \delta^*(q, ay)$$

The first equality is the definition of δ_1; the second uses the induction hypothesis, with the state $\delta(q, a)$; the third and fourth use the definition of δ^*; and the last uses the formula in Exercise 3.25(a).

(c) This definition certainly leaves something to be desired. If $|xy| > 1$, there are strings w and z other than x and y for which $xy = wz$. If for some such x, y, w, and z, $\delta^*(\delta^*(q, x), y)$ and $\delta^*(\delta^*(q, w), z)$ were different, then the definition would not be a valid definition, because it would give different answers for $\delta^*(q, xy)$ and $\delta^*(q, wz)$ and these are supposed to be the same. However, it is possible to show that this can't happen, and so the definition may be said to be valid.

3.45. Let $x, y \in \{0, 1\}^*$ with $x \neq y$. We consider three cases. If $|x| = |y|$, then $xx \in L$ and $yx \notin L$, so that x and y are distinguishable. Otherwise relabel the strings if necessary so that $|x| < |y| = |x| + k$. If k is odd, then $xx \in L$, and $xy \notin L$ because $|xy| = 2|x| + k$, which is odd. Finally, if $k = 2j > 0$, let $w = w_1 w_2$, where w_1 and w_2 are any strings for which $|w_1| = |w_2| = j$ and the first symbol of w_2 is different from the first symbol of y. Then $xwxw \in L$. However, $ywxw = (yw_1)(w_2 x w_1 w_2)$, where the two parenthesized strings are of equal lengths and start with different symbols, so that $ywxw \notin L$.

3.46. (a) One answer is any two strings, neither of which is a prefix of an element of L: 1 and 10, for example. Another answer is any two distinct nonnull elements of L. In both these answers, the two strings are indistinguishable because nothing can be appended to either string (except Λ in the second case) so as to obtain an element of L. Another answer is two strings like 001 and 00011. Here an element of L is obtained in both cases if 1 is added to the end, and an element of L' is obtained in both cases if anything else is added to the end.

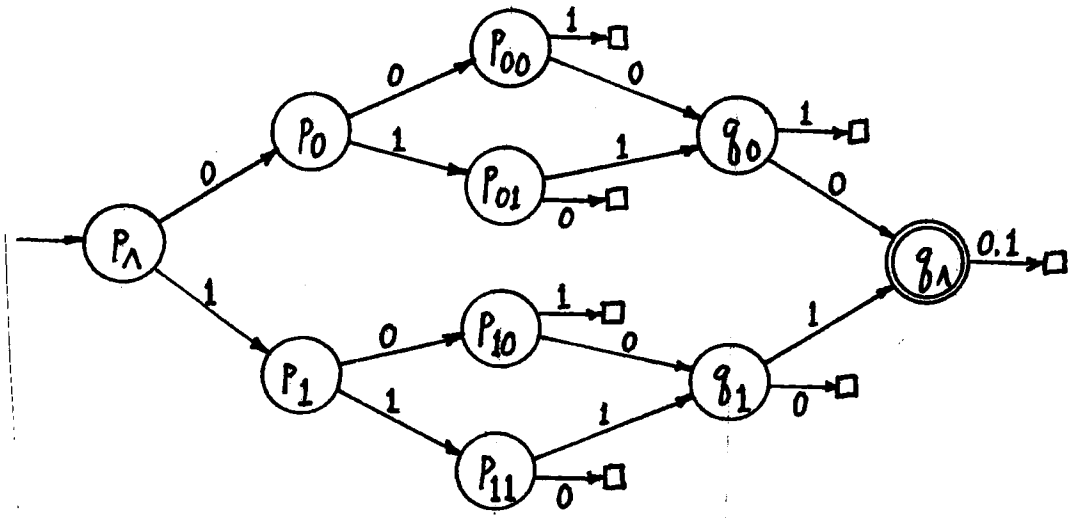
(b) Any two elements of $\{0^n \mid n \geq 0\}$ are distinguishable with respect to L.

3.47. In general, the 2^n states correspond to the 2^n possible strings of length n. We can see how to draw the transitions by thinking of the state corresponding to string x as representing the last n symbols of the input string: $\delta(a_1 a_2 \ldots a_n, a) = a_2 \ldots a_n a$. A string x of length $k < n$ corresponds to the string $0^{n-k}x$ with leading 0's. The initial state, therefore, is the one corresponding to the string 0^n. The accepting states are the ones corresponding to strings beginning with 1.

3.48. For L to be nonempty, n must obviously be even. If $n = 2m$, there is an FA with $(m+1)^2 + 1$ states accepting L. It has a state for each of the ordered pairs (i, j), where $0 \leq i \leq m$ and $0 \leq j \leq m$, and i and j represent the numbers of 0's and 1's, respectively, in the input string. For such a pair (i, j), each string with i 0's and j 1's is a prefix of an element of L. There is one additional state N corresponding to all the nonprefixes of elements of L. The transitions are the natural ones: if x has i 0's and j 1's, and $i < m$, then $\delta((i, j), 0) = (i + 1, j)$; similarly for $\delta((i, j), 0)$ if $j < m$. For each $i < m$, $\delta((m, i), 0) = \delta((i, m), 1) = N$.

This is the minimum possible number of states. It is straightforward to show that any set of $(m+1)^2 + 1$ strings consisting of one for each state is pairwise distinguishable with respect to L.

3.50. Here is a diagram of an FA accepting L in the case $n = 2$, which we call M_2.



We have simplified the picture by leaving out one of the states. There is a state q_D, which is not an accepting state, and $\delta(q_D, 0) = \delta(q_D, 1) = q_D$. Every arrow in the picture that ends in a square actually goes to q_D.

It is straightforward to generalize this to arbitrary n, so as to obtain an FA M_n. For each string x of length $\leq n$, there is a state p_x corresponding to x (that is, for which $\{y \mid \delta^*(q_0, y) = p_x\}$ is $\{x\}$. In addition, for each x with $|x| < n$, there is another state, which we call q_x, corresponding to the set $\{y \mid yx \in L\}$. The states p_x and q_x account collectively for all the strings that are prefixes of elements of L, and all other strings are taken care of by the single state q_D. Since there are $2^{n+1} - 1$ strings of length $\leq n$ and $2^n - 1$ strings of length $< n$, the total number of states in the FA is $(2^{n+1} - 1) + (2^n - 1) + 1 = 3 * 2^n - 1$. Now we show that no FA with fewer states can accept L, by showing that two strings corresponding to different states of M_n are distinguishable with respect to L.

Clearly any string that is a prefix of an element of L is distinguishable from any string that is not. Also, if x and y are both prefixes of elements of L and $|x| \neq |y|$, then x and y are distinguishable. It is therefore sufficient to show: (i) if $|x_1| = |x_2| \leq n$ and $x_1 \neq x_2$, then x_1 and x_2 are distinguishable; and (ii) if $|y_1| = |y_2|$, $y_1 \neq y_2$, $x_1 y_1 \in L$, and $x_2 y_2 \in L$, then x_1 and x_2 are distinguishable. Statement (i) is easy:

$$x_1 0^{2n-2|x_1|} x_1^r \in L \text{ and } x_2 0^{2n-2|x_1|} x_1^r \notin L$$

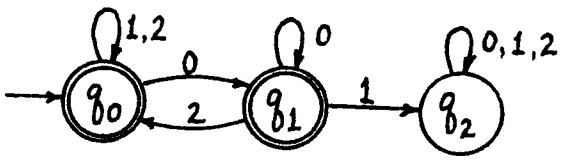
Statement (ii) is just as easy: if $x_1 y_1 \in L$ and $y_1 \neq y_2$, then since $x_2 y_2 \in L$, we must have $x_2 y_1 \notin L$. Therefore, y_1 distinguishes x_1 and x_2.

3.51. Suppose there an n and a set S of strings of length n so that for any x, $x \in L$ if and only if $x = x_1 x_2$ for some $x_2 \in S$. Then $\{x \in L \mid |x| \geq n\} = \Sigma^* S$, which is the concatenation of regular languages and therefore regular. L is the union of this language and a finite language, so that L is also regular.

3.52. A finite language L satisfies the property in a trivial way. We may choose n bigger than the length of the longest string in L, and S to be the empty set. Then for any string x with length n or greater, $x \in L$ if and only if x ends with some element of S, because neither condition is true for x.

3.53. Let $L = \{0x \mid x \in \{0,1\}^*\}$. Suppose n is any integer and S is any set of strings of length n. If $S = \emptyset$, the equivalence cannot be correct, because L is nonempty. However, if x is any element of S, the string $1x$ is not an element of L, even though it ends with an element of S. Therefore, L does not have this property.

3.54. The FA is pictured here.



Let a_i, b_i, and c_i be the number of strings x of length i for which $\delta^*(q_0, x)$ is q_0, q_1, and q_2, respectively. Then for each i, we may write these equations:

$$a_{i+1} = 2a_i + b_i \qquad b_{i+1} = a_i + b_i$$

(The first equation, for example, follows from the fact that there are two arrows to state q_0 from state q_0 and one to state q_0 from state q_1.) If we let n_i denote the number of strings of length i that don't contain the substring 01, then $n_i = a_i + b_i$. We may write

$$\begin{aligned} n_{i+1} &= a_{i+1} + b_{i+1} &&= 2a_i + b_i + a_i + b_i \\ &= 3(a_i + b_i) - b_i &&= 3n_i - (a_{i-1} + b_{i-1}) \\ &= 3n_i - n_{i-1} \end{aligned}$$

It is easy to verify that the sequence $m_i = f(2i+2)$ satisfies the same recursive relationship: $m_{i+1} = 3m_i - m_{i-1}$ for every $i \geq 1$. Furthermore, $n_0 = m_0$ and $n_1 = m_1$. Therefore, $n_i = m_i$ for every $i \geq 0$.

3.55. (a) The relation is reflexive, since for any FA $M = (Q, \Sigma, q_0, A, \delta)$, the identity function $f : Q \to Q$ defined by $f(q) = q$ is an isomorphism from M to itself.

Suppose that for $1 \leq k \leq 3$, $M_k = (Q_k, \Sigma, q_k, A_k, \delta_k)$, and $i : Q_1 \to Q_2$ and $j : Q_2 \to Q_3$ are isomorphisms from M_1 to M_2 and from M_2 to M_3, respectively. We construct isomorphisms from M_2 to M_1 and from M_1 to M_3. It will follow that the relation is both symmetric and transitive.

Since i and j are bijections, $i^{-1} : Q_2 \to Q_1$ and $j \circ i : Q_1 \to Q_3$ are also bijections. For any $p \in Q_2$ and $a \in \Sigma$, $i(i^{-1}(\delta_2(p, a))) = \delta_2(p, a)$, by definition of the function i^{-1}. But since i is an isomorphism from M_1 to M_2, we also have $i(\delta_1(i^{-1}(p), a)) = \delta_2(i(i^{-1}(p)), a) = \delta_2(p, a)$. Since i is one-to-one, $i^{-1}(\delta_2(p, a)) = \delta_1(i^{-1}(p), a)$. This is the formula i^{-1} needs to satisfy in order for it to be an isomorphism from M_2 to M_1. In addition, since for any $q \in Q_1$, q is an accepting state if and only if $i(q)$ is, it is also true that for any $p \in Q_2$, p is an accepting state if and only if $i^{-1}(p)$ is. Finally, since $i(q_1) = q_2$, $i^{-1}(q_2) = q_1$. Therefore, i^{-1} is an isomorphism from M_2 to M_1.

Now we show that $j \circ i$ is an isomorphism from M_1 to M_3.

$$j \circ i(\delta_1(q, a)) = j(i(\delta_1(q, a))) = j(\delta_2(i(q), a)) = \delta_3(j(i(q)), a) = \delta_3(j \circ i(q), a)$$

The second equality holds because i is an isomorphism, the third because j is. It is easy to check that for any $q \in Q_1$, q is an accepting state if and only if $j \circ i(q)$ is, and clearly $j \circ i(q_1) = q_3$.

(b) The proof is by structural induction on x. First, $i(\delta_1^*(q, \Lambda)) = i(q) = \delta_2^*(i(q), \Lambda)$. Now suppose y is a string for which $i(\delta_1^*(q, y)) = \delta^*(i(q), y)$ for every q. Then for any $a \in \Sigma$,

$$i(\delta_1^*(q, ya)) = i(\delta_1(\delta_1^*(q, y), a)) = \delta_2(i(\delta_1^*(q, y)), a) = \delta_2(\delta_2^*(i(q), y), a) = \delta_2^*(i(q), ya)$$

The first equality holds by the definition of δ_1^*; the second holds because i is an isomorphism; the third follows from the induction hypothesis; and the last uses the definition of δ_2^*.

(c) Suppose $i : M_1 \to M_2$ is an isomorphism. A string x is accepted by M_1 if and only if $\delta_1^*(q_1, x) \in A_1$. This is true if and only if $i(\delta_1^*(q_1, x)) \in A_2$, and by (b), this is true if and only if $\delta_2^*(i(q_1), x)) = \delta_2^*(q_2, x) \in A_2$. Therefore, x is accepted by M_1 if and only if x is accepted by M_2.

(d) Two. There is only one way to draw the transitions. Two nonisomorphic FAs are obtained by making the state an accepting state and a nonaccepting state, respectively.

(e) Suppose the two states are q_0 and q_1, with q_0 the initial state. In order to complete the transition diagram, we must decide three things: how to draw the transitions from q_0, how to draw the transitions from q_1, and which states to designate as accepting states. Since we require both states to be reachable from q_0, there are three ways to draw the transitions from q_0. There are four ways to draw the transitions from q_1; and since there is to be at least one accepting state, there are three ways of designating accepting states. The total number of transition diagrams is $3 * 4 * 3 = 36$, and it is easy to see that any two of these represent nonisomorphic FAs.

(f) All the FA's in which both states are accepting accept the language $\{0, 1\}^*$. However, any two of the remaining 24 accept different languages. This can be seen as follows. Let M_1 and M_2 be the two FAs, with transition functions δ_1 and δ_2, respectively. If q_0 is an accepting state in one and not the other, then Λ is in one but not both of the two languages; thus we assume that M_1 and M_2 have exactly the same accepting states. If

$\delta_1(q_0, 0) \neq \delta_2(q_0, 0)$, then 0 is in one of the languages but not the other, and similarly if $\delta_1(q_0, 1) \neq \delta_2(q_0, 1)$. Finally, if the transitions from the accepting state are the same in M_1 and M_2, and $\delta_1(q_1, 0) \neq \delta_2(q_1, 0)$, then there is a string with second symbol 0 that is in one but not both of the languages, and similarly if $\delta_1(q_1, 1) \neq \delta_2(q_1, 1)$. The conclusion is that the 36 nonisomorphic FAs in (e) accept 25 distinct languages.

Chapter 4

Nondeterminism and Kleene's Theorem

4.1. (a) $\delta^*(1,b) = \delta(1,b) = \{1,2\}$. $\delta^*(1,bb) = \delta(1,b) \cup \delta(2,b) = \{1,2\} \cup \emptyset = \{1,2\}$.
(b) $\delta^*(ba) = \delta(1,a) \cup \delta(2,a) = \{4,6,7\} \cup \{3\} = \{3,4,6,7\}$. $\delta^*(bab) = \cup_{p \in \{3,4,6,7\}} \delta(p,b) = \{1\} \cup \emptyset \cup \{9\} \cup \{8\} = \{1,8,9\}$.
(c) $\{1,2,9\}$ (d) $\{1,8,9\}$ (e) $\emptyset$

4.2. (a) $\delta^*(1,a) = \cup_{p \in \delta^*(1,\Lambda)} \delta(p,a) = \delta(1,a) = \{1,2\}$. $\delta^*(1,ab) = \delta(1,b) \cup \delta(2,b) = \{1,3\}$.
(b) $\delta^*(1,aba) = \delta(1,a) \cup \delta(3,a) = \{1,2,4\}$. $\delta^*(1,abaa) = \cup_{p \in \{1,2,4\}} \delta(p,a) = \{1,2,3,5\}$.
$\delta^*(1,abaab) = \delta(1,b) \cup \delta(2,b) \cup \delta(3,b) \cup \delta(5,b) = \{1,3,4,5\}$.

4.3. $\delta^*(q,a) = \cup_{p \in \delta^*(q,\Lambda)} \delta(p,a) = \cup_{p \in \{q\}} \delta(p,a) = \delta(q,a)$.

4.4. $\lceil \log_2 n \rceil$ (i.e., the smallest integer $\geq \log_2 n$). Reason: if there is an NFA accepting L with k states, then by the subset construction, there is an FA accepting L with no more than 2^k states. If there were an NFA with fewer than $\lceil \log_2 n \rceil$ states, there would be one with fewer than $\log_2 n$ states, and so there would be an FA with fewer than $2^{\log_2 n} = n$ states.

4.5. For every $q \in Q$, $\delta^*(q,\Lambda) = \{q\}$; for every $q \in Q$, $y \in \Sigma^*$, and $a \in \Sigma$, $\delta^*(q,ay) = \cup_{p \in \delta(q,a)} \delta^*(q,y)$.

4.6. An example is the language $\{\Lambda, 0\} \subseteq \{0\}^*$. Since Λ is in the language, the initial state must be an accepting state. Since 0 is also in the language, the 0-transition from the initial state must go to an accepting state. The only way for this to happen with just one accepting state is for all the strings 0^k to be accepted.

4.7. Yes. If $M = (Q, \Sigma, q_0, A, \delta)$ is any NFA for which $\Lambda \notin L(M)$, we can construct another NFA $M' = (Q \cup \{p\}, \Sigma, q_0, \{p\}, \delta')$ accepting $L(M)$, as follows. The state p is a new state not in Q, and it is the only accepting state of M'. All transitions in M are still present in M'. In addition, for every transition of the form $q \xrightarrow{a} r$, where $r \in A$, there is an additional transition $q \xrightarrow{a} p$ in M'. There are no transitions from p. Then it is easy to see that M' accepts $L(M)$. On the one hand, if $x = a_1 a_2 \ldots a_n \in L(M)$, then there is a sequence of transitions

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n$$

for some $q_n \in A$. The string is accepted by M' because of the sequence of transitions that agrees with this except that the last one is replaced by $q_{n-1} \xrightarrow{a_n} p$. On the other hand, if $x \in L(M')$, then there is a sequence of transitions ending with $q_n \xrightarrow{a_n} p$ by which x is accepted, and the state p doesn't appear earlier in the sequence. Therefore, there is a corresponding sequence of transitions by which M accepts x.

4.8. For both parts, we observe that for any sequence of transitions

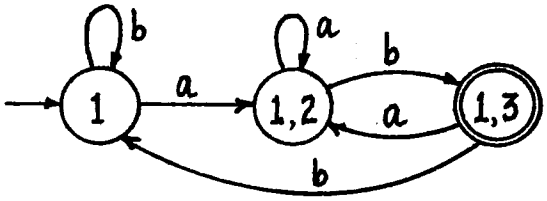
$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n$$

by which a string is accepted by M, each state q_i has the property that it is reachable from q_0 and some element of A is reachable from it. Therefore, neither the modification in (a) nor the one in (b) has any effect on the set of strings accepted by the NFA.

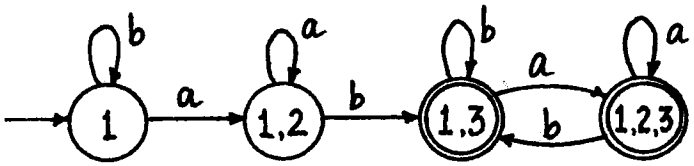
4.9. (a) There can be no more than m^n paths corresponding to x, since at each step there are at most m choices for the next state.

(b) If $x = x_1x_2$, $\delta^*(q_0, x) = \cup_{p\in\delta^*(q_0,x_1)}\delta^*(p, x_2)$. In order to use this formula to compute $\delta^*(q_0, x)$, it is necessary to know only the states in $\delta^*(q_0, x_1)$, not all the possible paths by which those states are reached. The computation tree really contains more information than we need; "pruning" it means ignoring everything except the set of states the NFA might end up at at each level of the tree.

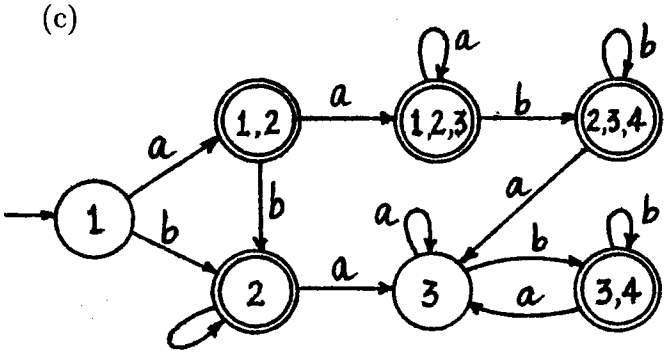
4.10. (a)



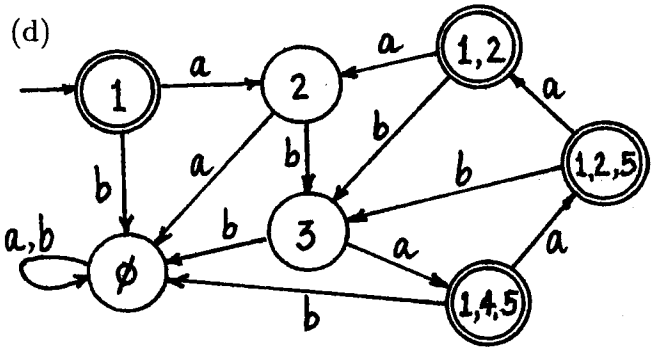
(b)



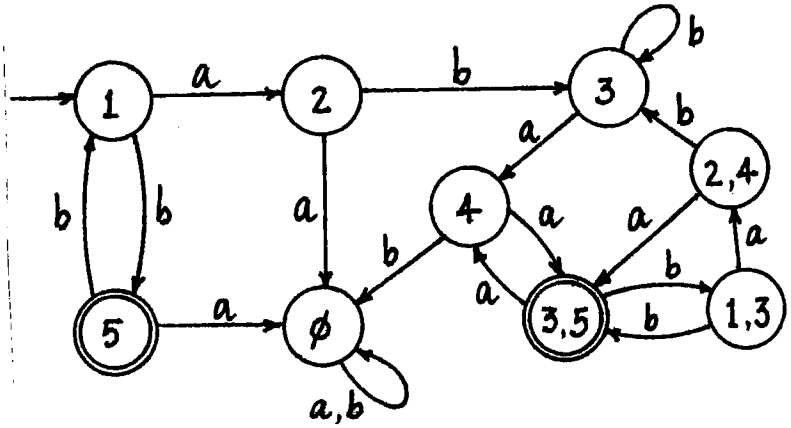
(c)



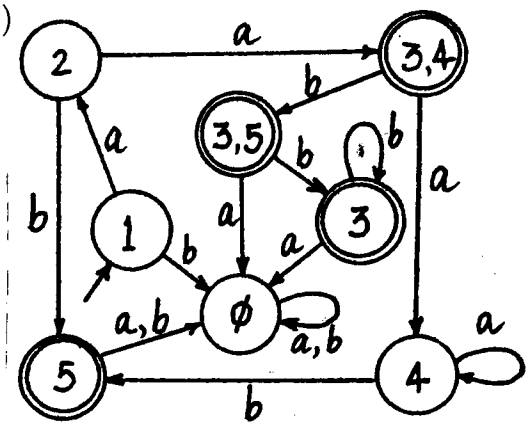
(d)



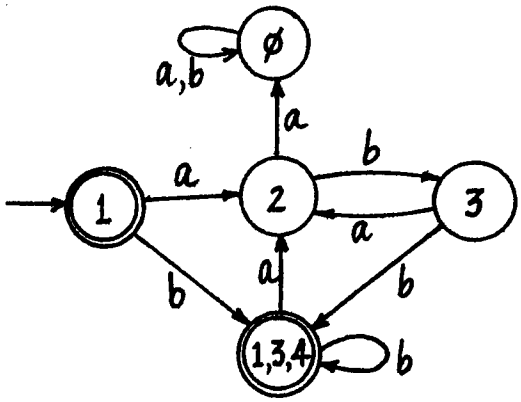
(e)



4.10 (f)



(g)



4.11. No, because there may be a string x so that in M, one sequence of transitions corresponding to x ends at an accepting state and another sequence doesn't. Then M' has this property also, which means that $x \in L(M)$ and $x \in L(M')$. An example can easily be constructed with just two states.

4.12. The conclusion of Theorem 3.4 does not hold in the case of unions. For example, if M_1 has an a-transition from q_1 to an accepting state, and M_2 has no a-transitions from q_2, then there are no a-transitions from (q_1, q_2), and so a is not accepted by M even though it is in $L(M_1) \cup L(M_2)$.

The conclusion holds in the case of intersections. On the one hand, if $x = a_1 a_2 \dots a_n$ is accepted by both M_1 and M_2, then there are sequences of transitions

$$q_1 \xrightarrow{a_1} q_1^{(1)} \xrightarrow{a_2} q_1^{(2)} \xrightarrow{a_3} \dots \xrightarrow{a_{n-1}} q_1^{(n-1)} \xrightarrow{a_n} q_1^{(n)}$$

and

$$q_2 \xrightarrow{a_1} q_2^{(1)} \xrightarrow{a_2} q_2^{(2)} \xrightarrow{a_3} \dots \xrightarrow{a_{n-1}} q_2^{(n-1)} \xrightarrow{a_n} q_2^{(n)}$$

where the $q_1^{(i)}$'s are elements of Q_1, with $q_1^{(n)} \in A_1$, and the $q_2^{(i)}$'s are elements of Q_2, with $q_2^{(n)} \in A_2$. In this case, the sequence of transitions

$$(q_1, q_2) \xrightarrow{a_1} (q_1^{(1)}, q_2^{(1)}) \xrightarrow{a_2} \dots \xrightarrow{a_n} (q_1^{(n)}, q_2^{(n)})$$

is defined in M and causes x to be accepted by M. This argument can essentially be reversed, so that if x is accepted by M, then it is accepted by both M_1 and M_2.

The conclusion does not hold in the case of differences. The example given for unions is also a counterexample here.

4.13. (a) yes (b) no (c) yes

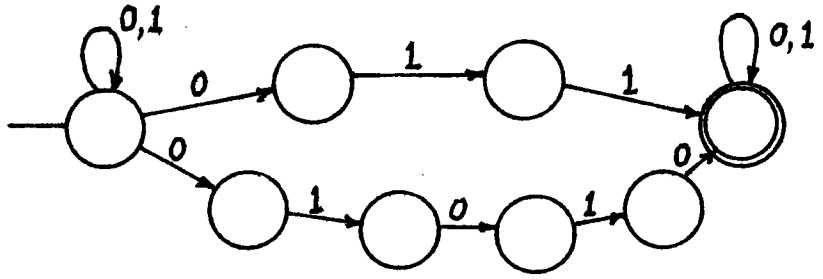
4.14. $aa^*b^*(a+b)(a+ba^*b^*(a+b))^*$

4.15. (a) $a(ab+baa)^*(aba)^*(aa+bab)a$
(b) $(ab^*a+baaba)^*$ (c) $((a^*b+ab^*)ab)^+$

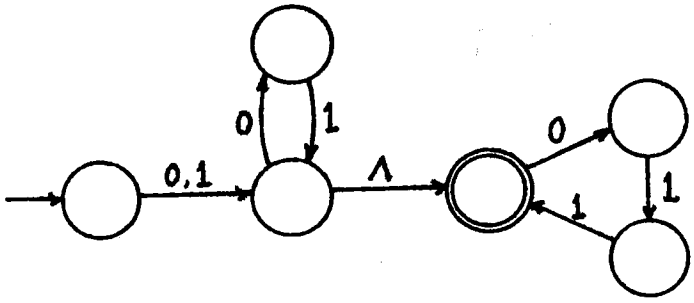
4.16. (a) $\{2,3,5\}$ (b) $\{1,2,5\}$ (c) $\{1,2,3,4,5\}$ (d) $\{3,5\}$ (e) $\{1,2,4,5\}$ (f) $\{1,2,3,4,5\}$

4.17. $\{1,3,4,6\}$

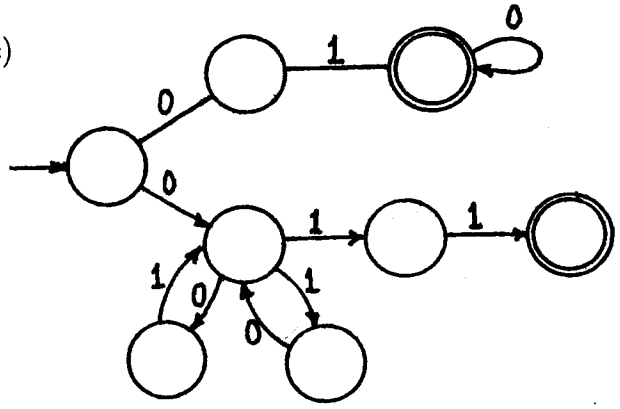
4.18. (a)



(b)



(c)



4.19. The new NFA-Λ, M^r, is constructed as follows. It has the same states as M, in addition to one new one, p_0, which will be the initial state of M^r. The only accepting state of M^r is q_0. All the transitions of M are present in M^r but in reverse; for example, if the transition $q \xrightarrow{a} r$ appears in M, then $r \xrightarrow{a} q$ appears in M^r, and if $q \xrightarrow{\Lambda} r$ appears in M, $r \xrightarrow{\Lambda} q$ appears in M'. Finally, M^r also has Λ-transitions from p_0 to all the states in the set A.

For any sequence of transitions in M from q_0 to $q_f \in A$, there is a sequence in M^r that starts at q_0, goes to q_f by a Λ-transition, and then follows the original path in reverse, ending up at q_0. On the other hand, for any sequence of transitions by which a string is accepted in M^r, the reverse sequence (except for the transition involving p_0) appears in M, so that the reverse of the string is accepted by M.

4.20. (a) Yes. (b) Yes. (c) No. The string ab is accepted by the first but not the second. (d) Yes. (e) No. The null string is accepted by the second but not the first.

4.21. The proof is by structural induction. We first observe that since there are no Λ-transitions in M_1, $\delta_1^*(q,\Lambda) = \{q\}$ and $\delta_1^*(q,xa) = \cup_{p\in\delta_1^*(q,x)}\delta_1(p,a)$, for every $q \in Q$, every

$x \in \Sigma^*$, and every $a \in \Sigma$.

For the basis step, we've already observed that $\delta_1^*(q, \Lambda) = \{q\}$, and $\delta^*(q, \Lambda) = q$ by definition of δ^*. Suppose that for some y, $\delta_1^*(q, y) = \{\delta^*(q, y)\}$ for every q. Then for $a \in \Sigma$,

$$\begin{aligned}
\delta_1^*(q, ya) &= \cup_{p \in \delta_1^*(q,y)} \delta_1(p, a) \\
&= \cup_{p \in \{\delta^*(q,y)\}} \delta_1(p, a) \\
&= \delta_1(\delta^*(q, y), a) \\
&= \{\delta(\delta^*(q, y), a)\} \\
&= \{\delta^*(q, ya)\}
\end{aligned}$$

The second equality uses the induction hypothesis, the fourth the definition of δ_1, and the last the definition of δ^*.

4.22. Yes, because for any set S, the Λ-closure of S is the same, no matter which way $\delta_1(q, \Lambda)$ is defined.

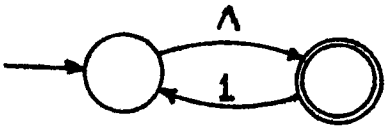
4.23. Let $M = (Q, \Sigma, q_0, A, \delta)$ be the NFA-Λ. Then the FA $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ can be defined as follows. $Q_1 = 2^Q$; $q_1 = \{q_0\}$; A_1 is the set $\{q \in Q_1 \mid q_0 \in q \text{ or } q \cap A \neq \emptyset\}$, if $\Lambda(\{q_0\}) \cap A \neq \emptyset$, or $\{q \in Q_1 \mid q \cap A \neq \emptyset\}$, if $\Lambda(\{q_0\}) \cap A = \emptyset$.

4.24. If $x \neq \Lambda$, and $\delta_1^*(q_0, x)$ contains a state q for which $\Lambda(\{q\}) \cap A \neq \emptyset$, then $q \in \delta_1^*(q_0, x)$ since $\delta_1^*(q_0, x) = \delta^*(q_0, x)$ and the second set is Λ-closed. As we observed in the proof of Theorem 4.2, however, $\delta_1^*(q_0, \Lambda)$ is $\{q_0\}$ by definition, and this is *not* $\delta^*(q_0, \Lambda)$ if $\{q_0\}$ is not Λ-closed.

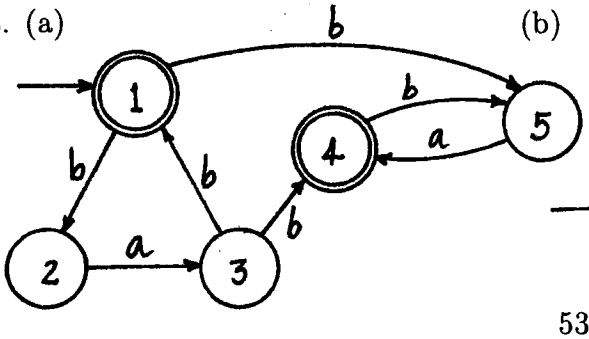
4.25. L^+

4.26. In both cases, a single state can be added. In (a), the new state is the initial state, and there is a Λ-transition from it to the state that was the initial state of the original machine. In (b), the new state is the only accepting state, and there are Λ-transitions from each of the previous accepting states to the new one. Otherwise, the transitions are the same as in the original.

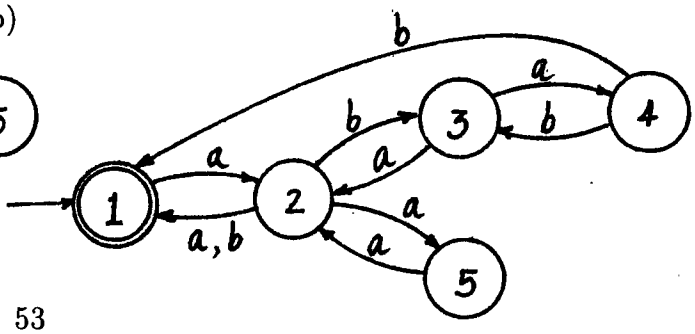
4.27. (This works for both cases.)



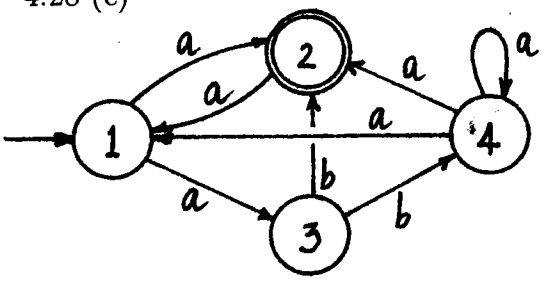
4.28. (a)



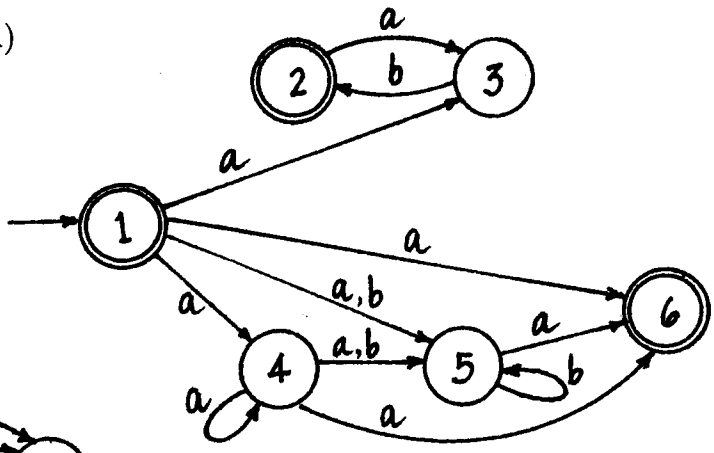
(b)



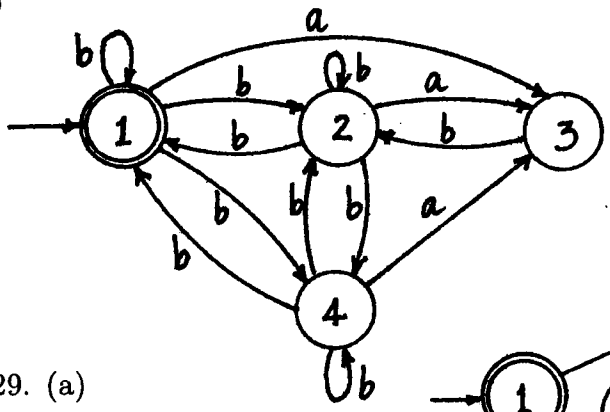
4.28 (c)



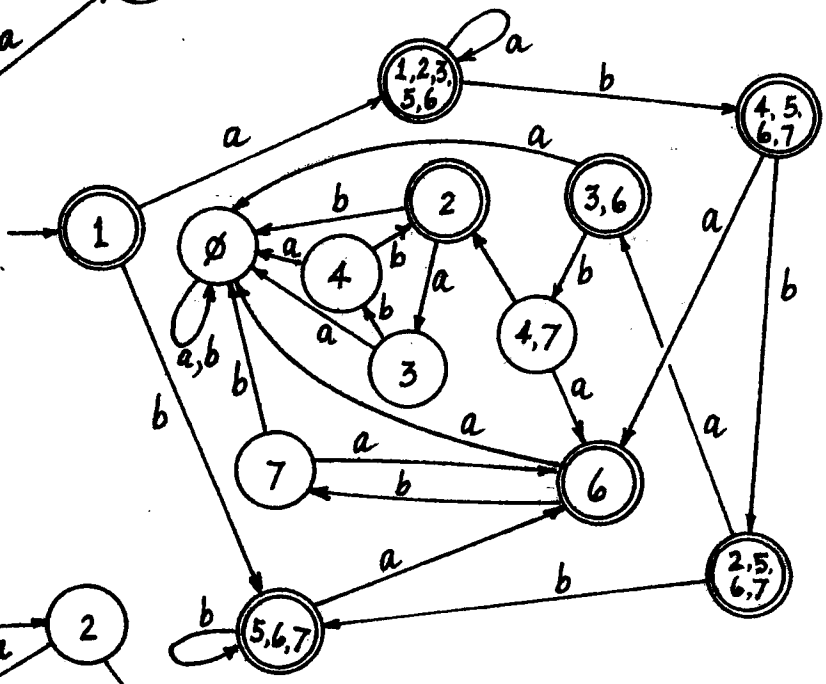
(d)



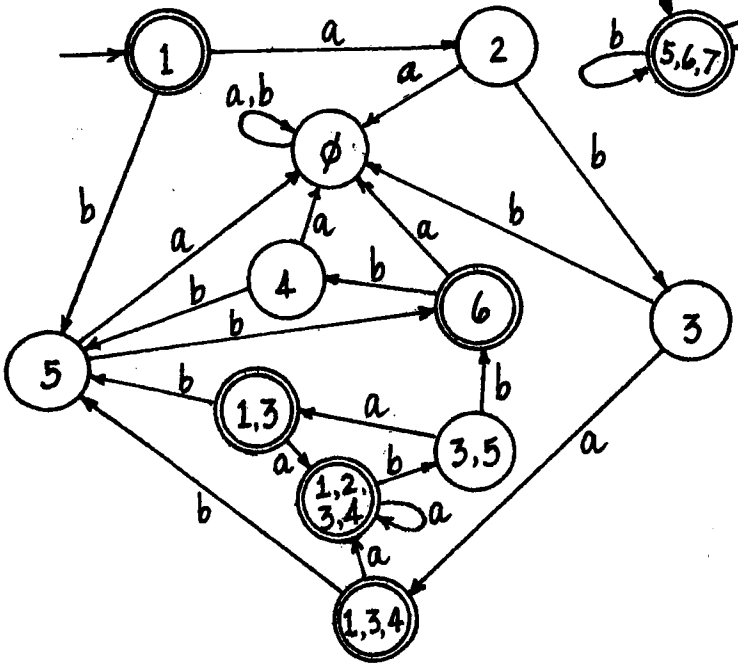
(e)



4.29. (a)

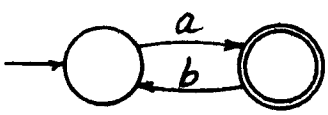


(b)

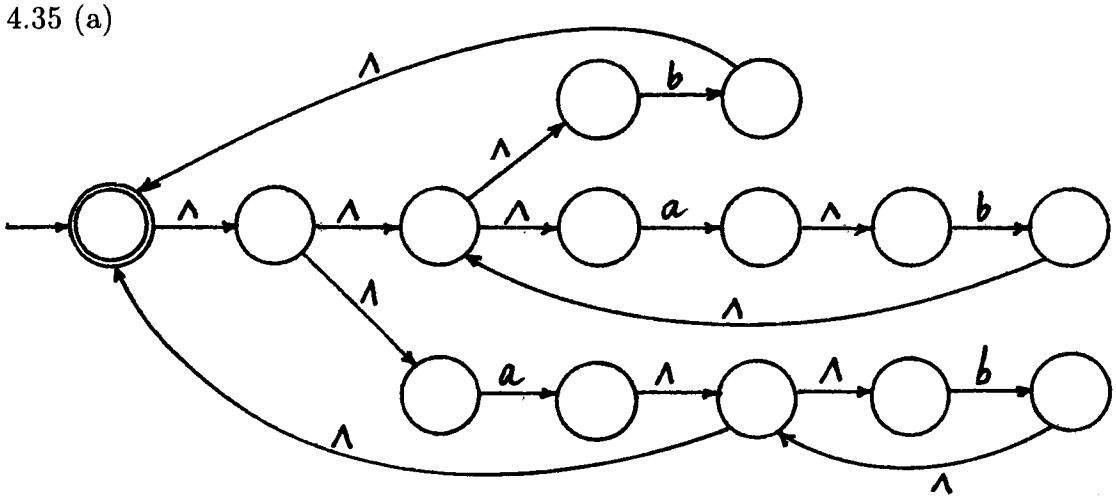


4.32 and 4.33. If M_1 and M_2 are one-state NFAs accepting $\{a\}^*$ and $\{b\}^*$, respectively, then neither construction is correct.

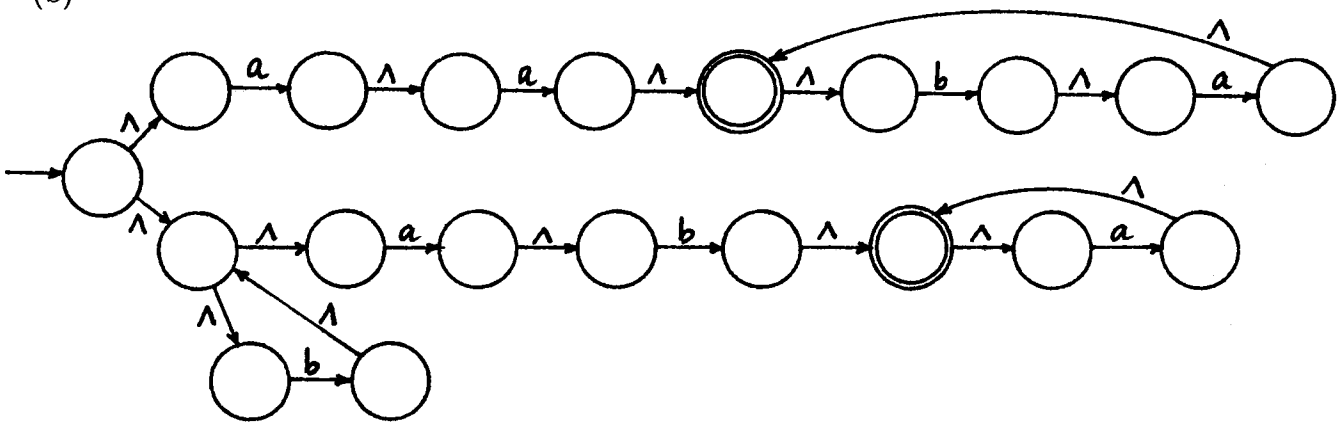
4.34. If M_1 is the machine shown, accepting $L = \{a\}\{ba\}^*$, this construction would allow ab, which isn't in L^*, to be accepted.



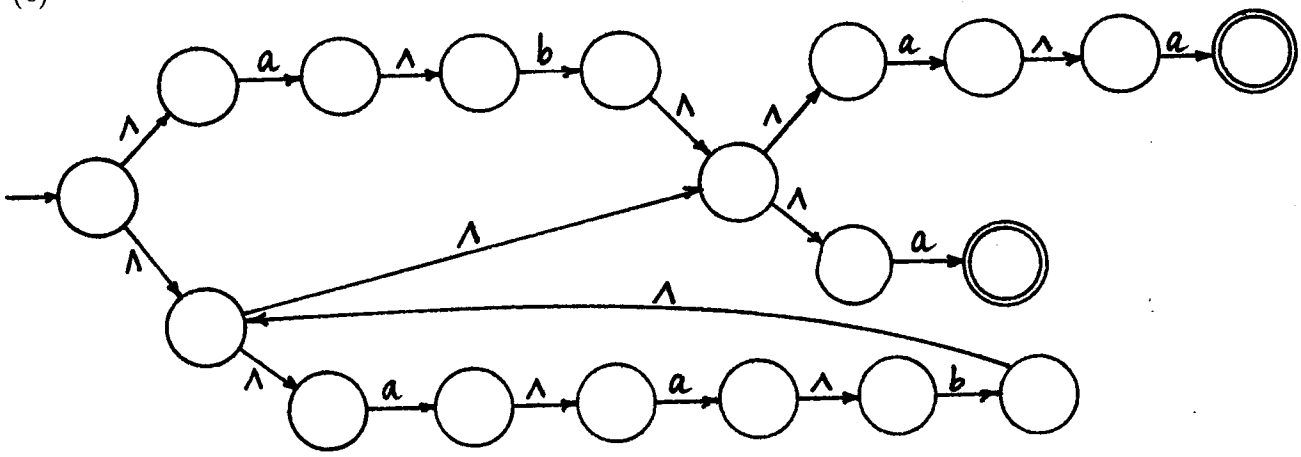
4.35 (a)



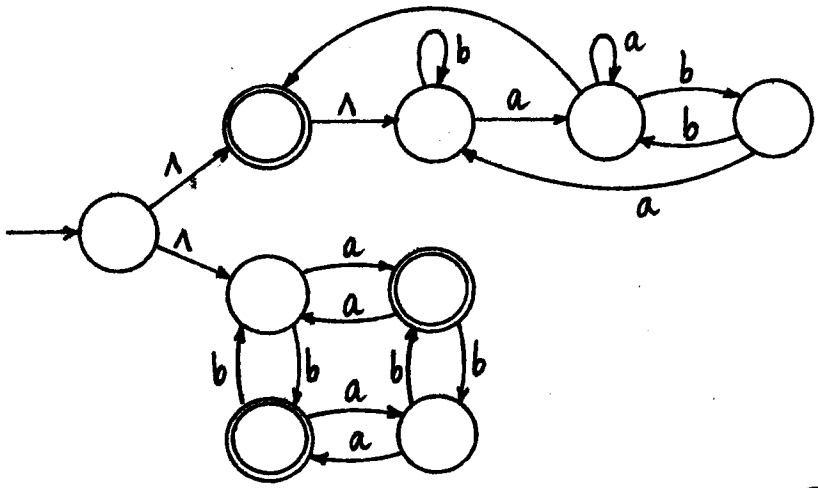
(b)



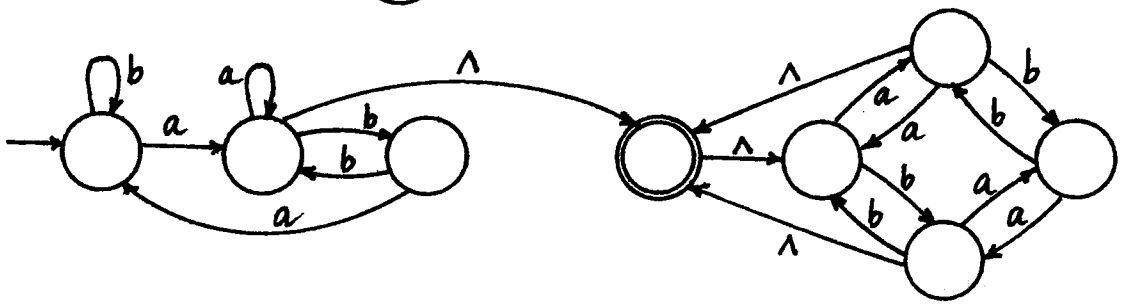
(c)



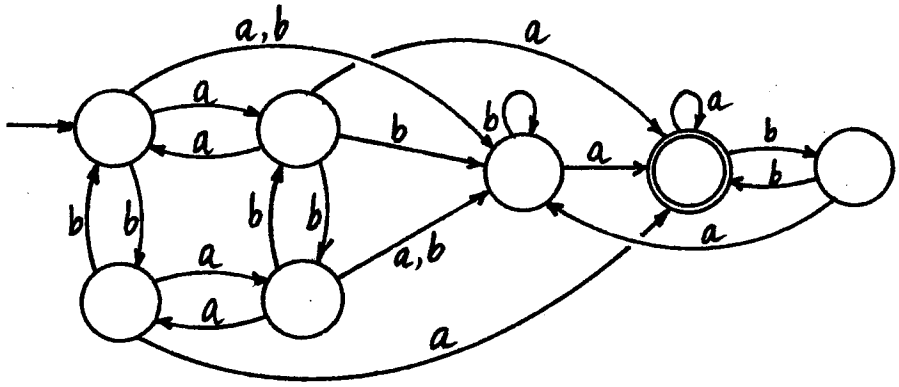
4.36. (e)

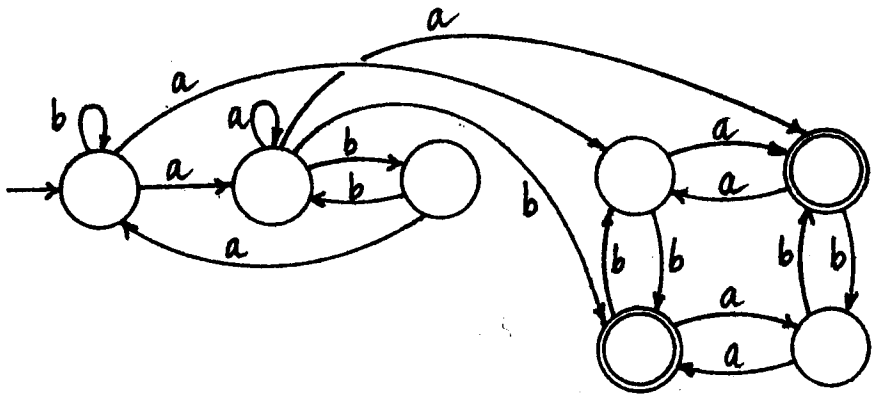


(f)



4.37.





4.38. Here are the tables with the regular expressions $r(p,q,k)$ for $0 \le k \le 2$.

p	$r(p,1,0)$	$r(p,2,0)$	$r(p,3,0)$
1	Λ	b	a
2	a	Λ	b
3	b	a	Λ

p	$r(p,1,1)$	$r(p,2,1)$	$r(p,3,1)$
1	Λ	b	a
2	a	$\Lambda + ab$	$b + aa$
3	b	$a + bb$	$\Lambda + ba$

p	$r(p,1,2)$	$r(p,2,2)$	$r(p,3,2)$
1	$(ba)^*$	$b(ab)^*$	$a + b(ab)^*(b + aa)$
2	$a(ba)^*$	$(ab)^*$	$(ab)^*(b + aa)$
3	$b + (a + bb)(ab)^*a$	$(a + bb)(ab)^*$	$\Lambda + ba + (a + bb)(ab)^*(b + aa)$

The language corresponds to the regular expression $r(1,3,3) = r(1,3,2) + r(1,3,2)r(3,3,2)^*r(3,3,2)$, or $r(1,3,2)r(3,3,2)^*$, or

$$(a + b(ab)^*(b + aa))(ba + (a + b)(ab)^*(b + aa))^*$$

(b) Here are simplified tables with the regular expressions $r(p,q,k)$ for $0 \le k \le 2$.

p	$r(p,1,0)$	$r(p,2,0)$	$r(p,3,0)$
1	Λ	$a + b$	$\emptyset$
2	$\emptyset$	$\Lambda + a$	b
3	b	a	Λ

p	$r(p,1,1)$	$r(p,2,1)$	$r(p,3,1)$
1	Λ	$a + b$	$\emptyset$
2	$\emptyset$	$\Lambda + a$	b
3	b	$a + ba + bb$	Λ

p	$r(p,1,2)$	$r(p,2,2)$	$r(p,3,2)$
1	Λ	$(a + b)a^*$	$(a + b)a^*b$
2	$\emptyset$	a^*	a^*b
3	b	$(a + ba + bb)a^*$	$\Lambda + (a + ba + bb)a^*b$

The language corresponds to the regular expression $r(1,2,2) + r(1,3,2)r(3,3,2)^*r(3,2,2)$, or

$$(a + b)a^* + (a + b)a^*b((a + ba + bb)a^*b)^*(a + ba + bb)a^*$$

(c) Here is the table $r(p,q,0)$.

p	$r(p,1,0)$	$r(p,2,0)$	$r(p,3,0)$	$r(p,4,0)$
1	Λ	$a + b$	$\emptyset$	$\emptyset$
2	$\emptyset$	Λ	$a + b$	$\emptyset$
3	$\emptyset$	$\emptyset$	Λ	$a + b$
4	a	b	$\emptyset$	Λ

The table $r(p,q,1)$ is exactly the same except that $r(4,2,1) = b + a(a + b)$. The table $r(p,q,2)$ is the same as $r(p,q,1)$ except that $r(1,3,2) = (a+b)^2$ and $r(4,3,2) = a(a+b)^2 + b(a + b)$.

Here is the table $r(p,q,3)$.

p	$r(p,1,0)$	$r(p,2,0)$	$r(p,3,0)$	$r(p,4,0)$
1	Λ	$a+b$	$(a+b)^2$	$(a+b)^3$
2	$\emptyset$	Λ	$a+b$	$(a+b)^2$
3	$\emptyset$	$\emptyset$	Λ	$a+b$
4	a	$b+a(a+b)$	$a(a+b)^2+b(a+b)$	$\Lambda+a(a+b)^3+b(a+b)^2$

The language corresponds to the expression $r(1,2,4)+r(1,3,4)$, or $r(1,2,3)+r(1,4,3)r(4,4,3)^*r(4,2,3$ $r(1,3,3)+r(1,4,3)r(4,4,3)^*r(4,3,3)$, or (rearranging and simplifying slightly)

$$a+b+(a+b)^2+(a+b)^3(a(a+b)^3+b(a+b)^2)^*)(b+(\Lambda+a)(a+b)^2)$$

(d) Here are the tables with the regular expressions $r(p,q,k)$ for $0 \le k \le 2$.

p	$r(p,1,0)$	$r(p,2,0)$	$r(p,3,0)$	$r(p,4,0)$
1	$\Lambda+b$	$\emptyset$	a	$\emptyset$
2	a	Λ	b	$\emptyset$
3	$\emptyset$	a	Λ	b
4	$\emptyset$	b	$\emptyset$	$\Lambda+a$

p	$r(p,1,1)$	$r(p,2,1)$	$r(p,3,1)$	$r(p,4,1)$
1	b^*	$\emptyset$	b^*a	$\emptyset$
2	ab^*	Λ	$b+ab^*a$	$\emptyset$
3	$\emptyset$	a	Λ	b
4	$\emptyset$	b	$\emptyset$	$\Lambda+a$

p	$r(p,1,2)$	$r(p,2,2)$	$r(p,3,2)$	$r(p,4,2)$
1	b^*	$\emptyset$	b^*a	$\emptyset$
2	ab^*	Λ	$b+ab^*a$	$\emptyset$
3	aab^*	a	$\Lambda+ab+aab^*a$	b
4	bab^*	b	$bb+bab^*a$	$\Lambda+a$

Here are the regular expressions $r(p,q,3)$:

$r(1,1,3) = b^* + b^*a(ab+aab^*a)^*aab^*$
$r(1,2,3) = b^*a(ab+aab^*a)^*a$
$r(1,3,3) = b^*a(ab+aab^*a)^*$
$r(1,4,3) = b^*a(ab+aab^*a)^*b$
$r(2,1,3) = ab^* + (b+ab^*a)(ab+aab^*a)^*aab^*$
$r(2,2,3) = \Lambda + (b+ab^*a)(ab+aab^*a)^*a$
$r(2,3,3) = (b+ab^*a)(ab+aab^*a)^*$
$r(2,4,3) = (b+ab^*a)(ab+aab^*a)^*b$
$r(3,1,3) = (ab+aab^*a)^*aab^*$
$r(3,2,3) = (ab+aab^*a)^*a$
$r(3,3,3) = (ab+aab^*a)^*$

$r(3,4,3) = (ab + aab^*a)^*b$
$r(4,1,3) = bab^* + (bb + bab^*a)(ab + aab^*a)^*aab^*$
$r(4,2,3) = b + (bb + bab^*a)(ab + aab^*a)^*a$
$r(4,3,3) = (bb + bab^*a)(ab + aab^*a)^*$
$r(4,4,3) = a + (bb + bab^*a)(ab + aab^*a)^*b$

Finally, the language corresponds to the regular expression $r(1,1,4)$, which is
$r(1,1,3) + r(1,4,3)r(4,4,3)^*r(4,1,3) = b^* + b^*a(ab + aab^*a)^*aab^* \quad +$
$b^*a(ab + aab^*a)^*b(b^* + b^*a(ab + aab^*a)^*aab^*)^*(bab^* + (bb + bab^*a)(ab + aab^*a)^*aab^*)$.

4.39. If $\delta(q,\Lambda) = \{p\}$, $\delta(p,a) \neq \emptyset$, and $\delta(q,a) = \emptyset$, for example, each of these definitions would say that $\delta^*(q,a) = \emptyset$, and so none of them is correct.

4.40. (a) We must show two things: first, that for every $s \in S$, $s \in \Lambda(T)$, and second, that for any $t \in \Lambda(T)$, $\Lambda(\{t\}) \subseteq \Lambda(T)$. The first is true because $S \subseteq T$ and $T \subseteq \Lambda(T)$ by definition. The second is part of the definition of $\Lambda(T)$.

(b) We know from the definition of Λ-closure that $\Lambda(S) \subseteq \Lambda(\Lambda(S))$. To show the opposite inclusion using structural induction, we must show that for every $s \in \Lambda(S)$, $s \in \Lambda(S)$, and for every $t \in \Lambda(S)$, $\Lambda(\{t\}) \subseteq \Lambda(S)$. The first is trivial, and the second is part of the definition of $\Lambda(S)$.

(c) The statement $\Lambda(S \cup T) \subseteq \Lambda(S) \cup \Lambda(T)$ is easily shown by structural induction. The opposite inclusion follows from the two statements $\Lambda(S) \subseteq \Lambda(S \cup T)$ and $\Lambda(T) \subseteq \Lambda(S \cup T)$, both of which are true by part (a).

(d) From (c), we have the result when $|S| = 2$, and it is easy to extend the result to arbitrary n by induction.

(e) $\Lambda(S \cap T)$ is a subset both of $\Lambda(S)$ and of $\Lambda(T)$, by part (a), and therefore $\Lambda(S \cap T) \subseteq \Lambda(S) \cap \Lambda(T)$.

(f) We always have $S \subseteq \Lambda(S)$ and therefore $\Lambda(S)' \subseteq S' \subseteq \Lambda(S')$. If the first and third of these two sets are equal, then the equality of the first two implies that $S = \Lambda(S)$, and the equality of the second and third implies that $S' = \Lambda(S')$. The converse is also clearly true: if $S = \Lambda(S)$ and $S' = \Lambda(S')$, then $\Lambda(S)' = \Lambda(S')$. Thus, the condition is true precisely when there is neither a Λ-transition from an element of S to an element of S' nor a Λ-transition from an element of S' to an element of S. (A more concise way to say this is to say that S and S' are both Λ-closed—see the next exercise.)

4.41. (a) For any two sets S and T, we have $S \cup T \subseteq \Lambda(S \cup T) \subseteq \Lambda(S) \cup \Lambda(T)$. (The second inclusion is easily checked using structural induction.) Therefore, if S and T are Λ-closed, $S \cup T = \Lambda(S \cup T)$.

(b) For any two sets S and T, we have $S \cap T \subseteq \Lambda(S) \cap \Lambda(T) \subseteq \Lambda(S) \cap \Lambda(T)$. If S and T are Λ-closed, it follows that $S \cap T = \Lambda(S \cap T)$.

(c) $\Lambda(S)$ is a Λ-closed set containing S as a subset. If T is any other such set, then since $S \subseteq T$, $\Lambda(S) \subseteq \Lambda(T)$ (by Exercise 4.40(a)), and therefore $\Lambda(S) \subseteq T$ because T is Λ-closed.

4.42. We show the formula for an NFA-Λ, and the formula for NFAs will follow as a special

case. The proof is by structural induction on y and will be very enjoyable for those who like mathematical notation. The basis step is for $y = \Lambda$. For any $q \in Q$ and any $x \in \Sigma^*$,

$$\bigcup_{p \in \delta^*(q,x)} \delta^*(p, \Lambda) = \bigcup_{p \in \delta^*(q,x)} \Lambda(\{p\}) = \Lambda(\bigcup_{p \in \delta^*(q,x)} \{p\}) = \Lambda(\delta^*(q,x)) = \delta^*(q,x) = \delta^*(q, x\Lambda)$$

In this formula, we are using parts of Exercise 4.40. The first equality uses the definition of δ^* for an NFA-Λ. The second uses part (d) of Exercise 4.40. The third is simply the definition of union. The fourth uses part (b) of Exercise 4.40 (and the fact that according to the definition, $\delta^*(q,x)$ is in fact $\Lambda(S)$ for some set S).

Now assume that for the string y,

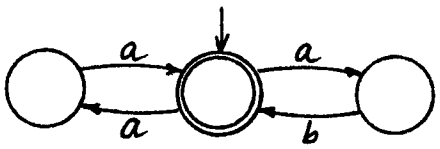
$$\delta^*(q, xy) = \bigcup_{p \in \delta^*(q,x)} \delta^*(p, y)$$

for every state q and every string $x \in \Sigma^*$. Then for any $a \in \Sigma$,

$$\begin{aligned}
\delta^*(q, x(ya)) &= \delta^*(q, (xy)a) \\
&= \Lambda(\bigcup_{p \in \delta^*(q,xy)} \delta(p, a)) \\
&= \Lambda(\{s \mid \exists p(p \in \delta^*(q, xy) \wedge s \in \delta(p, a))\}) \\
&= \Lambda(\{s \mid \exists p(p \in \bigcup_{r \in \delta^*(q,x)} \delta^*(r, y) \wedge s \in \delta(p, a))\}) \\
&= \Lambda(\{s \mid \exists r(r \in \delta^*(q, x) \wedge \exists p(p \in \delta^*(r, y) \wedge s \in \delta(p, a)))\}) \\
&= \Lambda(\{s \mid \exists r(r \in \delta^*(q, x) \wedge s \in \bigcup_{p \in \delta^*(r,y)} \delta(p, a))\}) \\
&= \Lambda(\bigcup_{r \in \delta^*(q,x)} (\bigcup_{p \in \delta^*(r,y)} \delta(p, a))) \\
&= \bigcup_{r \in \delta^*(q,x)} \Lambda(\bigcup_{p \in \delta^*(r,y)} \delta(p, a)) \\
&= \bigcup_{r \in \delta^*(q,x)} \delta^*(r, ya)
\end{aligned}$$

The second equality uses the definition of δ^*. The third is just a restatement of the definition of union. The fourth uses the induction hypothesis. The fifth, sixth, and seventh are also just restatements involving the definitions of various unions. The eighth uses part (c) of Exercise 4.40 (actually, the generalization of part (c) from the union of two sets to an arbitrary union). The ninth is the definition of δ^*.

4.43. The language accepted by each M_1^c is a subset of $L(M)$. However, there may be strings in $L(M)$ not accepted by any M_1^c. If M is the FA below, there are two possible M_1^c's, each with an unreachable state, and neither accepts the string $aaab$.



4.44. (a) The set of suffixes of elements of L. On the one hand, if $x = yz \in L$, $q \in \delta^*(q_0, y)$, and $q_f \in \delta^*(q, z)$, then M_1 contains a Λ-transition from q_0 to q, which implies that z is accepted by M_1. On the other hand, if $z = b_1 b_2 \dots b_m \in L(M)$, where each $b_i \in \Sigma \cup \{\Lambda\}$, and the states $q_0 = p_0, p_1, p_2, \dots, p_m = q_f$ are such that $p_i \in \delta_1(p_{i-1}, b_i)$ for each i (where δ_1 is the transition function of M_1), then the assumptions regarding q_0 and q_f imply that the only one of these transitions that may not be present in M is the first one, which may be one of the Λ-transitions added to M_1. It follows that there is some string y so that $p_1 \in \delta^*(q_0, y)$ and that $q_f \in \delta^*(q_0, yz)$. In other words, z is a suffix of an element of L.

(b) An argument like the one in part (a) shows that M_2 accepts the strings that are prefixes of elements of L.

(c) M_3 accepts the language of substrings of elements of L.

(d) M_4 accepts the set of all strings that can be obtained from elements of L by deleting zero or more substrings: that is, strings of the form $y_1 y_2 \dots y_n$ for which there are strings $x_0, x_1, \dots, x_n$ satisfying $x_0 y_1 x_1 y_2 x_2 \dots x_{n-1} y_n x^n \in L$.

4.45. (a) It is at least clear that the two one-state NFAs accepting 0^* and 1^* cannot be joined so that the composite NFA accepts the union of the two languages. No matter which state was chosen as the initial state, either 01 or 10 would have to be accepted. It is easy to show by looking at cases that no two-state NFA can work.

(b) A simple condition that makes it possible to incorporate M_1 and M_2 into a composite NFA is that there are no transitions to the initial state of M_1, and either $\Lambda \in L(M_1)$ or $\Lambda \notin L(M_2)$. In this case, for each $a \in \Sigma$, a-transitions can be drawn from the initial state of M_1 to every state q in M_2 for which $q \in \delta_2(q_2, a)$ (where q_2 is the initial state of M_2). If M_1 contains transitions to q_1, the initial state, then it may still be possible to draw transitions from q_1 to states of M_2, provided every string x with $q_1 \in \delta_1^*(q_1, x)$ is a prefix of an element of $L(M_2)$.

4.46. (a) $f(\Lambda) = f(\Lambda\Lambda) = f(\Lambda)f(\Lambda)$, and it follows that $f(\Lambda) = \Lambda$.

(b) We observe that for any $L_1, L_2 \subseteq \Sigma_1^*$, we have $f(L_1 \cup L_2) = f(L_1) \cup f(L_2)$, $f(L_1 L_2) = f(L_1)f(L_2)$, and $f(L_1^*) = f(L_1)^*$. (The last two are because f is a homomorphism.) Now we can show very easily, using structural induction, that if L is regular, so is $f(L)$. First, if L is any of the regular languages $\emptyset$, $\{\Lambda\}$, or $\{a\}$ (where $a \in \Sigma_1$), then $f(L)$ is regular, since $f(\emptyset) = \emptyset$, $f(\{\Lambda\}) = \{\Lambda\}$, and $f(\{a\}) = \{f(a)\}$. Second, if $f(L_1)$ and $f(L_2)$ are regular, then so are $f(L_1 \cup L_2)$, $f(L_1 L_2)$, and $f(L_1^*)$.

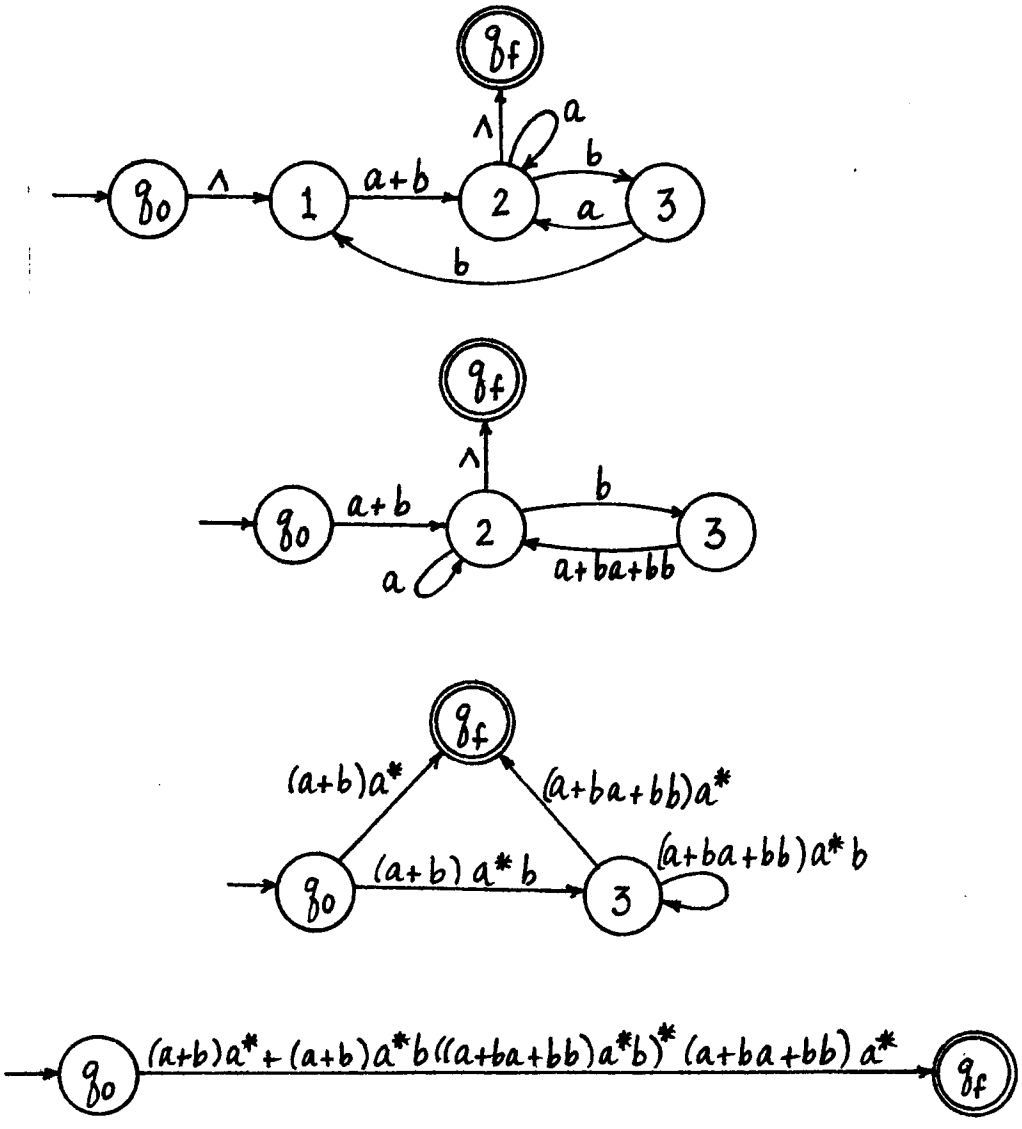
(In fact, it is easy to check that if r is a regular expression representing L, then a regular expression representing $f(L)$ can be obtained by replacing every alphabet symbol a in r by the string $f(a)$.)

(c) Suppose $M = (Q, \Sigma_2, q_0, A, \delta)$ is an FA accepting $L \subseteq \Sigma_2^*$. We define an FA $M_1 = (Q, \Sigma_1, q_0, F, \delta_1)$ accepting $f^{-1}(L)$ as follows: for $q \in Q$ and $a \in \Sigma_1$, $\delta_1(q, a) = \delta^*(q, f(a))$. Then it follows easily from the fact that f is a homomorphism that for every $q \in Q$ and every $x \in \Sigma_1^*$, $\delta_1^*(q, x) = \delta^*(q, f(x))$. Therefore, M_1 accepts x if and only if M accepts $f(x)$. This means that for any $x \in \Sigma_1^*$, $x \in f^{-1}(L)$ if and only if M_1 accepts x.

4.47. The reduction step in which p is eliminated is to redefine $e(q,r)$ for every pair (q,r) where neither q nor r is p. (This includes the pairs (q,q).) This is done using the formula

$$e(q,r) = e(q,r) + e(q,p)e(p,p)^*e(p,r)$$

We illustrate this process for the FA shown in part (b) of Figure 4-22. The first picture shows the NFA-Λ with the states q_0 and q_f; the subsequent pictures describe the machines obtained by eliminating the states 1, 2, and 3, in that order. The regular expression $e(q_0, q_f)$ in the last picture describes the language. (Note that it looks the same as the regular expression obtained in the solution to Exercise 4.38(b), although the two algorithms do not always produce regular expressions that look the same.)



Chapter 5

Regular and Nonregular Languages

5.1. If there is only one equivalence class, then there cannot be two strings for which one is in L and the other isn't. Therefore, either $L = \emptyset$ or $L = \{0,1\}^*$.

5.2. There are $|x| + 2$ equivalence classes, one for each prefix of x and one containing all the nonprefixes.

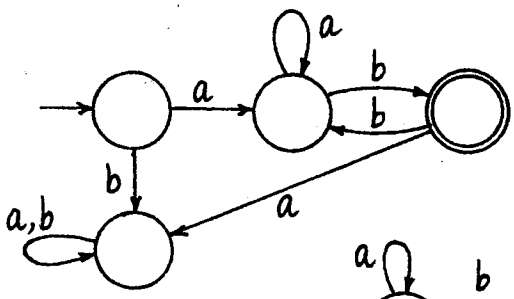
5.3. According to the proof of Theorem 3.3, the language $pal \subseteq \{0,1\}^*$ is an example.

5.4. It is enough to show that any two elements of S are equivalent, and no element of S is equivalent to any element of S'. If $x, y \in S$, then neither is a prefix of any element of L, and therefore xI_Ly. If $x \in S$, $y \notin S$, and z is a string for which $yz \in L$, then z distinguishes x from y.

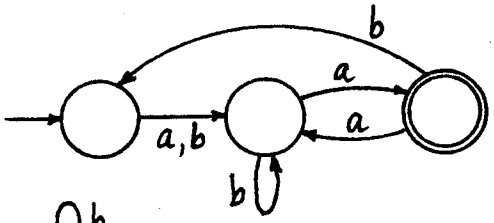
5.5. Suppose $xI_L\Lambda$ and $x \neq \Lambda$. Then for any $i \geq 1$, $x^{i+1}I_Lx^i$. (For any y, consider $x^iy = \Lambda(x^iy)$ and $x^{i+1}y = x(x^iy)$; since ΛI_Lx, either both these strings are in L or neither is.) Since $[\Lambda]$ contains all the strings x^i, it must be infinite. (Note: the same argument shows that if there are strings x and xy for which xI_Lxy and $y \neq \Lambda$, then $[x]$ is infinite.)

5.6. If there were some string x that were not a prefix of any element of L, then for every y, xy would also not be a prefix of any element of L, and so the set of strings that are not prefixes of elements of L would be infinite. By Exercise 5.4, however, this set is one of the equivalence classes. Therefore, if all equivalence classes are finite, there cannot be any such x.

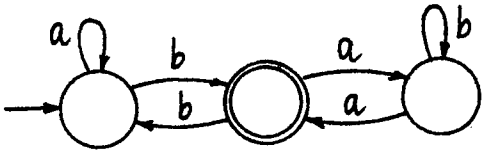
5.7.



5.8.

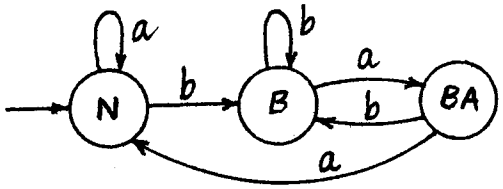


5.9.



5.10. Denote the three sets by B, BA, and N, respectively, and let $Q = \{B, BA, N\}$. If we consider an FA with state set Q, the picture looks like the one below except that the picture below doesn't indicate accepting states. There are eight different FAs, accepting eight different languages, obtained by making all possible choices of $A \subseteq Q$. However, the question asks only for languages L for which I_L has three equivalence classes, and some of the eight languages can be accepted by FAs with fewer than three states. Obvious examples are the languages $\emptyset$ and $\{a,b\}^*$, obtained by making none or all states accepting. The two

other examples are $\{a,b\}^*\{b\}$ and its complement, obtained by making B either the only accepting state or the only nonaccepting state, because both B and B' can be accepted by 2-state FAs.



There are four remaining languages, all of which have the desired property: BA, N, $B \cup BA$, and $B \cup N$.

5.11. Denote the three sets by AB, A, and B, respectively. By an argument similar to that in Exercise 5.10, there are four such languages: B, A, $B \cup AB$, and $A \cup AB$.

5.12. The partition is the same. The sets $\{\Lambda\}$ and $\{0^i1^i \mid i > 0\}$ are both subsets in the partition, just as before. The only difference is that before these were $\{\Lambda\}$ and L, now they are $\{\Lambda\}$ and $L - \{\Lambda\}$.

5.13. (a) and (b) Consider a string $x \in \{0,1\}^*$, and let $k = n_0(x) - n_1(x)$. Saying that $k = 0$ is the same as saying that $x \in L$. If $k > 0$, then x has an excess of k 0's, so that for any z, $xz \in L$ if and only if z has an excess of k 1's. Similarly, if $k < 0$, $xz \in L$ if and only if z has an excess of $-k$ 0's. This means that if $n_0(x) - n_1(x) = n_0(y) - n_1(y)$, then x and y have the property that xz and yz will be in L for precisely the same strings z—i.e., x and y are equivalent. Conversely, if $n_0(x) - n_1(x) \neq n_0(y) - n_1(y)$, then any string z for which $n_1(z) - n_0(z) = n_0(x) - n_1(x)$ distinguishes x from y.

(c) Parts (a) and (b) say that an equivalence class containing a string x is determined by the difference $n_0(x) - n_1(x)$: another string y will be in this equivalence class precisely if $n_0(y) - n_1(y)$ is the same value as for x. This means that for every integer k, there is an equivalence class containing all those strings x for which the difference is k. Another way to say this is to say that the equivalence classes are

$$\ldots, [111], [11], [1], [\Lambda] = L, [0], [00], [000], \ldots$$

For each $k > 0$, $[1^k] = \{x \mid n_1(x) - n_0(x) = k\}$, and $[0^k] = \{x \mid n_0(x) - n_1(x) = k\}$.

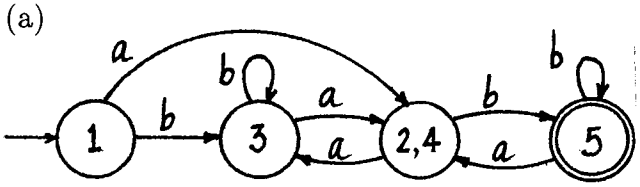
5.14. An easy induction proof shows that for every $q \in Q_1$ and every $x \in \Sigma^*$, $\delta^*(q,x) \in Q_1$. Now let $p, q \in Q_1$ and $x \in \Sigma^*$, and consider the states $\delta^*(p,x)$ and $\delta^*(q,x)$. In (a) we can say that neither is in A, and in (b) we can say that both are in A.

5.15. (a) For x and y to be distinguishable with respect to $\{0,1\}^*\{010\}$, one of the two strings must end with a longer prefix of 010 than the other. In this case, if the longer prefix is α, and $\alpha\beta = 010$, the string β distinguishes the two strings. Therefore, the longest string that might be necessary is one of length 2. (Another way to say it is that if a string of length 3 was required to distinguish x and y, they wouldn't be distinguishable at all, since adding a string of length 3 or more would have the same effect in both cases.)

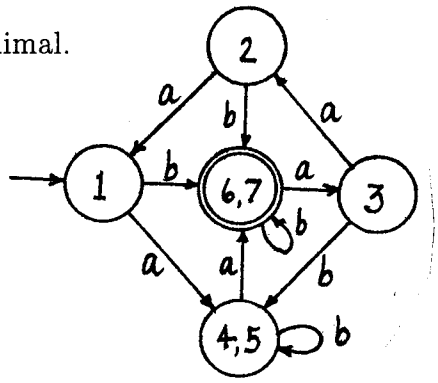
(b) If x has i more left parentheses than right, and y has j more left than right, then saying x and y are distinguishable with respect to L means that $i \neq j$. If k is the smaller of the two numbers i and j, a string of k right parentheses is the shortest string distinguishing x and y. Since i can be no larger than m, and j no larger than n, the smaller of i and j can be no larger than the smaller of m and n.

5.16. The FAs in parts (b), (c), and (f) are already minimal.

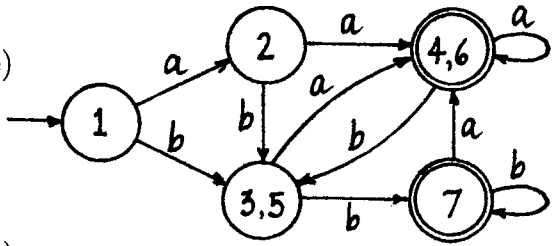
(a)



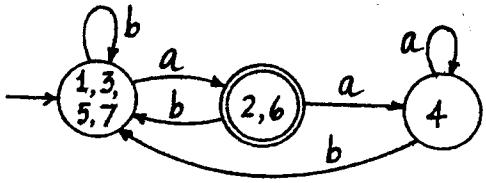
(d)



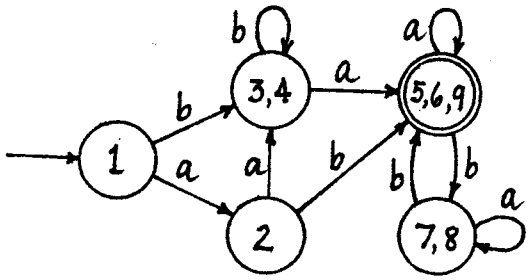
(e)



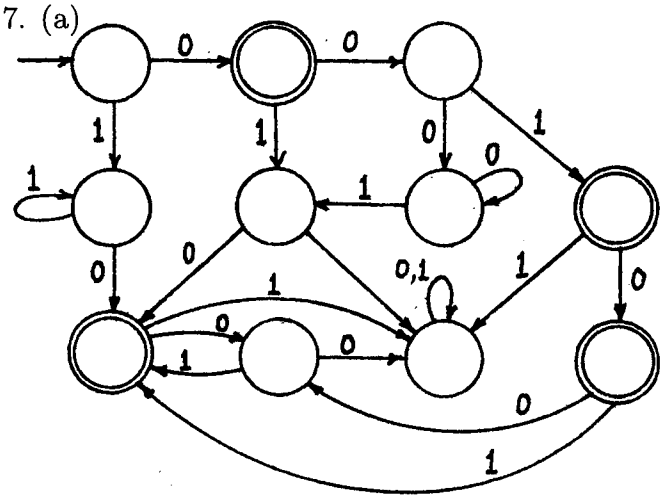
(g)



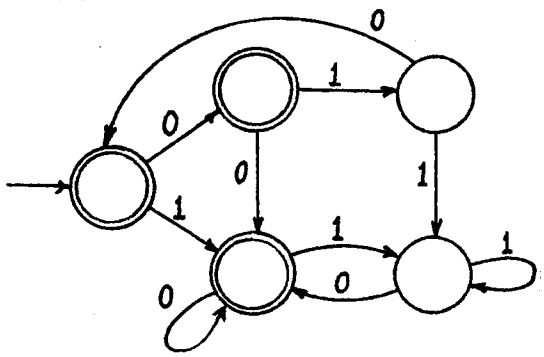
(h)



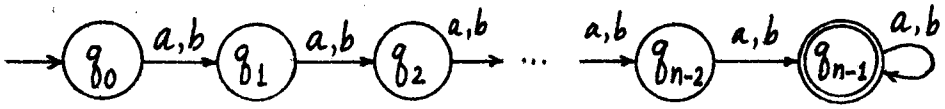
5.17. (a)



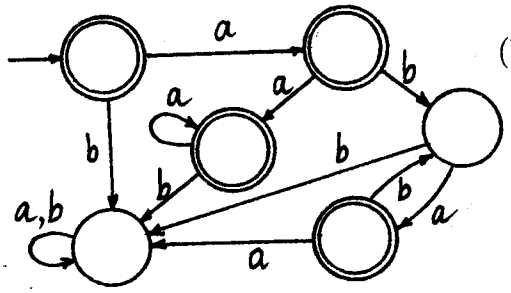
(b)

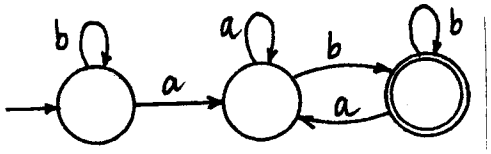


5.18. (a) Suppose the FA has n states. For the FA shown below, the only pair marked on the first pass is (q_{n-2}, q_{n-1}). If starting with pass 2 the pairs are considered in the order (q_0, q_1), (q_1, q_2), $\ldots$, (q_{n-3}, q_{n-2}), then one pair is marked on each pass. The total number of passes is $n - 1$, and no order would require more.



(b) For a pair (p, q) that is ultimately marked, let $n_{(p,q)}$ be the length of the shortest string z for which $\delta^*(p, z) \in A$ and $\delta^*(q, z) \notin A$ (or vice versa). If we process the pairs in nondecreasing order of $n_{(p,q)}$ (i.e., first all the pairs (p, q) with $n_{(p,q)} = 1$, then all the pairs (p, q) with $n_{(p,q)} = 2$, and so forth), then every pair that will ever be marked will be marked on the second pass.

5.19. (a) 

(b) 

5.20. Assume in each part that $0 \le i < j$.

(a) The string 10^{2i} distinguishes 0^i and 0^j.

(b) The string 10^{i+2} distinguishes 0^i and 0^j.

(c) The string 1^{2j} distinguishes 0^i and 0^j.

(d) The string 01^{i+1} distinguishes 0^i and 0^j.

(e) If i is odd, say $i = 2p + 1$, the string 1^{p+1} distinguishes 0^i and 0^j; if i is even, say $i = 2p$, the string 01^{p+1} distinguishes 0^i and 0^j.

(f) The string 1^j distinguishes 0^i and 0^j.

5.21. Suppose L is regular. Then by Theorem 5.3, there are strings u, v, and w so that $|v| > 0$ and $uv^m w \in L$ for every $m \ge 0$. However, this is impossible, because if $v = 0^k$ or $v = 1^k$, then for sufficiently large m the string $uv^m w$ will have more of one symbol than the other; and if v contains both 0's and 1's, then uv^2w contains 1's before 0's. In both cases, therefore, $uv^m w \notin L$.

5.22. Consider the two strings ab^j and ab^k, where j and k are any numbers satisfying $0 \le j < k$. Then the string c^j distinguishes the two strings. This means that all the equivalence classes $[ab^j]$, where $j \ge 0$, are distinct, so that by Corollary 5.1 L cannot be regular.

5.23. (a) $L = \{0^n 10^{2n} \mid n \ge 0\}$. Suppose L is regular, and let n be the integer in the statement of the pumping lemma. Let $x = 0^n 10^{2n}$. Then $x \in L$, and $|x| \ge n$. Therefore, by the statement of the pumping lemma, $x = uvw$ for some strings u, v, and w satisfying $|uv| \le n$, $|v| > 0$, and $uv^m w \in L$ for every $m \ge 0$. The first of these three conditions implies that v is a string of 0's from the first group of 0's in x, and the second implies that $v \ne \Lambda$. Therefore, for some $j > 0$, $uv^2w = 0^{n+j}10^{2n}$. However, the third condition says that uv^2w must be in L. This contradiction proves that L cannot be regular.

(b) $L = \{0^i 1^j 0^k \mid k > i + j\}$. Suppose L is regular, let n be the integer in the pumping lemma, and let $x = 0^n 1^n 0^{2n+1}$. Then $|x| \ge n$ and $x \in L$. Therefore, $x = uvw$ for some u, v, w satisfying the same three conditions as in the first part. As before, v must be 0^j for

some $j > 0$, and the 0's making up v come from the first group of 0's in x. Thus $uv^2w = 0^{n+j}1^n0^{2n+1}$, and this cannot be in L because $2n+j \geq 2n+1$. Again we have a contradiction, proving that L is not regular.

(c) $L = \{0^i1^j \mid j = i \text{ or } j = 2i\}$. Suppose L is regular, let n be the integer in the pumping lemma, and let $x = 0^n1^n$. Then $x = uvw$ for some u, v, and w satisfying the usual three conditions. As before, v must be 0^j for some $j > 0$. Therefore, $uv^2w = 0^{n+j}1^n$, and this string cannot be in L because n is neither $n + j$ nor $2(n + j)$. This contradiction implies that L is not regular.

(d) $L = \{0^i1^j \mid j \text{ is a multiple of } i\}$. Suppose L is regular, let n be the integer in the pumping lemma, and let $x = 0^n1^n$. Then $x = uvw$ for some u, v, and w satisfying the usual three conditions. As before, v must be 0^j for some $j > 0$, and $uv^2w = 0^{n+j}1^n$. Since n cannot be a multiple of $n + j$, this is a contradiction.

(e) $L = \{x \in \{0,1\}^* \mid n_0(x) < 2n_1(x)\}$. Suppose L is regular, let n be the integer in the pumping lemma, and let $x = 0^{2n-1}1^n$. Then $x = uvw$ for some u, v, and w satisfying the usual three conditions. As before, v must be 0^j for some $j > 0$, and $uv^2w = 0^{2n+j-1}1^n$. Since $j - 1 \geq 0$, $2n + j - 1 \geq 2n$, and this contradiction implies that L is not regular.

(f) $L = \{x \in \{0,1\}^* \mid \text{no prefix of } x \text{ has more 1's than 0's}\}$. Suppose L is regular, let n be the integer in the pumping lemma, and let $x = 0^n1^n$. Then $x = uvw$, where u, v, and w satisfy the usual three conditions. As above, $v = 0^j$ for some $j > 0$. Therefore, $uv^0w = uw = 0^{n-j}1^n$, and this string cannot be in L because the string itself (which is a prefix of itself) has more 1's than 0's.

5.24. (a) Suppose L is regular, let n be the integer in the pumping lemma, and let $x = 0^n1^n0^n1^n$. Then $x = uvw$, where the usual three conditions on u, v, and w hold. Then $v = 0^j$ for some $j > 0$, and $z = uv^2w = 0^{n+j}1^n0^n1^n$. According to the pumping lemma, $z \in L$. However, $4n < |z| \leq 5n$, so that $2n < |z|/2 \leq 5n/2$. This means that even if $|z|$ is even, its midpoint is somewhere in the first string of 1's. However, this implies that the second half of z begins with 1. Since z itself begins with 0, z cannot be ww for any w, and this contradiction implies that L is not regular.

(b) $L = \{xy \mid x, y \in \{0,1\}^* \text{ and } y \text{ is either } x \text{ or } x^r\}$. Suppose L is regular, let n be the integer in the pumping lemma, and let $x = 0^n1^n0^n1^n$. Then $x = uvw$, where u, v, and w satisfy the usual three conditions. v must be 0^j for some j, and the 0's in v come from the first part of x. Therefore, $uv^2w = 0^{n+j}1^n0^n1^n$. The length of this string is $4n + j$. If j is odd we have a contradiction immediately, since all strings in L have even length; otherwise, the midpoint is between two of the 1's in the first group of 1's. This means that $0^{n+j}1^n0^n1^n$ cannot be of the form zz, since the first half starts with 0 and the second with 1. This string cannot be of the form zz^r either, because its initial and final symbols are different. This contradiction proves that L is not regular.

(c) Suppose L is regular, let n be the integer in the pumping lemma, and let $x = ({}^n a)^n$ (that is, a preceded by n left parentheses and followed by n right parentheses). Then $x = uvw$, where the usual three conditions hold. It follows that $v = ({}^j$ for some $j > 0$, so that $uv^2w = ({}^{n+j}a)^n$. This string is not in L, and this contradiction implies that L is not regular.

5.25. No. Let L be the language of nonpalindromes over $\{0,1\}$. L is not regular because its complement is not. However, if x begins with 0, $x1 \in L$; if x begins with 1, $x0 \in L$; and if $x = \Lambda$, $x01 \in L$. Therefore, L satisfies the condition for $k = 2$.

5.26. (a) False. Σ^* has a nonregular subset.

(b) False. Nonregular languages have finite subsets, and finite languages are regular.

(c) False. The union of any language and its complement is Σ^*, which is regular.

(d) False. The intersection of a nonregular language and its complement is empty, and the empty language is regular.

(e) True. The complement of a regular language is regular.

(f) False. L_2 could be a subset of L_1, for example.

(g) True. Since $L_1 \cap L_2$ is regular, so is $L_1 - (L_1 \cap L_2) = L_1 - L_2$. Now $L_2 = (L_1 \cup L_2) - (L_1 - L_2)$. Therefore, if $L_1 \cup L_2$ were regular, L_2 would also be regular.

(h) False. L_1 could be Σ^*, for example.

(i) False. Any language is the union of one-element languages; for example, $\{0^n1^n \mid n \geq 0\} = \{\Lambda\} \cup \{01\} \cup \{0011\} \cup \ldots$.

(j) False. Here is a counterexample. For each $k \geq 1$, let $S_k = \{0^{2^k i}1^{2^k i} \mid i \geq 0\}$. Thus $S_1 = \{\Lambda, 0^21^2, 0^41^4, 0^61^6, \ldots\}$, $S_2 = \{\Lambda, 0^41^4, 0^81^8, \ldots\}$, and so forth. Then $S_{k+1} \subseteq S_k$ for each k. The set $\cap_{k=0}^{\infty} S_k$ is $\{\Lambda\}$, which is regular, but it is easy to show that for every k, S_k is nonregular. Now let $L_k = S_k'$. It follows that L_k is nonregular and $L_k \subseteq L_{k+1}$, but $\cup_{k=1}^{\infty} L_k = (\cap_{k=1}^{\infty} S_k)' = \Sigma^* - \{\Lambda\}$, and this set is regular.

5.27. (a) Nonregular. For $i \geq 0$, let $x_i = 01^i$. Then for any i and j with $i < j$, the string 01^i distinguishes x_i and x_j, since 01^j01^i has no nonnull prefix of the form ww.

(b) Regular. For any x, $x \in L$ if and only if x contains one of the substrings 00, 11, 0101, or 1010. (Reason: if x contains neither 00 nor 11, then x must consist of alternating 1's and 0's; therefore, if x contains a nonnull substring ww, it must contain either 0101 or 1010.)

(c) Nonregular. This can easily be proved using the pumping lemma, starting with a string of the form 1^n01^n.

(d) Nonregular. Let $S = \{0^{2i} \mid i \geq 0\}$. Then if $0 \leq i < j$, the strings 0^{2i} and 0^{2j} are distinguished by 1^{2i}, because the middle two symbols of $0^{2i}1^{2i}$ are unequal and the middle two symbols of $0^{2j}1^{2i}$ are both 0. S is therefore an infinite set, any two elements of which are distinguishable with respect to L.

(e) Nonregular. Suppose L is regular, and let n be the integer in the pumping lemma. Let $x = 0^n1^n0^n1^n = (0^n1^n)\Lambda(0^n1^n)$. Then $x = uvw$, where u, v, and w satisfy the usual three conditions. As usual, v must consist of one or more 0's from the first group of 0's, say $v = 0^j$. Then $uv^2w = 0^{n+j}1^n0^n1^n$, and it is easy to see that this string is not of the form xyx for any x with $|x| \geq 1$.

(f) Nonregular, since the set of palindromes is nonregular.

(g) Nonregular. Let $S = \{001^i \mid i \geq 0\}$. For any i and j with $0 \leq i < j$, the two elements 001^i and 001^j of S are distinguished by the string 01^i00.

(h) Nonregular. First we observe that if L were regular, then $\{a^{n^2} \mid n \geq 0\}$ would be

also. Now we can use Theorem 5.4. If the set of perfect squares contained an arithmetic progression, then there would be integers p and q, with $q > 0$, so that $p + iq$ was a perfect square for every $i \geq 0$. In particular, two perfect squares could be found that were as large as we liked but differed by exactly q. However, $(n+1)^2 - n^2 = 2n + 1$, which means that if two distinct perfect squares are both $\geq q^2$, their difference is at least $2q + 1$.

5.28. (a) We have an algorithm to construct an FA accepting the complement of the language accepted by a given FA. Apply this to find M_1' and M_2' accepting $L(M_1)'$ and $L(M_2)'$, respectively. Now construct the FA accepting $L(M_1') \cap L(M_2)'$; finally, apply the algorithm on page 187 to determine whether this FA accepts any strings.

(c) On page 187 there is an algorithm to determine the states reachable from the initial state q_0 of an FA. This algorithm can be easily modified so that it determines the states reachable from q using nonnull strings. (The set T_0 is initialized to be $\emptyset$ instead of q.) The problem is then reduced to determining whether q is an element of this set.

(g) Apply the minimization algorithm described in Section 5.2 to obtain a minimum-state FA $M_1 = (Q_1, \Sigma, q_1, A_1, \delta)$ accepting L. x and y are distinguishable with respect to L if and only if $\delta_1^*(q_1, x) \neq \delta_1^*(q_1, y)$.

(h) If $\delta^*(q_0, x) = q$, then x is a prefix of an element of L if and only if the set of states reachable from q includes at least one accepting state.

(j) The string x is a substring of an element of L if and only if there is a pair (p, q) of states in M satisfying these three conditions: i) p is reachable from q_0, the initial state; ii) $\delta^*(p, x) = q$; iii) the set of states reachable from q includes an accepting state. By testing each pair (p, q) if necessary, we can determine whether there is such a pair.

5.29. One example is the language L of all strings having equal numbers of 0's and 1's. L is nonregular and $L^* = L$.

5.30. One example is *pal*, the set of palindromes. For this L, $L^* = \{0, 1\}^*$, since Λ and all strings of length 1 are palindromes.

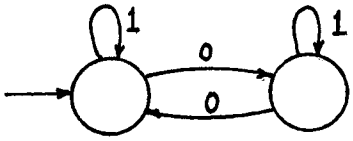
5.31. Suppose x and y are distinguishable with respect to L_1, and specifically that for some z, $xz \in L_1$ and $yz \notin L_1$. Then for some w, $xzw \in L$, since xz is a prefix of an element of L; and $yzw \notin L$, since yz is not a prefix of an element of L. Therefore, if x and y are in different equivalence classes with respect to I_{L_1}, they are in different equivalence classes with respect to I_L. This means that the partition determined by I_L is at least as fine as that determined by I_{L_1}: Any equivalence class with respect to I_{L_1} is the union of equivalence classes with respect to I_L.

5.32. One obvious way to approach this problem is to think about two-state FAs. If there are exactly two equivalence classes, then both states must be reachable from the initial state, and there must be exactly one accepting state. To answer (a), we don't need to worry about which is the accepting state; we only need to draw the transitions in every possible way. There are three ways of drawing transitions from the initial state, since the other state must be reachable, and there are four ways of drawing transitions from the

other state. Each of these twelve combinations gives a different set of strings corresponding to the initial state, and each of these sets is a possible answer to (a). They are described as follows, using regular expressions:

$$\begin{array}{ccc} \Lambda & 1^* & 0^* \\ ((0+1)0^*1)^* & (1+00^*1)^* & (0+10^*1)^* \\ ((0+1)1^*0)^* & (1+01^*0)^* & (0+11^*0)^* \\ (00+01+10+11)^* & (1+00+01)^* & (0+10+11)^* \end{array}$$

Each column represents a possible way of drawing transitions from the initial state, and each row represents a possible way of drawing transitions from the other state. For example, the entry in the second column, third row is obtained by drawing the FA shown here:



(b) For each of these twelve sets, there are two languages L for which the equivalence class $[\Lambda]$ is this set: one obtained by designating the initial state as the accepting state, the other obtained by designating the other state as the accepting state. In other words, the set could be either the language or the complement of the language.

5.33. It follows from Exercise 3.45 that the equivalence classes are all of the form $\{x\}$, where $x \in \{0,1\}^*$.

5.34. According to Exercise 5.4, one equivalence class is the set of all strings that are not prefixes of any element of L. These are the strings having a prefix with more right parentheses than left. Of the strings that are prefixes of balanced strings of parentheses, the equivalence class is determined by the number of excess left parentheses. (If x_1 and x_2 are both prefixes of balanced strings, and x_1 has k more left parentheses than right, and x_2 has j more left parentheses than right, then x_1 and x_2 are equivalent if $k = j$, and otherwise the string $)^k$ distinguishes x_1 and x_2 relative to L.)

This means that the equivalence classes are N, $L = [\Lambda] = [(^0]$, $[(]$, $[(^2]$, $[(^3]$, ..., where N is the set of nonprefixes of elements of L and $[(^k]$ is the set of all strings of parentheses that are prefixes of elements of L and have k more left parentheses than right.

5.35. (a) $\{\Lambda\}$, L, and the set of strings that are not prefixes of any element of L are three of the equivalence classes. We can describe the general form of all the other equivalence classes by considering a specific example. Consider a string that is a prefix of an element of L, in which there are 3 unmatched left parentheses (parentheses without the matching right parentheses). Then there are sixteen "archetypal" pairwise inequivalent strings of this type: $(((, (((i, (((i+, (((i+i, ((i+(, ((i+(i, ((i+(i+, ((i+(i+i, (i+((, (i+((i, (i+((i+, (i+((i+i, (i+(i+(, (i+(i+(i, (i+(i+(i+, and $(i+(i+(i+i$. It is not hard to see that any two of these are distinguishable with respect to L. (For example, consider the two strings $x = ((i+(i$ and $y = (i+((i+i$. Let $z = +i)) + i)$. Then $xz \in L$ and $yz \notin L$.) Furthermore, every other string that is a prefix of an element of L and that has exactly

three unmatched left parentheses is equivalent to one of these, and looks like one of these except that one or more of the i's may be replaced by other elements of L. For example, a string in the same equivalence class as $((i + (i+$ is $(((i + i) + (((i + i) + i)+$. (The first i has been replaced by $(i + i)$ and the second by $((i + i) + i)$.)

We can enumerate these sixteen as follows. For each of the first two unmatched left parentheses, there are two possibilities: either it is followed immediately by another unmatched left parenthesis, or it is followed immediately by an element of L and a $+$. For the third unmatched left parenthesis, there are four possibilities: there is nothing after it; it is followed by an element of L and nothing else; it is followed by an element of L and a $+$; or it is followed by an element of L, a $+$, and another element of L. Altogether, then, there are $2 * 2 * 4 = 16$ combinations.

In general, for each $n > 0$, there are $2^{n-1} * 4$ equivalence classes, which can be enumerated by considering the 2 possibilities for each of the first $n - 1$ unmatched parentheses and the 4 possibilities for the last.

(b) If the expressions are not required to be "fully parenthesized", it is much easier to describe the equivalence classes. The first three are the same as in (a). In addition, for each $n > 0$, there are two equivalence classes corresponding to n: the set of all strings with n unmatched left parentheses in which the last unmatched parenthesis is followed either by nothing or by an element of L and a $+$; and the set of all strings with n unmatched left parentheses in which the last unmatched parenthesis is followed by an element of L. We can see by looking at the sixteen strings in (a) why there are fewer equivalence classes in this case: the strings $(i + ((i + i$ and $(((i$, for example, are equivalent, because of the additional options regarding parentheses.

5.36. (a) Every equivalence class has exactly one element, which means that any two distinct strings are distinguishable with respect to L. The proof is very similar to the proof of Exercise 3.45.

(b) Every equivalence class has exactly two elements, except the one containing Λ, which has only one. For any $x \neq \Lambda$, x and $x^{\sim}$ are equivalent, but no other string is equivalent to either of these.

5.37. We show that any two distinct strings are distinguishable with respect to L, so that every equivalence class has exactly one element. Suppose $x, y \in \{0,1\}^*$ and $x \neq y$. We denote $n_0(x)$ by i, $n_0(y)$ by j, $n_1(x)$ by $i + p$, and $n_1(y)$ by $j + q$.

We consider two cases. First, suppose $p = q$—i.e., x has the same excess of one symbol as y. Then since $x \neq y$, the two numbers i and j are different. In this case we choose z containing N 0's and $(2N + i - p)$ 1's, where N is yet to be described. Then $n_0(xz) = i + N$, and $n_1(xz) = i + p + 2N + i - p = 2(N + i)$, so that $n_1(xz)/n_0(xz) = 2$. On the other hand,

$$\frac{n_1(yz)}{n_0(yz)} = \frac{j + p + 2N + i - p}{j + N} = \frac{2 + \frac{i+j}{N}}{1 + \frac{j}{N}}$$

Now it is clear that since $i + j \neq 2j$, N can be chosen large enough so that this fraction is not an integer. This means that $xz \in L$ and $yz \notin L$.

In the second case, when $p \neq q$, we choose z containing N 0's and $(N-p)$ 1's. Then xz has $(i+N)$ 0's and $i+p+N-p=(i+N)$ 1's, while yz has $(j+N)$ 0's and $(j+q+N-p)$ 1's. It is easy to see that since $q-p \neq 0$, then by choosing N sufficiently large we can guarantee that $(j+q+N-p)/(j+N)$ is not an integer, so that $xz \in L$ and $yz \notin L$.

5.38. Suppose R is a right invariant equivalence relation on Σ^* so that the set of equivalence classes of R is finite and L is the union of some of the equivalence classes. If xRy, then for any z, $xzRyz$, since R is right invariant. This means that xz and yz are in the same equivalence class with respect to R. But since L is the union of some of the equivalence classes, any equivalence class that intersects L must be completely within L; therefore, for any z, $xz \in L$ if and only if $yz \in L$. Therefore, xI_Ly. This means that the partition of Σ^* determined by R is finer than that determined by I_L, which means that every equivalence class with respect to I_L is the union of equivalence classes with respect to R. Since the number of equivalence classes with respect to R is finite, the number of equivalence classes with respect to I_L must be finite.

5.39. (a) Suppose on the one hand that there is a right invariant partition of $\{0,1\}^*$, so that S is one of the subsets in the partition. Right invariant means that whenever x and y are in the same subset, then for any z, xz and yz are in the same subset. Therefore, if x and y are both in S, then for any z, xz and yz are in the same subset of the partition, so that if one of the two is in S, the other must be also.

On the other hand, suppose S has the property that for any $x, y \in S$ and any z, xz and yz are either both in S or both not in S. We consider the following sets: first, all the one-element sets $\{x\}$, where no prefix of x is in S; second, the set S; finally, all sets of the form $S\{\alpha\}$, where $S\{\alpha\} \not\subseteq S$ and $S\{\gamma\} \subseteq S$ for every proper prefix γ of α. (We summarize this last condition by saying that α is a *minimal* string so that $S\{\alpha\} \not\subseteq S$.)

A useful observation is that if $x \in S\{\alpha\}$ and $x \in S\{\beta\}$, where $S\{\alpha\} \not\subseteq S$, $S\{\beta\} \not\subseteq S$, and α and β are both minimal strings with this property, then $\alpha = \beta$. To see this, suppose $\alpha \neq \beta$ and $x = s_1\alpha = s_2\beta$, where $s_1, s_2 \in S$. The strings α and β can't be the same length; suppose $|\alpha| > |\beta|$. Then $\alpha = \gamma\beta$, for some γ, so that $s_1\gamma = s_2$. Because of the assumption on S, it follows that $S\{\gamma\} \subseteq S$, so that α can't be minimal.

Now we check that these subsets of Σ^* form a partition. Obviously, if x is a string such that no prefix of x is in S, then x cannot be in S or any of the sets $S\{\alpha\}$. If $S\{\alpha\} \not\subseteq S$, then $S\{\alpha\}$ and S must be disjoint, because of the assumption on S. Finally, as we observed in the previous paragraph, if $S\{\alpha\} \cap S\{\beta\} \neq \emptyset$, $S\{\alpha\} \not\subseteq S$, $S\{\beta\} \not\subseteq S$, and α and β are both minimal, then $\alpha = \beta$.

Finally, we check that this partition is right invariant. Because of the assumption on S, it is sufficient to show that if $x, y \in S\{\alpha\}$, and $xz \in S\{\beta\}$, where α and β are minimal as defined above, then yz is also in $S\{\beta\}$. We know that $x = s_1\alpha$ and $y = s_2\alpha$, for some $s_1, s_2 \in S$; therefore, xz and yz are both in $S\{\alpha z\}$. Let γ be the longest prefix of αz satisfying $S\{\gamma\} \subseteq S$. Then our previous discussion shows that $\alpha z = \gamma\beta$, so that $S\{\alpha z\} \subseteq S\{\beta\}$, and $y \in S\{\beta\}$.

(b) In the case where S is finite, the condition reduces to saying that no element of S is a prefix of any other element of S.

(c) If S is finite and satisfies this condition, let M be the maximum length of elements of S. We can consider the following partition of Σ^*: first, all the one-element sets $\{x\}$, where $|x| \leq M$ and no prefix of x is in S; second, the set S; third, the single set T containing all other strings. This is obviously a finite partition. It is right invariant, because of the assumption on S and the fact that for any $x \in T$ and any string y, $xy \in T$.

(d) If S satisfies the condition in (a), then S is one of the subsets of a finite right invariant partition if and only if S is regular. On the one hand, if P is a finite right invariant partition, then there is an FA M so that the states of M correspond to the subsets in P, each of which is therefore regular. On the other hand, if S is regular, let M be a minimum-state FA accepting S. The subsets L_p, for the states p of M, form a finite right invariant partition, and L is the union of all the L_p's for which p is accepting. However, the condition in A implies (by Lemma 5.2) that $p \equiv q$ for any two accepting states p and q; therefore, since M is minimal, there is only one accepting state. (Of course, at this point, we have several other ways to characterize regular sets.)

5.40. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA accepting L_1. Then for a string $x \in \Sigma^*$, $x \in L_1/L_2$ if and only if there is a string $y \in L_2$ with $xy \in L$; and this is true if and only if there is a string $y \in L_2$ so that $\delta^*(q_0, xy) = \delta^*(\delta^*(q_0, x), y) \in A$. With this in mind, we define $B = \{q \in Q \mid \text{ for some } y \in L_2, \delta^*(q, y) \in A\}$. Then we have observed that for any x, $x \in L_1/L_2$ if and only if $\delta^*(q_0, x) \in B$. In other words, the FA $M' = (Q, \Sigma, q_0, B, \delta)$ accepts the language L_1/L_2.

Notice that the argument does not really use the fact that L_2 is regular. However, if L_2 is not regular, there may be no way to determine exactly which states are in the set B. If L_2 is regular, on the other hand, there is an algorithm to do this. We do not present all the details, but the idea is that for $q \in Q$, we can determine for each $r \in A$ the language $L(q, r) = \{x \in \Sigma^* \mid \delta^*(q, x) = r\}$, and we can then determine whether the intersection $L(q, r) \cap L_2$ is nonempty. (See the proof of Theorem 4.5, and the discussion in Section 5.4.)

5.41. Using the paintcan analogy, we observe that with p distinct primary colors, there are 2^p possible combinations that can be used to create distinct colors, since there are 2^p subsets of a set of p elements. If we want the number of distinct colors to be at least n, then p must satisfy $2^p \geq n$. Therefore, $p \geq \log_2 n$. We conclude that in order to distinguish all possible pairs from among n strings, at least $\log_2 n$ strings are required.

5.42. For $z = a_1 a_2 \ldots a_n$, let $p_i = \delta^*(p, a_1 \ldots a_i)$ and $q_i = \delta^*(q, a_1 \ldots a_i)$, for each i with $1 \leq i \leq n$. We observe that if z has the property we want (that is, exactly one of the two states $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A), and if the pairs (p_i, q_i) and $(p_{i+d} q_{i+d})$ are identical, then we can shorten z by omitting the substring $a_{i+1} a_{i+2} a_{i+d}$, and the shortened string will still have the desired property. Therefore, we may assume that all the pairs (p_i, q_i) are distinct, for $1 \leq i \leq n$, and thus that n is no larger than the maximum number of distinct ordered pairs of states, which is s^2 (where $s = |Q|$).

5.43. If L is regular, and M is an s-state FA accepting L, then any two strings x and y distinguishable with respect to L can be distinguished by a string of length s^2 or less. (We

let $p = \delta^*(q_0, x)$ and $q = \delta^*(q_0, y)$ and use the result of Exercise 5.42.) On the other hand, if there is an n so that any two strings distinguishable with respect to L can be distinguished by a string of length n or less, then L must be regular. (According to Exercise 5.41, using only the finite set S of strings of length $\leq n$, it is impossible to have an infinite set of strings, any two of which can be distinguished by an element of S.)

5.44. As is pointed out in the statement of the problem, each state in either FA corresponds to one of the equivalence classes of I_L. This means that the L_q partition determined by one of the FA's is the same as that determined by the other. So the bijection $i : Q_1 \to Q_2$ is described as follows: if $q \in Q_1$, and $\delta_1^*(q_1, x) = q$, then $i(q) = \delta_2^*(q_2, x)$.

We must show these three things:
(i) $i(q_1) = q_2$; (ii) For any $q \in Q_1$, $q \in A_1$ if and only if $i(q) \in A_2$; and (iii) For any $q \in Q_1$ and any $a \in \Sigma$, $i(\delta_1(q, a)) = \delta_2(i(q), a)$.

Statement (i) follows from our definition of i and the fact that $\delta_1^*(q_1, \Lambda) = q_1 = \delta_2^*(q_2, \Lambda)$. To show (ii), we take a state $q \in A_1$ and a string x such that $\delta_1^*(q_1, x) = q$. Then $\delta_2^*(q_2, x) = i(q) \in A_2$, because the string x is in L (since M_1 accepts L). The argument is also reversible: If $i(q) \in A_2$, then $x \in L$ because M_2 accepts L, so that $q \in A_1$. Finally, for $q \in Q_1$ and $a \in \Sigma$, if $\delta_1^*(q_1, x) = q$, then

$$i(\delta_1(q, a)) = i(\delta_1^*(q_1, xa)) = \delta_2^*(q_2, xa) = \delta_2(\delta_2^*(q_2, x), a) = \delta_2(i(q), a)$$

5.46. (a) This follows immediately from the fact that if M is any FA accepting L relative to L_{i+1}, then M accepts L relative to L_i.

(b) Suppose there is a constant N so that $n_i \leq N$ for every i. Let M_i be an FA with n_i states that accepts L relative to L_i. We may assume that the states of each M_i are elements of the set $\{q_0, q_1, \ldots, q_{N_1}\}$. Therefore, there must be infinitely many i's for which the FAs M_i are all equal, say to M. Then M must accept L relative to L_i for every i; therefore, since $\cup_{i=1}^{\infty} L_i = \Sigma^*$, M accepts L, which is assumed to be nonregular.

5.47. Let n be the number of states in an FA $M = (Q, \Sigma, q_0, A, \delta)$ accepting L, and suppose $x \in L$ and $x = x_1x_2x_3 = x_1(a_1a_2 \ldots a_k)x_3$, where $k \geq n$. Denote by p_i the state $\delta^*(q_0, x_1a_1a_2 \ldots a_i)$ for each i with $0 \leq i \leq k$. Then just as in the proof of the pumping lemma, at least two of the states p_i must be the same; suppose $p_i = p_{i+j}$, where $j > 0$. Then let $u = a_1 \ldots a_i$, $v = a_{i+1} \ldots a_{i+j}$, and $w = a_{i+j+1} \ldots a_k$. We have $x_2 = uvw$ and $|v| > 0$, and the same argument as in the proof of the pumping lemma shows that $x_1uv^mwx_3 \in L$ for every $m \geq 0$.

5.48. The language L in Example 5.11 is such a language. Suppose L is regular, let n be the integer in the statement, and let $x = ab^nc^n$, and let $x_1 = a$, $x_2 = b^n$, and $x_3 = c^n$. The statement says that for some u, v, and w with $|uv| \leq n$ and $|v| > 0$, $b^n = uvw$ and $auv^mwc^n \in L$ for every $m \geq 0$. This would mean that there are strings ab^ic^n in L for which $i \neq n$, which contradicts the definition of L.

5.50. (a) Nonregular. We use the pumping lemma to show L' is nonregular, and it will follow that L is also. Suppose L' is regular, let n be the integer in the pumping lemma, and let x be a string in L' with $|x| \geq n$. (The fact that there is such an x follows from the parenthetical statement in the exercise.) Then $x = uvw$ for some strings u, v, w such that $|v| > 0$ and $uv^iw \in L'$ for every $i \geq 0$. In particular, $uv^3w \in L'$. However, this string contains three consecutive occurrences of v and is therefore an element of L by definition.

(b) Regular. There is an FA recognizing L having 6 states. Five of the states correspond to the five possible values of $n_0(x) - n_1(x)$ between -2 and 2, and the sixth is the "dead" state for strings not in L.

(c) Nonregular. Any two of the strings 1, 1^2, 1^3, ..., say 1^i and 1^j (where $i < j$), can be distinguished with respect to L by the string 0^{i+3}.

(d) Regular. One way to construct an FA is to have states corresponding to the possible pairs ($n_0(x)$ mod 5, $n_1(x)$ mod 5), 25 states in all. The initial state, which is the only accepting state, is $(0, 0)$.

(e) Nonregular. For each $i \geq 1$, let $x_i = 0^{p_i}$, where p_i is the ith prime ($p_1 = 2$, $p_2 = 3$, $p_3 = 5$, ...). If $i \neq j$, then x_i and x_j are distinguished by the string 1^{p_i}.

(f) Regular. Let $M = (Q, \{0, 1\}, q_0, A, \delta)$ be an FA accepting L, and let $M_1 = (Q, \{0, 1\}, q_0, A_1, \delta)$, where A_1 is the set of states q in A for which there is no string z satisfying $\delta^*(q, z) \in A$. Then M_1 accepts Max(L).

(g) Regular. Let $M = (Q, \{0, 1\}, q_0, A, \delta)$ be an FA accepting L, and let $M_1 = (Q, \{0, 1\}, q_0, A_1, \delta)$, where A_1 is the set of states q in A for which there do not exist strings w and z satisfying $|z| > 0$, $\delta^*(q_0, w) \in A$, and $\delta^*(q_0, wz) = q$. In other words, A_1 is the set of states q in A for which no path from q_0 to q reaches an accepting state until the last step. Then M_1 accepts Min(L).

5.51. (b) Notice that this definition of arithmetic progression does not require the integer p to be positive; if it did, every arithmetic progression would be an infinite set, and the result would not be true. Suppose A is accepted by the FA $M = (Q, \{0\}, q_0, F, \delta)$. Since Q is finite, there are integers m and p so that $p > 0$ and $\delta^*(q_0, 0^m) = \delta^*(q_0, 0^{m+p})$. It follows that for every $n \geq m$, $\delta^*(q_0, 0^n) = \delta^*(q_0, 0^{n+p})$. In particular, for every n satisfying $m \leq n < m + p$, if $0^n \in A$, then $0^{n+ip} \in A$ for every $i \geq 0$, and if $0^n \notin A$, then $0^{n+ip} \notin A$ for every $i \geq 0$. If $n_1, n_2, \ldots, n_r$ are the values of n between m and $m + p - 1$ for which $0^n \in A$, and $P_1, \ldots, P_r$ are the corresponding arithmetic progressions (that is, $P_j = \{n_j + ip \mid i \geq 0\}$), then S is the union of the sets $P_1, \ldots, P_r$ and the finite set of all k with $0 \leq k < m$ for which $0^k \in A$. Therefore, S is the union of a finite number of arithmetic progressions.

5.52. (c) We show the result in the case where f is periodic—i.e., where $f(n) = f(n + p)$ for every n. It is sufficient to show that the equivalence relation I_L has only a finite number of equivalence classes, and to do that it is sufficient to find a finite set S of strings so that any string is indistinguishable with respect to L from some element of S. Let M be the maximum value taken by the function f. For each i with $0 \leq i < p$ and each j with $0 \leq j \leq M$, let $x_{i,j} = b^ia^j$, and let $x_0 = a^{M+1}$. Now let x be any string. If $n_a(x) > M$, then xz cannot be in L for any z, and so xI_Lx_0. Otherwise, let $n_b(x) = i$ and $n_a(x) = j$. Clearly $xI_Lb^ia^j$, since the order of the symbols in a string is irrelevant as far as whether

the string is in L. Moreover, since $f(n) = f(n+p)$ for every n, the only significant thing about the number of b's is the the value mod p; therefore, $xI_Lx_{i_1,j}$, where $i_1 = i \bmod p$.

(d) Suppose L is regular, and let n be the integer in the pumping lemma. Consider the string $x = b^m a^{f(m)}$, where m is any integer with $m \geq n$. By the pumping lemma, $x = uvw$, where $|uv| \leq n$, $|v| > 0$, and $uv^iw \in L$ for every $i \geq 0$. Since $m \geq n$, $v = b^p$ for some $p > 0$. This means that each of the strings $b^{m+ip}a^{f(m}$ is in L. It follows from the definition of L that $f(m+ip) = f(m)$ for every $i \geq 0$. To summarize: for every $m \geq n$, there is a number p_m so that $0 < p_m \leq n$ and $f(m+ip) = f(m)$ for every $i \geq 0$. Now there are only a finite number of integers p that can be p_m for some $m \geq n$, since for any such p_m, $p_m \leq n$. Let P be the least common multiple of all of them—i.e., the smallest positive integer that is evenly divisible by all of them. Then for any $m \geq n$, $P = kp_m$ for some k. Therefore, for any $i \geq 0$, $f(m+iP) = f(m+(ik)p_m) = f(m)$. Therefore, f is eventually periodic.

5.53. Let L be the set $\{0^n \mid n > 0$ and n is not prime$\}$. L^2 contains all sufficiently long strings of 0's, because every sufficiently large integer (7 or greater, say) is the sum of two positive nonprimes. (If n is even, n is the sum of two even numbers, each 4 or larger; if n is odd, n is $1 + (n-1)$.) Therefore, L^2 is regular. However, L is not, by Example 5.10.

5.54. Let D be the set of integers that are lengths of strings in L^*. Then D is a subset of $\mathcal{N}$ that is closed under addition. We will show this implies that D is the union of a finite number of arithmetic progressions, and the result we want will follow (see Exercise 5.51).

Let p be the smallest positive integer so that p is an integer combination of elements of D (i.e., of the form $\sum_{i=1}^{k} a_i n_i$, where the a_i's are integers, either positive or negative, and the n_i's are elements of D). We show first that the set of all integer combinations of elements of D is precisely the set of all integer multiples of p. It is clear that every multiple of p has this form. On the other hand, suppose that j is an integer combination of elements of D but not a multiple of p. We can divide j by p, and get a quotient and a remainder: that is, $j = qp + r$, where $0 \leq r < p$. But since j and p are both integer combinations of elements of D, so is r. However, p is defined to be the smallest positive one, so $r = 0$; this contradicts the fact that j is not divisible by p. At this point, we have $p = \sum_{i=1}^{k} a_i n_i$, where each $n_i \in D$; we know that every integer combination of the n_i's is a multiple of p; and we know that every integer combination of the n_i's for which the coefficients are all nonnegative is itself in D (since D is closed under addition). Finally, since n_1 is a multiple of p, let $n_1 = m_1 p$. We can now show that all sufficiently large multiples of p are elements of D. To be specific, let $N = \sum_{i=1}^{n} m_1 |a_i| n_i$. (Except for the extra factor of m_1, this is just the formula for p, but with any negative coefficients replaced by their absolute values.) We show that for every $j \geq 0$, $N + jp \in D$. It is sufficient to show that $N + jp$ is an integer combination of the n_i's, with nonnegative coefficients.

Let us first consider j satisfying $0 \leq j \leq m_1$. For such a j,

$$N + jp = \sum_{i=1}^{n} (m_1 |a_i| + j a_i) n_i$$

and all the coefficients of this sum are nonnegative. This takes care of all multiples of p

with values between N and $N + n_1$. However, it is clear that the same argument will work for the multiples of p between $N + rn_1$ and $N + (r + 1)n_1$, where r is any natural number. We have shown that $\{n \in D \mid n \geq N\} = \{N + jp \mid j \geq 0\}$. We conclude that D is the union of a finite set and an arithmetic progression, which means that it is the union of a finite number of arithmetic progressions.

Chapter 6

Context-free Languages and Pushdown Automata

6.1. (a) The language of even-length palindromes over $\{a, b\}$.

(b) The language of odd-length palindromes over $\{a, b\}$.

(c) The set of even-length strings x over $\{a, b\}$ such that x^r is obtained from x by reversing a's and b's.

(d) The set of strings over $\{a, b\}$ that are not palindromes but could be made into palindromes by changing one symbol from a to b or vice versa.

(e) and (f) The set $\{a, b\}^*\{a\}$

(g) The set $\{ba\}^*\{b\}$

(h) The set of even-length strings in $\{a, b\}^*$

6.2. $S \to TS \mid \Lambda \qquad T \to a \mid bS \mid cUd \qquad U \to AeU \mid A \qquad A \to f \mid gBi \qquad B \to hjB \mid h$

6.3. One CFG is the one with start symbol P and productions

$$P \to pI; \mid pI(FF_1) \qquad F_1 \to ;FF_1 \mid \Lambda \qquad F \to G \mid vG \mid fG \mid pII_1$$
$$I_1 \to ,II_1 \mid \Lambda \qquad G \to II_1 : I \qquad I \to L \mid LL_2 \qquad L_2 \to L_1L_2 \mid \Lambda$$
$$L_1 \to L \mid D \qquad L \to a \mid b \qquad D \to 0 \mid 1$$

6.4. (a) $S \to aSa \mid aSb \mid bSa \mid bSb \mid a$

(b) $S \to aSa \mid aSb \mid bSa \mid bSb \mid aa \mid bb$

(c) $S \to aTa \mid bUb \qquad T \to aTa \mid aTb \mid bTa \mid bTb \mid a \qquad U \to aUa \mid aUb \mid bUa \mid bUb \mid b$

6.5. (a) If this CFG generated L, then every nonnull element of L would be either of the form $x01y$ or of the form $x10y$, where in either case $x, y \in L$. This is not the case, however, as the string 0011 illustrates.

(b) If this CFG generated L, then every nonnull element of L would have one of the forms $0y1$, $1y0$, $01y$, $10y$, $y01$, or $y10$, where y is an element of L. The string 00111100, however, does not fit any of these patterns.

6.6. No. The string $aabbcc$ cannot be obtained from this CFG. If it could, the first production in any derivation would have to be $S \to aSbScS$, because all the others would result in b's appearing before a's or c's appearing before a's or c's appearing before b's. The only way $aabbcc$ could be obtained from $aSbScS$ would be for the first S to eventually be replaced by either a or ab, and neither a nor ab has equal numbers of a's, b's, and c's.

6.7. We give an answer for the alphabet $\Sigma = \{a, b\}$, and it can easily be modified for the more general case.

(a) $S \to (S + S) \mid (SS) \mid (S^*) \mid a \mid b \mid \lambda$
(b) $S \to S + S \mid SS \mid S^* \mid (S) \mid a \mid b \mid \lambda$

6.8. None of these works. A counterexample to (a) is $S_1 \to aS_1b \mid \Lambda \qquad S_2 \to b$. The string abb is not in $L(G_1) \cup L(G_2)$, but the grammar obtained by the construction allows it. A counterexample to (b) is $S_1 \to a \qquad S_2 \to b$. The string abb is not in $L(G_1)L(G_2)$, but the construction allows the derivation $S_1 \Rightarrow S_1S_2 \Rightarrow S_1S_2S_2 \Rightarrow^* abb$. A counterexample to (c) is $S_1 \to aS_1a \mid b$. The string $abba$ is not in $L(G_1)^*$, but the grammar obtained by the construction allows it.

6.9. (a) $S \to aSc \mid T \qquad T \to aTb \mid \Lambda$
(b) $S \to AB \qquad A \to aAb \mid \Lambda \qquad B \to bBc \mid \Lambda$
(c) $S \to AT \mid UC \qquad A \to aA \mid \Lambda \qquad C \to cC \mid \Lambda \qquad T \to bTc \mid \Lambda \qquad U \to aUb \mid \Lambda$
(d) $S \to TC \mid U \qquad C \to cC \mid \Lambda \qquad T \to aTb \mid \Lambda \qquad U \to aUc \mid B \qquad B \to bB \mid \Lambda$
(e) $S \to TBC \mid AU \qquad T \to aTb \mid \Lambda \qquad B \to bB \mid b \qquad C \to cC \mid \Lambda$
$A \to aA \mid a \qquad U \to aUc \mid V \qquad V \to bV \mid \Lambda$
(f) $S \to aaSb \mid aSb \mid Sb \mid \Lambda$
(g) $S \to aS_1b \mid S_1b \qquad S_1 \to aaS_1b \mid aS_1b \mid S_1b \mid \Lambda$
(h) $S \to aSb \mid aSbb \mid \Lambda$

6.10 (a) Strings over $\{a, b\}$ containing an even number of a's and an odd number of b's.
(b) One description is the language corresponding to the regular expression $(b+aa^*bb)^*(\Lambda+aa^*)b$.

6.11. Statement (a) obviously implies statement (b). Suppose L can be generated by a grammar G in which all productions have either the form $A \to xB$ or the form $A \to x$, where $x \in \Sigma^+$. Construct the grammar G_1 as follows. All the productions $A \to aB$ or $A \to a$ in G (where A, B are variables and $a \in \Sigma$) are also in G_1. For each production $A \to xB$ in G with $|x| > 1$, say $x = a_1a_2 \ldots a_k$, put the productions $A \to a_1A_1$, $A_1 \to a_2A_2$, $\ldots$, $A_{k-2} \to a_{k-1}A_{k-1}$, $A_{k-1} \to a_kB$ in G_1. Here the variables A_1, A_2, $\ldots$, A_{k-1} are new variables that are used nowhere else. Similarly, for each production $A \to x$ in G with $x = a_1a_2 \ldots a_k$ where $k > 1$, put the productions $A \to a_1A_1$, $A_1 \to a_2A_2$, $\ldots$, $A_{k-2} \to a_{k-1}A_{k-1}$, $A_{k-1} \to a_k$ in G_1. Here also, these new variables are specific to this production and are used nowhere else. G_1 is clearly regular, and it is not hard to see that $L(G_1) = L$.

To show that (a) and (c) are equivalent, we observe that the grammar obtained from the one in (c) by reversing the right sides of the productions has the property in (b) and generates the language L^r. Therefore, L can be generated by a grammar of the type in (c) if and only if L^r is regular. But this is true if and only if L is regular.

6.13. $A \to aB \qquad B \to aB \mid bC \mid b \qquad C \to cC \mid aB \mid b$

6.16. The "only if" part is trivial. The converse is false: the CFG with productions

$S \to aT \mid \Lambda$ and $T \to Sb$ is equivalent to the one with productions $S \to aSb \mid \Lambda$, and this grammar generates the nonregular language $\{a^n b^n \mid n \geq 0\}$.

6.17. If L is regular, then $L - \{\Lambda\}$ can be generated by a grammar G that is not self-embedding, by Theorem 6.2. If $\Lambda \in L$, we can generate L by the grammar G_1, whose start symbol is S (not in G) and whose productions are those in G as well as the two additional productions $S \to \Lambda \mid S_1$ (where S_1 is the start symbol of G). G_1 is obviously not self-embedding.

6.18. (a) A regular expression is $(aa + aab + aba + abab)^*(a + ab)$, and a regular grammar is

$$S \to aT \mid a \qquad T \to aU \mid bV \mid b \qquad U \to aT \mid a \mid bS$$
$$V \to aW \qquad W \to aT \mid a \mid bS$$

(b) A regular expression is $(a + b)^*ab(ab + b)^+$. A regular grammar is

$$A \to aA \mid bA \mid aB \qquad B \to bC \qquad C \to aD \mid bE$$
$$D \to bE \qquad E \to aD \mid bE \mid \Lambda$$

(c) A regular expression is $(ab + ba)^*(ab + aab)$, and a regular grammar is

$$S \to aT \mid bU \qquad T \to aV \mid bW \mid b$$
$$U \to aS \qquad W \to aT \mid bU \qquad V \to bX \mid b$$

(d) Although A generates the language of odd-length palindromes, which is not regular, every nonnull string begins with an odd-length palindrome. Therefore, the language is $\{a, b\}^+$, and a regular grammar is $S \to aS \mid bS \mid a \mid b$.

(e) A generates all strings with an odd number of a's. S generates the strings that have either a positive even number of a's or no a's—in other words, all strings with an even number of a's. A regular expression is $(b^*ab^*a)^*b^*$, and a regular grammar is the one with productions

$$A \to bA \mid aB \mid \Lambda \qquad B \to bB \mid aA$$

6.19. (c) 5 (d) 14 (e) 1

6.20. Let the productions be $S \to ABA \qquad A \to \Lambda \qquad B \to b$. Then the string B has the derivation $S \Rightarrow ABA \Rightarrow BA \Rightarrow B$, but B has neither a leftmost derivation nor a rightmost derivation.

6.21. (a) If $a = 3$, the resulting value of x is 3, and if $a = 1$, the resulting value of x is 1.

(b) If $a = 3$, the resulting value of x is 1, and if $a = 1$, the resulting value of x is 3.

6.22. The string $abaa$ has two leftmost derivations, one beginning $S \Rightarrow SbS$, the other beginning $S \Rightarrow Sa$.

6.23. False. The string ab has the two derivations $S \Rightarrow AB \Rightarrow aAB \Rightarrow aB \Rightarrow abB \Rightarrow ab$ and $S \Rightarrow AB \Rightarrow B \Rightarrow ab$. If x is a string derivable from AB, then for each way of writing $x = yz$, where $A \Rightarrow^* y$ and $B \Rightarrow^* z$, there is only one way of deriving y from A and only one way of deriving z from B. However, there is more than one choice for writing $x = yz$ in this way.

6.24. All the grammars except those in (f) and (g) are unambiguous. We prove this result for (c). We will show that for every $n \geq 0$, and every x with $|x| = n$, i) if $S \Rightarrow^* x$, x has only one leftmost derivation from S; and ii) if $A \Rightarrow^* x$, x has only one leftmost derivation from A.

For the basis step, it is clear that Λ can be derived only from the variable A, and it has only one derivation from A. Suppose that $k \geq 0$ and that any string of length k or less that can be derived from one of the two variables has only one leftmost derivation from that variable. Now suppose $|x| = k+1$. If $S \Rightarrow^* x$, then it is clear that $k+1 \geq 2$. The first step of a derivation of x from S is determined by the beginning and ending symbols of x. In the first case, where the first step is $S \Rightarrow aSa$, it follows that $x = aya$, where $S \Rightarrow^* y$. By the induction hypothesis, y has only one leftmost derivation from S, and it follows that x has only one leftmost derivation from S. The other three cases are similar.

If $A \Rightarrow^* x$, there are five cases: i) $x = aya$; ii) $x = byb$; iii) $x = a$; iv) $x = b$; v) $x = \Lambda$. In each of these cases, the first step in any derivation is determined. In the last three cases the first step is the only one, and in the first two the induction hypothesis allows us to conclude that x has only one leftmost derivation from A.

(f) Ambiguous. The string aaa has two leftmost derivations.

(g) Ambiguous. The string $babab$ has two leftmost derivations.

6.25. (a) (Example 6.3) Unambiguous. The proof is similar to part (c) of Exercise 6.24.

(b) (Example 6.9) Unambiguous. We will not present a detailed proof, but it is not hard to see that for any string x corresponding to the regular expression, x can be written uniquely as yz, where y corresponds to $(011 + 1)^*$ and z corresponds to $(01)^*$. This might not have been the case: for example, if the regular expression had been $(0101 + 1)^*(01)^*$, the string 0101 could match in two different ways, and the corresponding grammar would be ambiguous.

(c) (Example 6.11) Unambiguous. Again we only sketch the argument. If $x = 0^i1^j0^k$, where $j > i + k$, the first step of the derivation of x is $S \rightarrow ABC$. The string to be derived from A must be 0^i1^i, and this string has only one derivation from A. The string to be derived from C must be 1^k0^k, and it also has only one derivation from C. The string to be derived from B is 1^{j-i-k}, and it also has only one derivation from B.

6.26. (a) The string aaa has two different leftmost derivations. An equivalent unambiguous grammar is $S \rightarrow aS \mid bS \mid \Lambda$.

(b) The string a can be derived $S \Rightarrow ABA \Rightarrow aBA \Rightarrow^* a$ or $S \Rightarrow ABA \Rightarrow BA \Rightarrow A \Rightarrow a$. It's easy to see, however, that any string with at least one b has only one leftmost derivation. Therefore, an equivalent unambiguous grammar is $S \rightarrow A \mid ABA \quad A \rightarrow aA \mid \Lambda \quad B \rightarrow bB \mid b$.

(c) ab has two leftmost derivations, but it's the only string that does. Therefore, an equivalent unambiguous grammar is one that allows ab to be derived only one way, such as $S \to A \mid B \qquad A \to aAb \mid aabb \qquad B \to abB \mid \Lambda$.

(d) The grammar is ambiguous because in deriving $aaabb$, for example, we could use either $S \Rightarrow aSb \Rightarrow aaaSbb \Rightarrow aaabb$ or $S \Rightarrow aaSb \Rightarrow aaaSbb \Rightarrow aaabb$. In other words, we could use the two productions in either order. To make it unambiguous, fix it so that we have to use one first, as many times as necessary. One way to do this is $S \to aaSb \mid T$ $T \to aTb \mid \Lambda$. In this grammar, if we want to end up with 5 more a's than b's, for example, we have to use the first production 5 times and then the second as many times as there are b's.

(e) The ambiguity here seems to be directly related to the Λ-production, since aSb and abS yield the same thing when S is replaced by Λ. We must keep Λ in the language, but aside from that we can eliminate it from the grammar, as follows:

$$S \to T \mid \Lambda \qquad T \to aTb \mid abT \mid ab$$

It is reasonably clear that this generates the same language as the original grammar. We show that the new grammar is unambiguous by showing that for any $n \geq 1$, if x is any string that can be derived from T in n steps, then x has only one derivation from T. The basis step, $n = 1$, is clear. Suppose that $k \geq 1$ and that a string derivable from T in k or fewer steps can be derived in only one way. Let x be a string derivable from T in $k + 1$ steps.

We consider two cases. If $x = aby$ for some y, then any derivation of x from T must begin $T \Rightarrow abT$, and the remainder of the derivation consists of deriving y from T in k steps. By the induction hypothesis, there is only one way to do this; therefore, x has only one derivation from T. Otherwise, $x = ayb$ for some y derivable from T in k steps. Again the induction hypothesis implies that x has only one derivation from T.

6.27. The nonnegative integers that can be expressed in the form $4i + 7j$, where i and j are nonnegative, are 0, 4, 7, 8, 11, 12, 14, 15, 16, and (by Exercise 2.50) all the integers greater than 17. An unambiguous grammar is therefore

$$S \to \Lambda \mid a^4 \mid a^7 \mid a^8 \mid a^{11} \mid a^{12} \mid a^{14} \mid a^{15} \mid a^{16} \mid T \qquad T \to aT \mid a^{18}$$

6.28. The grammar obtained from an FA as in the proof of Theorem 6.2 is always unambiguous. The grammar obtained from an NFA (see Exercise 6.15) may be ambiguous.

6.29. Convert the regular grammar into an NFA, as in the proof of Theorem 6.2. Use the subset construction to find an equivalent FA. Then convert the FA into a regular grammar, as in the proof of Theorem 6.2. It will be unambiguous.

6.30. Consider a left parenthesis $(_0$ in a balanced string x. First, there must be at least one right parenthesis in x after $(_0$. This is true because x has equal numbers of left and right parentheses, and if there were no right parentheses following $(_0$, the prefix of x ending

just before $(_0$ would have more right than left.

Second, of the right parentheses following $(_0$, at least one must have the property that the substring beginning with $(_0$ and ending with this one has equal numbers of left and right parentheses. To see this, consider substrings beginning with $(_0$. The substring of length 1 has more left parentheses than right, and every subsequent symbol changes by 1 the difference between the number of left and the number of right. Therefore, if there were no such right parenthesis, all the substrings beginning with $(_0$ would have more left parentheses than right. In particular, the suffix of x beginning with $(_0$ would, and therefore, the prefix ending just before $(_0$ would have more right than left.

Let $)_0$ be the first right parenthesis following $(_0$ such that the substring y beginning with $(_0$ and ending with $)_0$ has equal numbers of left and right parentheses. Then every shorter nonnull prefix of $y$has more left than right. (Otherwise, there would have been a right parenthesis before $)_0$ having the same property as $)_0$.) Therefore, y is balanced.

6.31. First observe that two distinct left parentheses a_1 and a_2 can't have the same mate. Otherwise the string beginning with a_1, the leftmost of the two, and extending up to but not including a_2, would be a balanced string, which would imply that the mate of a_1 was within that string.

Now, suppose there is a left parenthesis a_1 to the right of a so that the string from a_1 to b was balanced. We may assume a_1 is the rightmost left parenthesis having this property. Since the mate b_1 of a_1 can't be b, it must appear before b. The string now looks like axa_1yb_1zb, where the entire string, the substring a_1yb_1, and the substring a_1yb_1zb are all balanced. This implies, however, that the substring zb is also balanced. This substring must start with a left parenthesis, and now we have contradicted the assumption that a_1 is the rightmost left parenthesis for which the string from a_1 to b is balanced.

6.32. $S \to S + T \mid S - T \mid T \qquad T \to T * F \mid T/F \mid F \qquad F \to (S) \mid a$

6.33. First we show that if A is nullable, then $A \Rightarrow^* \Lambda$. Using structural induction, it is sufficient to show that if there is a production $A \to \Lambda$, then $A \Rightarrow^* \Lambda$ (which is obvious), and that if $A \to B_1B_2 \ldots B_n$ is a production and each B_i satisfies $B_i \Rightarrow^* \Lambda$, then $A \Rightarrow^* \Lambda$. This is also obvious.

For the converse, we show that if $A \Rightarrow^* \Lambda$, then A is nullable. $A \Rightarrow^* \Lambda$ means there is a derivation by which Λ is derived from A. The proof is on the number of steps in the derivation. The basis step is easy, since if $A \Rightarrow \Lambda$ then $A \to \Lambda$ must be a production. Suppose that $k \geq 0$ and that any variable from which Λ can be derived in k or fewer steps is nullable. Now suppose we have a derivation of Λ from A in $k + 1$ steps, and let the first step be $A \to B_1B_2 \ldots B_n$. Since $B_1B_2 \ldots B_n \Rightarrow^* \Lambda$, each B_i must be a variable from which Λ can be derived in k or fewer steps. By the induction hypothesis, each B_i is nullable. Therefore, A is nullable by the second part of the definition.

6.34. (a) $S \to AB \qquad A \to aASb \mid aAb \mid a \qquad B \to bS \mid b.$

(b) All the variables are nullable. Following the algorithm, we obtain

$$S \to AB \mid A \mid B \mid ABC \mid AC \mid BC \mid C$$
$$A \to BA \mid B \mid BC \mid C \mid a$$
$$B \to AC \mid A \mid C \mid CB \mid b$$
$$C \to BC \mid B \mid AB \mid A \mid c$$

6.35. (a) $S \to ABA \mid AB \mid BA \mid AA \mid aA \mid a \mid bB \mid b \qquad A \to aA \mid a \qquad B \to bB \mid b$
(c)

$$S \to aAa \mid bB \mid bb \mid aCaa \mid baD \mid abD \mid aa$$
$$A \to aAa \mid bB \mid bb \qquad B \to bB \mid bb$$
$$C \to aCaa \mid baD \mid abD \mid aa \qquad D \to baD \mid abD \mid aa$$

6.36. A definition of live variables is the following: 1. Any variable A for which there is a production $A \to x$, with $x \in \Sigma^*$, is live. 2. Any variable A for which there is a production $A \to \alpha$, where every symbol of α is either a terminal or a live variable, is live.

A corresponding algorithm to find the live variables is the following.

$L_0 = \{A \in V \mid P$ contains a production $A \to x$ with $x \in \Sigma^*\}$;
$i = 0$;
do
    $i = i + 1$;
    $L_i = L_{i-1} \cup \{A \mid P$ contains $A \to \alpha$, where $\alpha \in (\Sigma \cup L_{i-1})^*\}$;
while $L_i \neq L_{i-1}$;
L_i is the set of live variables.

6.37. A definition of reachable variables is the following: 1. S is reachable. 2. If A is reachable, and P contains a production $A \to \alpha B \beta$, where $B \in V$, B is reachable.

A corresponding algorithm to find the reachable variables is the following.

$R_0 = \{S\}$;
$i = 0$;
do
    $i = i + 1$;
    $R_i = R_{i-1} \cup \{B \in V \mid P$ contains $A \to \alpha B \beta$, where $A \in R_{i-1}\}$;
while $R_i \neq R_{i-1}$;
R_i is the set of reachable variables.

6.38. (a) Consider the grammar with productions $S \to AB \qquad A \to a$. A is live, since $A \to a$ is a production, and A is reachable, since $S \to AB$ is a production, but A is

obviously useless, since no strings can be derived from this grammar.

(b) We know that $L(G_1) = L(G)$, because the only productions left out of G_1 are those containing variables of G from which no strings of terminals can be derived in G. Similarly, $L(G_1) = L(G_2)$, since in constructing G_2 from G_1 the only productions omitted are those containing variables never obtained in any derivation of G_1 beginning with S.

We wish to show every variable in G_2 is useful. Let A be such a variable. A is reachable in G_1; i.e., $S \Rightarrow^* \alpha A \beta$ in G_1. This means that not only A, but any variables in α and β as well, are reachable in G_1. Therefore, this is a legitimate derivation in G_1, since any production in G_1 that involves only variables in G_1 is still present in G_1. On the other hand, since this is a derivation in G_1, all variables involved (in particular, A, as well as all variables in α and β) are variables in G_1. This means they are live variables in G. Thus there is a derivation $\alpha A \beta \Rightarrow^* x$ in G, for some $x \in \Sigma^*$. Any variable involved in this derivation is also a live variable in G, so all the productions involved are present in G_1; thus this is a legitimate derivation in G_1. In fact, it's a derivation in G_2 as well: $\alpha A \beta$ is derivable from S in G_1, so anything derivable from $\alpha A \beta$ in G_1 is also derivable from S in G_1. Since all variables involved are therefore present in G_2, all the productions involved must also be present in G_2. We have $S \Rightarrow^*_{G_2} \alpha A \beta \Rightarrow^*_{G_2} x$, which shows that A is a useful variable in G_2.

(c) The example in (a) also illustrates this fact. Nothing is eliminated in the first step, and only the first production is eliminated in the second step.

(d)(ii) The live variables are S, A, and B, and the grammar obtained from the first step has the productions

$$S \to AB \qquad A \to aAb \mid bAa \mid a \qquad B \to bbA \mid aaB \mid AB$$

Since all the variables in this grammar are reachable, this is the final result.

6.39. (b) Eliminating Λ-productions yields the productions

$$S \to S(S) \mid (S) \mid S()$$

There are obviously no unit productions. Converting to Chomsky normal form gives

$$\begin{aligned} &S \to SY_1 \qquad Y_1 \to X_(Y_2 \qquad Y_2 \to SX_) \\ &X_(\to (\qquad X_) \to) \\ &S \to X_(Y_2 \mid SY_3 \qquad Y_3 \to X_(X_) \end{aligned}$$

(The variables are S, Y_1, Y_2, Y_3, $X_($, and $X_)$.)

6.40. As we did at the beginning of Section 6.6., we consider the quantity $s = l + t$, the sum of the length of the current string in the derivation and the number of terminal symbols in the string. Initially, when the current string is S, $s = 1$. At the end of the derivation, $s = 2k$. Since the grammar is in Chomsky normal form, s increases by 1 at each step. Therefore, there are exactly $2k - 1$ steps in the derivation.

6.41. Strings over $\{a, b\}$ in which every prefix has at least as many a's as b's. We prove by induction that every string having this property can be derived from the grammar. If $|x| = 0$, x can obviously be derived. Suppose that $k \geq 0$ and that for any string x of length $\leq k$ in which every prefix has at least as many a's as b's, x can be derived. We wish to show that any string x of length $k + 1$ having this property can be derived.

Suppose that $|x| = k + 1$ and every prefix of x has at least as many a's as b's. Then $x = ay$ for some y. If every prefix of y has at least as many a's as b's, then y can be derived from the grammar, by the induction hypothesis; therefore, $x = ay$ can, because we can start the derivation $S \Rightarrow aS$ and continue by deriving y from S.

If y has a prefix with more b's than a's, let y_1 be the smallest such prefix. Then y_1 must end in b; let $y_1 = y_2 b$. The string y_2 has the same number of a's as b's, and every prefix of y_2 has at least as many a's as b's. We can write $x = ay_2bz$ for some string z. Since every prefix of x has at least as many a's as b's, and since ay_2b has equal numbers of a's and b's, every prefix of z must have at least as many a's as b's. Therefore, by the induction hypothesis, $S \Rightarrow^* y_2$ and $S \Rightarrow^* z$. It follows that $S \Rightarrow^* ay_2bz$, since we can start the derivation $S \Rightarrow aSbS$ and continue by deriving y_2 from the first S and z from the second.

If we think of a and b as (and), respectively, this language is the set of all prefixes of balanced strings of parentheses.

6.42. If these two productions were the only ones with variables on the right side, then we might as well assume that S is the only variable (any other variable could never be involved in a derivation from S), and so the only other productions are $S \rightarrow x_1 \mid x_2 \mid \ldots \mid x_n$, where each x_i is an element of $\{a, b\}^*$. In this case, however, no string that begins with a and ends with b and is not one of the x_i's can be derived. Therefore, the grammar could not generate all nonpalindromes.

6.43. The only string of length 1 produced by the grammar is 0, which has more 0's than 1's. Suppose that $k \geq 1$ and that every string of length k or less produced by the grammar has more 0's than 1's. Let x be a string of length $k + 1$ produced by the grammar, and consider a derivation of x. If the first step in the derivation is either $S \Rightarrow S0$ or $S \Rightarrow 0S$, then $x = y0$ or $x = 0y$, where in either case y is a string of length k produced by the grammar. By the induction hypothesis, y has more 0's than 1's, and so x obviously does also. If the first step in the derivation of x is one of the last three productions, then x is the concatenation of three strings, two of which are derivable from the grammar and one of which is 1. By the induction hypothesis, the two strings derivable from the grammar both have more 0's than 1's; therefore, since the two of them together have at least two more 0's than 1's, the result of adding a 1 still has more 0's than 1's.

6.44. We consider a string x satisfying $d(x) > 0$, and we complete the induction step of the proof that if x starts with 1, x can be derived in G_0. (The other remaining case, in which x ends with 1, is very similar.) We have $x = 1y$, where $d(y) = n_0(y) - n_1(y) \geq 2$. Since each symbol appended to a string changes the d value by 1, there must be some prefix z of y for which $d(z) = 1$, and if $y = zw$, it follows that $d(w) \geq 1$. The induction hypothesis implies that z and w can both be derived from S; it follows that x can, because we can

start a derivation of x with the production $S \to 1SS$ and continue by deriving z from the first S and w from the second.

6.46. We can show that the CFG generates $\{a, b\}^*$. Clearly Λ can be generated. Let us show by induction that for every $n \geq 1$, any string of length n can be derived from T. Suppose $k \geq 1$ and every string of length k can be derived from T. Let $|x| = k + 1$. If $x = ay$, then by the induction hypothesis $T \Rightarrow^* y$. It follows that $S \Rightarrow^* y$, because $S \Rightarrow ST \Rightarrow T$. Therefore, $T \Rightarrow^* x$, because we can start the derivation $T \Rightarrow aS$ and continue by deriving y from S. If $x = by$, then it follows from the induction hypothesis that $T \Rightarrow^* y$, and therefore $T \Rightarrow^* x$ because we can start the derivation $T \Rightarrow bT$ and continue by deriving y from T.

6.48. The grammar generates the language of strings in which the number of 0's and the number of 1's are both even. The proof that every string generated has this property is straightforward. In the other direction, the induction step uses the observation that if x is a nonnull string for which $n_0(x)$ and $n_1(x)$ are even, then for some y having the same property, one of the statements $x = 00y$, $x = 11y$, $x = y00$, $x = y11$, $x = 01y01$, $x = 01y10$, $x = 10y10$, $x = 10y01$ must be true.

6.50. Let $d(x) = n_0(x) - n_1(x)$. We can show by induction that for every $n \geq 0$, if $|x| = n$, these three statements hold: (i) if $S \Rightarrow^* x$, then $d(x) = 0$; (ii) if $A \Rightarrow^* x$, then $d(x) = 1$; and (iii) if $B \Rightarrow^* x$, then $d(x) = -1$. Assume all three statements are true for $n \leq k$, and consider $n = k + 1$. If $S \Rightarrow^* x$, then either $x = 0y$, where $B \Rightarrow^* y$, or $x = 1y$, where $A \Rightarrow^* y$. In the first case, $d(y) = -1$ by the induction hypothesis, and therefore $d(x) = 0$; in the second case, $d(y) = 1$ by the induction hypothesis, and therefore $d(x) = 0$. The proofs in the cases $A \Rightarrow^* x$ and $B \Rightarrow^* x$ are similar.

In the opposite direction, we can show that for every $n \geq 0$, if $|x| = n$, the converses of the three statements above hold. Suppose this is true for $n \leq k$, and consider $n = k + 1$. If $d(x) = 0$, then either $x = 0y$ for some y with $d(y) = -1$ or $x = 1y$ for some y with $d(y) = 1$. In the first case, it follows from the induction hypothesis that $B \Rightarrow^* y$, and therefore $S \Rightarrow^* x$ because we can start the derivation $S \Rightarrow 0B$. In the second case, it follows from the induction hypothesis that $A \Rightarrow^* y$, and so $S \Rightarrow^* x$ because we can start the derivation $S \Rightarrow 1A$.

If $d(x) = 1$ and $x = 0y$, then $d(y) = 0$; by the induction hypothesis, $S \Rightarrow^* y$; and so $A \Rightarrow^* x$, because we can start the derivation $A \Rightarrow 0S$. If $d(x) = 1$ and $x = 1y$, then it follows from the discussion in Example 6.8 that $y = wz$ for some w and z with $d(w) = d(z) = 1$. By the induction hypothesis, both w and z can be derived from A. Therefore, x can be derived from A, since we can start the derivation $A \Rightarrow 1AA$.

The proof in the case where $d(x) = -1$ is similar.

6.51. We show by induction on n that every string in L of length n can be generated by the grammar. The basis step is easy, since $S \to \Lambda$ is a production in the grammar. Suppose that $k \geq 0$ and that every string of length k or less having equal numbers of 0's and 1's can be generated by the grammar. Suppose x is a string of length $k + 1$ having equal numbers

of 0's and 1's. If x starts with either 01 or 10, then the induction hypothesis implies that the suffix y of length $k-1$ can be generated by the grammar. It follows that x can, since in the first case we can start the derivation with the production $S \rightarrow 0S1S$ and continue by replacing the first S by Λ and deriving y from the second. The argument is similar in the second case. If x starts with neither of these strings, then it begins with either 00 or 11. We complete the proof in the first case.

Suppose that $x = 00y$. Let $d(z) = n_0(z) - n_1(z)$, and consider $d(z)$ for the prefixes z of x. $d(00) = 2$ and $d(x) = 0$. Therefore, there must exist a prefix z for which $d(z) = 1$. Let z_1 be the longest such prefix. Then x must be $z_1 1 z_2$ for some string z_2—otherwise, since $d(z_1 0) = 2$, there would be a longer prefix for which $d = 1$. We now know that $x = 00y_1 1z_2$, and $d(0y_1) = d(00y_1 1) = 0$. Therefore, $d(z_2) = 0$ as well. By the induction hypothesis, both $0y_1$ and z_2 can be derived from S. Therefore, $x = 0(0y_1)1z_2$ can be also, since we can start a derivation with the production $S \rightarrow 0S1S$.

6.52. (a) The language of balanced strings of parentheses.

(b) Let G be the CFG with productions $S \rightarrow SS \mid (S) \mid \Lambda$, and let G_1 be the CFG with productions $S_1 \rightarrow S_1(S_1) \mid \Lambda$.

First we show that $L(G_1) \subseteq L(G)$, by showing that for any $n \geq 1$, if x can be derived from S_1 in n steps, then $x \in L(G)$. The basis step, $n = 1$, is clear. We assume that $k \geq 0$ and that any string derivable from S_1 in k or fewer steps is derivable from S. Now suppose x can be derived from S_1 in $k+1$ steps. The first step must be $S_1 \Rightarrow S_1(S_1)$, and thus $x = y(z)$, where y and z can both be derived from S_1 in k or fewer steps. By the induction hypothesis, both y and z can be derived from S. Therefore, x can, since we can take the first two steps of the derivation to be $S \Rightarrow SS \Rightarrow S(S)$, and we can continue by deriving y from the first S and z from the second.

Next we show that $L(G) \subseteq L(G_1)$, by showing that for any $n \geq 1$, if x can be derived from S in n steps, then $x \in L(G_1)$. Again the basis step is easy. Assume that $k \geq 1$ and that any string derivable from S in k or fewer steps is derivable from S_1. Let x be a string derivable from S in $k+1$ steps. We may assume there is no derivation with k or fewer steps, since otherwise the induction hypothesis would give us the result. Fix a specific $(k+1)$-step derivation. If the first step is $S \Rightarrow (S)$, then $x = (y)$, where y is derivable from S in k steps. By the induction hypothesis, y is derivable from S_1. Therefore, we can derive x from S_1, by starting the derivation $S_1 \Rightarrow S_1(S_1) \Rightarrow \Lambda(S_1) = (S_1)$ and continuing to derive y from S_1. We are left with the case in which the first step in the derivation of x is $S \Rightarrow SS$, which implies that $x = yz$, where y and z are nonnull strings in $L(G)$. (If either were null, x would have a derivation with k or fewer steps.) Suppose y and z are strings like this for which y is as short as possible. If y had a derivation in G that started $S \rightarrow SS$, in which nonnull strings y_1 and y_2 were subsequently derived from both S's, then we could write $x = y_1 y_2 z = y_1 z'$, where y_1 and z' are nonnull strings in $L(G)$ with $|y_1| < |y|$. Therefore, y must have a derivation beginning with $S \rightarrow (S)$. This implies that $x = (w)z$, where w and z are in $L(G)$ and both w and z can be derived in G in k or fewer steps. The induction hypothesis implies that w and z are in $L(G_1)$. It follows that $x \in L(G_1)$, because we may start a derivation $S_1 \rightarrow (S_1)S_1$ and continue by deriving w from the first S_1 and z from the second.

6.53. Call this grammar G, and let L be the language of strings x with $n_a(x) = 2n_b(x)$. The proof that $L(G) \subseteq L$ is a straightforward induction proof. We show, also using induction, that $L \subseteq L(G)$. It is clear that $\Lambda \in L(G)$. Suppose that $k \geq 0$ and that any string in L of length k or less is in $L(G)$. Let $x \in L$ with $|x| = k + 1$.

For any string z, let $d(z) = n_a(z) - 2n_b(z)$. We consider six cases, one of which must hold: i) x begins with aab; ii) x begins with ab; iii) x begins with aaa and ends with b; iv) x begins with aaa and ends with a; v) x begins with ba; and vi) x begins with bb.

In case i), we write $x = aaby$. Since $d(y)$ must also be 0, the induction hypothesis implies that $y \in L(G)$. Therefore, $x \in L(G)$, because we can start the derivation $S \Rightarrow aSaSbS \Rightarrow^* aabS$ and continue by deriving y from S.

In case ii), $x = aby$. We observe that $d(ab) = -1$ and $d(x) = d(aby) = 0$; consider the first prefix of x for which $d \geq 0$. Adding the last symbol of this prefix causes d to increase, and thus the prefix ends with a. This means that $x = abwaz$, where $d(w) = d(z) = 0$. By the induction hypothesis, w and z are both in $L(G)$. Therefore, x is also, because we can start the derivation $S \Rightarrow aSbSaS \Rightarrow abSaS$ and continue to derive w from the first S and z from the second.

In case iii), $x = aaayb$. All we really need in this case is that $x = aazb$. By the induction hypothesis, $z \in L(G)$; therefore, so is x, since we can start the derivation $S \Rightarrow aSaSbS \Rightarrow^* aaSb$ and continue by deriving z from S.

In case iv), $x = aaaya$. This time, since $d(aaa) > 0$, we consider the smallest nonnull prefix of x for which $d \leq 0$. This prefix must end with b, so that $x = aaawbza$. The possible values of $d(aaaw)$ are 1 and 2. If it is 1, then $d(aaw) = 0$, so that $x = a(aaw)bza$. In this case, the induction hypothesis implies that $aaw \in L(G)$ and $z \in L(G)$, and therefore x is also, since we can start the derivation $S \Rightarrow aSbSaS \Rightarrow aSbSa$ and continue by deriving aaw from the first S and z from the second. If the value is 2, then $d(aw) = 0$, and $x = aa(aw)b(za)$, so that $d(za)$ must be 0. The induction hypothesis then implies that aw and za are in $L(G)$. Therefore, x is, because we can start the derivation $S \Rightarrow aSaSbS \Rightarrow aaSbS$ and continue by deriving aw from the first S and za from the second.

In case v), $x = bay$. Since $d(ba) = -1$, we consider the smallest nonnull prefix for which $d \geq 0$. As in case ii), this prefix ends in a, which means that $x = bawaz$, where $d(w) = d(z) = 0$. By the induction hypothesis, w and z are in $L(G)$. Therefore, so is x, since we can start the derivation $S \Rightarrow bSaSaS \Rightarrow baSaS$ and continue by deriving w from the first S and z from the second.

Finally, in case vi) $x = bby$. Since $d(bb) < -1$, d must increase to -1 at some point, and later increase to 0. In both cases, the symbol causing this to happen is a. We may therefore write $x = bwaza$, where $d(w) = d(z) = 0$. By the induction hypothesis, w and z are in $L(G)$, and so starting a derivation $S \Rightarrow bSaSaS \Rightarrow bSaSa$ allows us to conclude that x is also.

6.54. No. The string $aabbaa$ cannot be generated by this grammar.

6.55. These two statements can be proved by induction: i) For every $n \geq 0$, and every x with $|x| = n$, if $S \Rightarrow^* x$ then $n_a(x) = 2n_b(x)$ and if $T \Rightarrow^* x$ then $n_a(x) = 2n_b(x) + 1$.

ii) For every $n \geq 0$, and every x with $|x| = n$, if $n_a(x) = 2n_b(x)$ then $S \Rightarrow^* x$ and if $n_a(x) = 2n_b(x) + 1$ then $T \Rightarrow^* x$. We omit the proof.

6.56. We describe the proof in the case when $\Sigma_1 = \Sigma_2 = \{0, 1\}$, and it can easily be adapted to the general case. Suppose $f : \{0,1\}^* \to \{0,1\}^*$ is a homomorphism, and let $f(0) = s_0$ and $f(1) = s_1$. Then f is determined by the two values s_0 and s_1. (For example, $f(011) = f(0)f(1)f(1)$, since f is a homomorphism, and so $f(011) = s_0s_1s_1$.) Suppose $G = (V, \{0,1\}, S, P)$ is a CFG generating L. Consider the new CFG $G_f = (V, \{0,1\}, S, P_f)$, where the productions in P_f are obtained by taking the productions in P and replacing all occurrences of 0 and 1 on the right sides by s_0 and s_1, respectively. Then it is not hard to see that G_f generates precisely the strings $f(x)$, for $x \in L(G)$.

For example, the language *pal* is generated by $S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \Lambda$. If $f(0) = 01$ and $f(1) = 110$, the grammar G_f has the productions $S \to 01S01 \mid 110S110 \mid 01 \mid 110 \mid \Lambda$. The string 010 has the derivation $S \Rightarrow 0S0 \Rightarrow 010$, and $f(010) = 0111001$ has the derivation $S \Rightarrow 01S01 \Rightarrow 0111001$.

6.57. We prove by induction that for any $n \geq 0$, and any x with $S \Rightarrow^* x$ and $|x| = n$, x has only one leftmost derivation. This is clear for $n = 0$. Suppose $k \geq 0$ and the statement is true for $n \leq k$. We must show that if $S \Rightarrow^* x$ and $|x| = k + 1$, x has only one leftmost derivation. The first step of any derivation of x must be $S \to (S)S$. Therefore, $x = (y)z$, where $S \Rightarrow^* y$ and $S \Rightarrow^* z$ and $|y|, |z| \leq k$. According to the induction hypothesis, both y and z have only one leftmost derivation. In order to conclude that x does, however, we must first eliminate the possibility that x can be written in *two different ways* as $(y)z$, where $S \Rightarrow^* y$ and $S \Rightarrow^* z$. The reason this is impossible is that if $x = (y)z$, where both y and z are balanced strings of parentheses, then the right parenthesis shown is the unique right parenthesis that matches the left parenthesis shown. Therefore, the grammar is unambiguous.

6.58. (a) We use the fact that this is $\{a^ib^jc^k \mid i < j + k\} \cup \{a^ib^jc^k \mid i > j + k\}$. For the first part, we can modify the CFG in Exercise 6.9(a) by requiring that at some point either some c's are generated to which no a's correspond, or some b's, or both. The variable B will stand for one or more b's, and C for one or more c's.

$$S \to aSc \mid TC \mid UC \mid C \qquad T \to aTb \mid \Lambda$$
$$U \to aUb \mid B \qquad B \to bB \mid b \qquad C \to cC \mid c$$

For the second part, this modification of the CFG in Exercise 6.9(a) will work:

$$S \to aSc \mid T \qquad T \to aTb \mid A \qquad A \to aA \mid a$$

We can now use the standard construction to form a CFG for the union of these two languages.

(b) This is the union of the two languages $L_1 = \{a^ib^jc^k \mid j > i+k\}$ and $L_2 = \{a^ib^jc^k \mid j < i + k\}$. Let $E = \{a^ib^i \mid i \geq 0\}$ and $F = \{b^jc^j \mid j \geq 0\}$, and let A, B, and C denote the languages $\{a^i \mid i > 0\}$, $\{b^i \mid i > 0\}$, and $\{c^i \mid i > 0\}$, respectively. Then $L_1 = EBF$ (this

was described in Example 6.11 for the alphabet $\{0,1\}$), and $L_2 = AEF \cup EFC \cup AEFC$. The second equality means that a string in L_2 consists of a string of the form a^ib^i followed by a string b^jc^j, with one or more additional a's at the beginning, one or more additional c's at the end, or both. Constructing CFGs for the languages E, F, A, B, and C is straightforward.

6.59. (a) $S \to aSb \mid aaSbbb \mid \Lambda$. It is easy to see that every string obtained by this CFG satisfies the given inequalities. We show that if i and j are nonnegative integers with $i \le j \le 3i/2$, then a^ib^j can be obtained from the grammar. The proof is by induction on i. If $i = 0$, then the inequalities are satisfied only if $j = 0$, and Λ can be obtained from the grammar. Suppose $k \ge 0$ and that for any i and j satisfying $0 \le i \le k$ and $i \le j \le 3i/2$, a^ib^j can be obtained from the grammar. We wish to show that if $i = k+1$ and $i \le j \le 3i/2$, then a^ib^j can be obtained from the grammar.

In the case $j = k+1$, $a^ib^j = a^{k+1}b^{k+1}$ can be obtained by using the first production $k+1$ times. Otherwise, $k+2 \le j \le 3(k+1)/2$. Provided $k \ge 1$, therefore, it follows that $i_1 = i-2 = k-1$ and $j_1 = j-3$ are nonnegative integers satisfying $i_1 \le j_1 \le 3i_1/2$. By the induction hypothesis, $a^{i_1}b^{j_1}$ can be obtained from the grammar. Therefore, by using the production $S \to aaSbbb$ one extra time, so can the string $a^{i_1+2}b^{j_1+3} = a^{k+1}b^j$. If $k = 0$, however, the inequalities $k+2 \le j \le 3(k+1)/2$ are impossible, and so the proof is complete.

(b) $S \to aaSb \mid aaSbbb \mid aSb \mid \Lambda$. Again it is easy to see that every string a^ib^j obtained from this CFG satisfies the inequalities $i/2 \le j \le 3i/2$. We show that if i and j are nonnegative integers satisfying these inequalities, then a^ib^j can be obtained from the grammar. It is straightforward to check that this is correct if $i \le 2$. For the induction step, suppose that $k \ge 2$, and for any $i \le k$ and any j satisfying $i/2 \le j \le 3i/2$, a^ib^j can be obtained from the grammar. Now we wish to show that if $(k+1)/2 \le j \le 3(k+1)/2$, then $a^{k+1}b^j$ can be obtained. We may do this by considering several cases.

If k is odd and $j = (k+1)/2$, $a^{k+1}b^j = (aa)^jb^j$, and thus the string can be obtained by using the first production j times.

If k is even and $j = k/2+1$, $a^{k+1}b^j = (aa)^{k/2}ab^{k/2}b$, and so the string can be obtained by using the first production $k/2$ times and the third production once.

If k is odd and $j = (k+3)/2$, $a^{k+1}b^j = (aa)^{(k-1)/2}a^2b^{(k-1)/2}b^2$, and the string can be obtained by using the first production $(k-1)/2$ times and the third production twice.

If k is even and $j = k/2+2$, $a^{k+1}b^j = (aa)^{k/2-1}a^3b^{k/2-1}b^3$, and the string can be obtained by using the first production $k/2-1$ times and the third production three times.

The only remaining case is the one in which $(k+5)/2 \le j \le 3(k+1)/2$ (and k is either even or odd). In this case, it is easy to check that $(k-1)/2 \le j-3 \le 3(k-1)/2$. By the induction hypothesis, therefore, the string $a^{k-1}b^{j-3}$ can be obtained from the grammar. Therefore, the string $a^{k+1}b^j$ can be, by using the same derivation except that one more step is added in which the third production is used.

6.60. (a) The string $aacbc$ has the two derivations $S \Rightarrow aS \Rightarrow aaSbS \Rightarrow^* aacbc$ and $S \Rightarrow aSbS \Rightarrow aaSbS \Rightarrow^* aacbc$.

(b) We show first that if x can be derived in G_1, then x can be derived in G. First, it is clear that if $T \Rightarrow^*_{G_1} x$, then $S \Rightarrow^*_G x$, since both the productions $S \to aSbS \mid c$ are present

in G. Let us prove by induction on $|x|$ that if either $S_1 \Rightarrow^*_{G_1} x$ or $U \Rightarrow^*_{G_1} x$, then $S \Rightarrow^*_G x$. The basis step, when $|x| = 1$, is easy, since the only string of length 1 that can be derived from either S_1 or U is c, and $S \rightarrow c$ is a production in G. Suppose that $k \geq 1$ and that every string x with $|x| \leq k$ that can be derived in G_1 from either S_1 or U can be derived in G from S. Now consider a string x with $|x| = k + 1$.

If $S_1 \Rightarrow T \Rightarrow^*_{G_1} x$, then we have already observed that $S \Rightarrow^*_G x$. If $S_1 \Rightarrow U \Rightarrow aS_1 \Rightarrow^*_{G_1} x$, then $x = ay$, where $S_1 \Rightarrow^*_{G_1} y$ and $|y| = k$. By the induction hypothesis, $S \Rightarrow^*_G y$. Therefore, $S \Rightarrow^*_G x$, because we can begin the derivation with the production $S \rightarrow aS$. Finally, if $S_1 \Rightarrow U \Rightarrow aTbU \Rightarrow^*_{G_1} x$, then $x = aybz$, where $T \Rightarrow^*_{G_1} y$, $U \Rightarrow^*_{G_1} z$, and $|y|$ and $|z|$ are both no larger than k. We know that y can be derived from S in G, and the induction hypothesis tells us that z can. Therefore, since we have the production $S \rightarrow aSbS$ in G, the string x can be derived from S in G.

In the other direction, we first observe that $L(G)$ can be described in a way similar to the language in Exercise 6.41—a slightly more complicated way, simply because the production $S \rightarrow \Lambda$ has been replaced by $S \rightarrow c$. It is not hard to see that for $x \in L(G)$, x matches the regular expression $a^*c(ba^*c)^*$, and every prefix of x has at least as many a's as b's. We can prove in a way similar to the solution to Exercise 6.41 that every x having this property is in $L(G)$. In the induction step, if x is a string of length $k + 1$ with this property, then x must start with a. The case $x = ay$, where y also has this property, is straightforward. Otherwise $x = ay$, where y also matches the regular expression but some prefix of y has more b's than a's. If y_1 is the smallest such prefix, then y_1 ends in b, and so y matches the regular expression $a^*c(ba^*c)^*b$. This means, as in the solution to 6.41, that the prefix of y up to the last b is a string matching $a^*c(ba^*c)^*$, of which every prefix has at least as many a's as b's, and that $x = aybz$, where z also has this property. Therefore, by the induction hypothesis, y and z can both be generated in G, and thus x can, since G contains the production $S \rightarrow aSbS$.

Now we wish to prove that $L(G) \subseteq L(G_1)$. This is implied by the following three statements, which we prove simultaneously by induction: (i) If $x \in L(G)$ and x has a derivation in G involving only the productions $S \rightarrow aSbS$ and $S \rightarrow c$ (this is the same as saying that $x \in L(G)$ and x has equal numbers of a's and b's), then $S_1 \Rightarrow T \Rightarrow^*_{G_1} x$; (ii) if $x \in L(G)$ and x has a derivation in G that begins with the production $S \rightarrow aS$, then $S_1 \Rightarrow U \Rightarrow^*_{G_1} x$; and (iii) if $x \in L(G)$, x has more a's than b's, and every derivation of x in G begins with the production $S \rightarrow aSbS$, then $S_1 \Rightarrow U \Rightarrow^*_{G_1} aTbU \Rightarrow^*_{G_1} x$.

The basis step is straightforward. Suppose that all three statements hold for strings of length $\leq k$, and suppose $x \in L(G)$ and $|x| = k + 1$. If x has a derivation involving only $S \rightarrow aSbS$ and $S \rightarrow c$, then either $x = c$, or $x = aybz$ where both y and z are in $L(G)$ and have derivations involving only these two productions. Since G_1 contains the productions $T \rightarrow aTbT \mid c$, it follows from the induction hypothesis that $S_1 \Rightarrow T \Rightarrow^*_{G_1} x$. If x has a derivation beginning with $S \rightarrow aS$, then $x = ay$ for some $y \in L(G)$. The induction hypothesis implies that $y \in L(G_1)$, and thus $x \in L(G_1)$ because G_1 contains the productions $S_1 \Rightarrow U \Rightarrow aS_1$. Finally, suppose that $x \in L(G)$, x has more a's than b's, and every derivation in G begins with the production $S \rightarrow aSbS$. Here we use the characterization of $L(G)$ described above. The string x matches the regular expression $a^*c(ba^*c)^*$, every prefix has at least as many a's as b's, and there is at least one b. Let

$x = x_1y_1$, where x_1 is of the form a^*cba^*c. If x started with aa, then x_1 would be ax_2 for some $x_2 \in L(G)$, so that x would have a derivation starting $S \rightarrow aS$. Therefore, $x = acbz$ for some z. The string z also matches the regular expression $a^*c(ba^*c)^*$, every prefix of z has at least as many a's as b's, and z has more a's than b's. The induction hypothesis implies that $U \Rightarrow^*_{G_1} z$, and therefore, $S_1 \Rightarrow U \Rightarrow aTbU \Rightarrow^*_{G_1} x$.

(c) We show that for any $n \geq 1$, and any x, if x can be derived from one of the three variables in n steps, then x has only one leftmost derivation from that variable. The basis step, $n = 1$, is clear. Suppose that $k \geq 1$ and that any string derivable from one of the variables in k or fewer steps has only one leftmost derivation from that variable. We wish to show the same result for a string x derivable from one of the variables in $k + 1$ steps.

It is easy to see that strings derivable from T have equal numbers of a's and b's, and those derived from U have an excess of a's. Therefore, if $S_1 \Rightarrow^* x$, the first step in a derivation of x from S_1 must be $S_1 \Rightarrow T$ if $n_a(x) = n_b(x)$, and $S_1 \Rightarrow U$ otherwise. Therefore, the induction hypothesis implies the result for strings derivable in $k + 1$ steps from S_1.

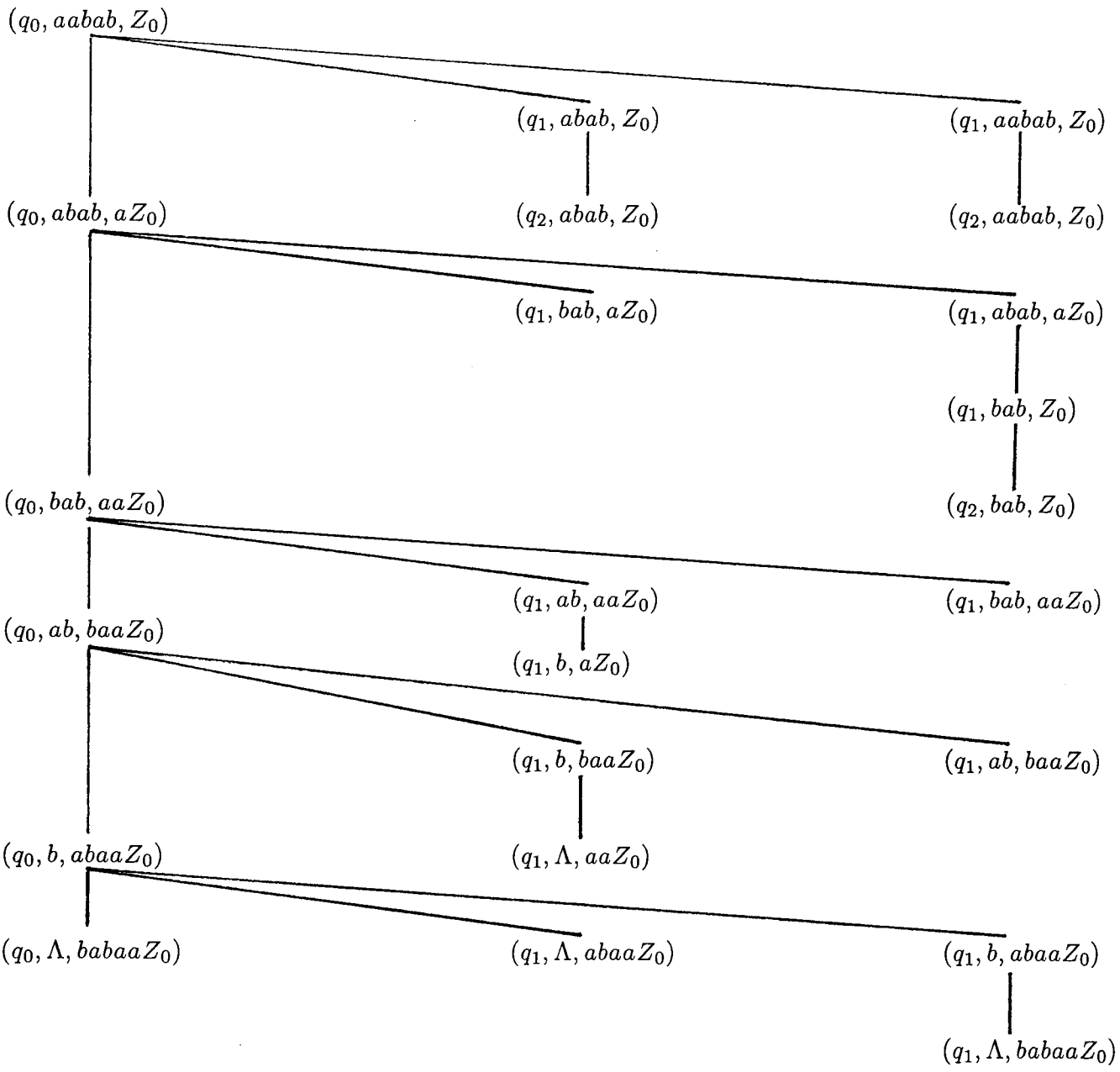
Suppose x is derivable from T in $k + 1$ steps. Since $k \geq 1$, the first step in a derivation from T must be $T \Rightarrow aTbT$. Thus $x = aybz$ for some strings y and z derivable from T in k or fewer steps. It is also easy to verify that in any string derivable from S, or in particular from T, every prefix has at least as many a's as b's. For this reason, the prefix ayb in the formula $x = aybz$ must be the smallest prefix ending with b having equal numbers of a's and b's. (Otherwise, $y = y_1by_2$, where ay_1b has equal numbers of a's and b's; but then the prefix y_1b of y has more b's than a's.) The induction hypothesis implies that both y and z have only one leftmost derivation from T. Since there is no choice for the strings y and z, it follows that x also has only one leftmost derivation from T.

Finally, suppose x is derivable from U in $k + 1$ steps. If some prefix of x is of the form ayb, where y has equal numbers of a's and b's, then the first step in a derivation of x from U must be $U \Rightarrow aTbU$. (If a derivation started $U \Rightarrow aS_1$, some string derivable from S_1 would have a prefix with more b's than a's.) Furthermore, if no prefix of x has this form, the first step in a derivation of x from U must be $U \Rightarrow aS_1$. If the first step is $U \Rightarrow aS_1$, the induction hypothesis tells us that there is only one way to continue a leftmost derivation, and therefore in this case x has only one leftmost derivation from U. If the first step is $U \Rightarrow aTbU$, then $x = aybz$, where y and z can be derived from T and U, respectively, in k or fewer steps. By the induction hypothesis, y has only one leftmost derivation from T and z has only one from U. Furthermore, just as in the preceding paragraph, there is only one choice for the string y. Therefore, x has only one leftmost derivation from U.

Chapter 7

Pushdown Automata

7.2. (for the string $aabab$)



7.3. $2n + 1$. One possible sequence is to push all the input symbols onto the stack. In addition, for each of the n symbols, there are the two sequences of moves in which the PDA guesses that the symbol is the middle symbol in an odd-length palindrome, and the first symbol in the second half of an even-length palindrome, respectively.

7.4.

(a)

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	(q_0, aZ_0)
2	q_0	b	Z_0	(q_0, bZ_0)
3	q_0	a	a	(q_0, aa)
4	q_0	b	a	(q_0, ba)
5	q_0	a	b	(q_0, ab)
6	q_0	b	b	(q_0, bb)
7	q_0	Λ	Z_0	(q_1, Z_0)
8	q_0	Λ	a	(q_1, a)
9	q_0	Λ	b	(q_1, b)
10	q_1	a	a	(q_1, Λ)
11	q_1	b	b	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)				none

(b)

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	$(q_0, aZ_0), (q_1, Z_0)$
2	q_0	b	Z_0	$(q_0, bZ_0), (q_1, Z_0)$
3	q_0	a	a	$(q_0, aa), (q_1, a)$
4	q_0	b	a	$(q_0, ba), (q_1, a)$
5	q_0	a	b	$(q_0, ab), (q_1, b)$
6	q_0	b	b	$(q_0, bb), (q_1, b)$
7	q_1	a	a	(q_1, Λ)
8	q_1	b	b	(q_1, Λ)
9	q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)				none

7.5. (a) In the following PDA q_3 is the only accepting state.

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	$(q_0, xZ_0), (q_1, aZ_0)$
2	q_0	b	Z_0	$(q_0, xZ_0), (q_1, bZ_0)$
3	q_0	a	x	$(q_0, xx), (q_1, ax)$
4	q_0	b	x	$(q_0, xx), (q_1, bx)$
5	q_1	a	a	(q_1, a)
6	q_1	b	b	(q_1, b)
7	q_1	b	a	$(q_1, a), (q_2, \Lambda)$
8	q_1	a	b	$(q_1, b), (q_2, \Lambda)$
9	q_2	a	x	(q_2, Λ)
10	q_2	b	x	(q_2, Λ)
11	q_2	Λ	Z_0	(q_3, Z_0)
(all other combinations)				none

This can be understood as follows. In state q_0 the PDA initially reads input symbols and pushes an x onto the stack for each one read. At some point, it guesses that the next symbol read will be one that fails to match the corresponding symbol in the second half. It pushes that symbol onto the stack and enters the state q_1. In state q_1, the PDA reads input, leaving the stack alone, until it guesses that it has reached the input symbol that corresponds to (i.e., fails to match) the top stack symbol. At this point it enters q_2, having read that input symbol and popped the nonmatching stack symbol. From that point on, the PDA is concerned only with the length of the remaining input. It pops an x from the stack for each input symbol read until the stack is empty except for Z_0, and at this point it enters the accepting state.

(b) In the PDA below, both states are accepting. (A string not in the language will cause the machine to crash before all the input is read.)

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	(q_0, aZ_0)
2	q_0	a	a	(q_0, aa)
3	q_0	b	a	(q_1, Λ)
4	q_1	a	a	(q_1, Λ)
5	q_1	b	a	(q_1, Λ)
(all other combinations)				none

(c)

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	$(q_1, Z_0), (q_4, Z_0)$
2	q_1	Λ	Z_0	(q_3, Z_0)
3	q_1	a	Z_0	(q_1, aZ_0)
4	q_1	a	a	(q_1, aa)
5	q_1	b	a	(q_2, Λ)
6	q_2	b	a	(q_2, Λ)
7	q_2	Λ	Z_0	(q_3, Z_0)
8	q_3	c	Z_0	(q_3, Z_0)
9	q_3	Λ	Z_0	(q_f, Z_0)
10	q_4	Λ	Z_0	(q_f, Z_0)
11	q_4	a	Z_0	(q_4, Z_0)
12	q_4	b	Z_0	(q_5, bZ_0)
13	q_5	b	b	(q_5, bb)
14	q_5	c	b	(q_6, Λ)
15	q_6	c	b	(q_6, Λ)
16	q_6	Λ	Z_0	(q_f, Z_0)
(all other combinations)				none

In this PDA, q_f is the accepting state. The machine makes an initial choice (move 1) as to whether it will compare the number of b's to the number of a's or to the number of

c's; q_1 and q_4 are the states that begin these respective processes. In q_1, one possibility is that there are no a's or b's at all; this is the reason for the Λ-transition in move 2. Otherwise, the PDA saves a's on the stack (moves 3 and 4), and when it sees a b pops a's off the stack to match inputs of b (moves 5 and 6). In either case (either from q_1 or from q_2), when the stack is empty except for Z_0 the PDA goes to q_3 (moves 2 and 7), from which the machine can move to the accepting state any time (move 9), and from which the only other move is to read a c, leaving the stack alone (move 8).

Starting in q_4 the moves are similar. The PDA moves to q_5 at any time, and the only move it can make before that is to read an a, leaving the stack alone. From q_5, the PDA pushes b's onto the stack, and when it sees a c it starts to remove b's from the stack to match the c's in the input. It enters the accepting state on a Λ-transition whenever the stack is empty except for Z_0.

(d) This language is the union of $L_1 = \{x \mid n_a(x) < n_b(x)\}$ and $L_2 = \{x \mid n_a(x) < n_c(x)\}$. Each can be accepted by a PDA similar to those in Example 7.4. For a general method of constructing a PDA to accept the union of two CFLs, see Exercise 7.14. (We could construct one directly by using the same approach as in part (c).)

7.6. (a) The language of strings over $\{a, b\}$ of length 2 or greater with first and last symbols the same.

(b) $\{xcy \mid x, y \in \{a, b\}^* \text{ and } |x| = |y|\}$.

7.7.

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	c	Z_0	(q_2, Z_0)
2	q_0	a	Z_0	$(q_0, xA'Z_0)$
3	q_0	b	Z_0	$(q_0, xB'Z_0)$
4	q_0	a	x	(q_0, xA)
5	q_0	b	x	(q_0, xB)
6	q_0	c	x	(q_0, Λ)
7	q_0	a	A	(q_0, Λ)
8	q_0	b	B	(q_0, Λ)
9	q_0	a	A'	(q_2, Λ)
10	q_0	b	B'	(q_2, Λ)
(all other combinations)				none

Here the symbol x on top of the stack tells the PDA that the input symbol is to be pushed onto the stack (and x placed on top). If A or B is on top of the stack, with no x, the PDA should try to match the input symbol with the stack symbol. (Once the symbol c is encountered, x will be removed from the stack and never placed there again.) The symbols A' and B' are used instead of A and B in the first step so that when they are matched with an input symbol the PDA can go to the accepting state.

7.8. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA accepting L. If we let p_0 and p_1 be the states

of the PDA, with p_0 being the initial state, then exactly one will be accepting: p_0 if $q_0 \in A$, and p_1 otherwise. The stack symbols of the PDA other than Z_0 will correspond to elements of Q. The PDA simulates the moves of M as follows. State p_0 and top stack symbol Z_0 is interpreted to mean that M is in state q_0. Thereafter, a move by M to state q is simulated by placing q on the stack (unless $q = q_0$, in which case Z_0 is placed on the stack), and going to the accepting state of the PDA if $q \in A$ and to the nonaccepting state otherwise. Thus, top stack symbol q is interpreted to mean that M is currently in state q.

7.9. We may construct an NFA-Λ M accepting L, by letting states of M be pairs (p, X), where p is a state of the PDA and X is a stack symbol. If p_0 is the initial state of the PDA, (p_0, Z_0) is the initial state of M, and a pair (p, X) is accepting if and only if p is an accepting state of the PDA. The PDA move from state p, using input a (either an alphabet symbol or Λ), with X on top of the stack, which moves to state q and places a string α on the stack, corresponds to the move $(p, X) \xrightarrow{a} (q, \sigma)$, where σ is the first symbol of α if $\alpha \neq \Lambda$ and X otherwise.

7.10. An NFA-Λ can be constructed to accept L, having as states pairs (q, α), where q is a state in M and α is a string of k or fewer stack symbols, representing the current contents of M's stack. Such a pair provides a complete description of M's current status, and for any such pair and any possible input (either Λ or an input symbol) it is possible using M's definition to specify the pairs that might result. Furthermore, there are only finitely many such pairs. The initial state of the NFA-Λ is (q_0, Z_0), where q_0 is the initial state of M, and the pairs (q, α) that are designated as accepting states are those for which q is an accepting state of M.

7.11. The original PDA M can be modified in two ways. The first is to have a preliminary sequence of moves that causes a new stack symbol U to be inserted underneath Z_0; the second is to add one more new state, i, having the property that once i is entered, the PDA continues to loop back to i on any input. The new machine enters the state i in either of two situations: when the top stack symbol is U (which in the original machine M would mean that the stack was empty), or as a result of any combination of state, input symbol, and stack symbol for which no move was defined in M. Since these modifications do not affect sequences of moves in M that end at an accepting state, and since they do not create any new sequences of moves leading to an accepting state, the new PDA still accepts the same language.

7.12. Any more general move, say $(q, \alpha) \in \delta(p, a, X)$, can be replaced by a sequence of moves of the restricted type that have the same ultimate effect. The new states introduced in order to accomplish this are specific to this move, and thus there is no change in the language accepted by the machine.

7.13. (a, b) In part (a), the state q_0 is the only accepting state. In part (b), the two states q_a and q_b are the accepting states. The approach taken by the PDA is similar to the one in Table 7.5.

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	(q_a, Z_0)
2	q_0	b	Z_0	(q_b, Z_0)
3	q_a	a	Z_0	(q_a, aZ_0)
4	q_a	a	a	(q_a, aa)
5	q_a	b	a	(q_a, Λ)
6	q_a	b	Z_0	(q_0, Z_0)
7	q_b	b	Z_0	(q_b, bZ_0)
8	q_b	b	b	(q_b, bb)
9	q_b	a	b	(q_b, Λ)
10	q_b	a	Z_0	(q_0, Z_0)
(all other combinations)				none

(c) In the PDA shown, the initial state is q_0 and the only accepting state is q_1. Let us denote by x the current string. Similar to what we did in Table 7.5 (in Example 7.4), if the PDA is in state q_1, Z_0 on top of the stack means $n_a(x) = 2n_b(x) - 1$ and b on top of the stack means $n_a(x) \leq 2n_b(x) - 2$. Since we wish to compare $n_a(x)$ to $2n_b(x)$, we will essentially treat every b we read as if it is two b's. This means that if the stack has no a's on it (which means we are in state q_1 already), we push two b's onto the stack. If there is an a on top of the stack, then we pop it off; at that point, we pop a second a off if there is one, and otherwise we go to state q_1.

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	(q_0, aZ_0)
2	q_0	b	Z_0	(q_1, bZ_0)
3	q_0	a	a	(q_0, aa)
4	q_0	b	a	(q_t, Λ)
5	q_t	Λ	a	(q_0, Λ)
6	q_t	Λ	Z_0	(q_1, Z_0)
7	q_1	a	Z_0	(q_0, Z_0)
8	q_1	b	Z_0	(q_1, bbZ_0)
9	q_1	b	b	(q_1, bbb)
10	q_1	a	b	(q_1, Λ)
(all other combinations)				none

(d) The accepting states in this PDA are q_0, q_3, and q_6. The state q_1 is for pushing a's onto the stack, q_2 is for matching a's on the stack with b's in the input, q_4 is for pushing subsequent b's onto the stack, and q_5 is for matching them with a's in the input.

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	(q_1, aZ_0)
2	q_0	b	Z_0	(q_4, bZ_0)
3	q_1	a	a	(q_1, aa)
4	q_1	b	a	(q_2, Λ)
5	q_2	b	a	(q_2, Λ)
6	q_2	Λ	Z_0	(q_3, Z_0)
7	q_3	b	Z_0	(q_4, bZ_0)
8	q_4	b	b	(q_4, bb)
9	q_4	a	b	(q_5, Λ)
10	q_5	a	b	(q_5, Λ)
11	q_5	Λ	Z_0	(q_6, Z_0)
(all other combinations)				none

7.14. (a) In all three parts of this problem, we start by making sure (relabeling if necessary) that the states of the two machines don't overlap. We can take care of unions the same way we did for FAs. The composite machine can choose one of two Λ-transitions from the initial state: one to the initial state of M_1, the other to the initial state of M_2. Thereafter, it executes the moves of the appropriate M_i.

(b) The basic idea is the same as for FAs, which is that the composite machine acts like M_1 until it reaches an accepting state, then takes a Λ-transition to the initial state of M_2. However, we have to be more careful because of the stack. For example, it's possible for M_1 to accept a string but to have an empty stack when it's done processing that string; however, if our composite machine ever has an empty stack, it can't move. The solution is to fix it so that the composite machine can never empty its stack. This can be done as in Exercise 7.11, by inserting a new stack symbol Z under the initial stack symbol of M_1 and then returning to the initial state of M_1.

From that point on, the moves are the same as those of M_1, unless Z appears on top of the stack or unless the state is an accepting state of M_1. If Z is on top of the stack and the state is not an accepting state in M_1, the machine crashes. If it reaches an accepting state of M_1, then (regardless of what's on top of the stack) it takes a Λ-transition to the initial state of M_2 and pushes the initial stack symbol of M_2 onto the stack. From then on, it acts like M_2, except that it never removes the initial stack symbol of M_2 from the stack. (If M_2 ever removed its initial stack symbol, that would be the last move it made, so our new machine can achieve the same result as M_2 without ever removing this symbol.)

(c) Same general idea as (b). The new PDA has the same states as the original, except that there's one new state q, which is the only accepting state. Every time the machine is in q, it makes a Λ-transition to the initial state of the original machine, in which it also pushes two symbols onto the stack, first a new symbol Z and then the initial stack symbol of the original machine on top of Z. From that point, the machine makes the same moves as the original machine, except that if it ever sees Z on the stack and it is not in one of the accepting states of the original machine, it crashes. Whenever it's in one of the accepting states of the original machine, it has the same moves as originally, plus one more, which is a Λ-transition to q.

7.15. If M is a DPDA accepting L by empty stack, x and y are distinct strings in L, and x is a prefix of y, then the sequence of moves M must make in order to accept x leaves the stack empty, since $x \in L$. But then y cannot be accepted, since M can't move with an empty stack.

7.16. Let $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ be a DPDA accepting L. We wish to construct a DPDA M_1 that accepts $L\{\$\}$ by empty stack. M_1 has all the states in Q and two new ones, q_e and q_i; it has all the stack symbols in Γ as well as one new one U. It doesn't matter what its accepting states are, since the mode of acceptance doesn't involve states. Denote its transition function by δ_1.

q_i is the initial state of M_1, and from this state M_1 places the stack symbol U underneath Z_0 before entering q_0. For states in Q, inputs in $\Sigma \cup \{\Lambda\}$, and stack symbols in Γ, the moves of M_1 are identical to those of M. For every $q \in A$ and every symbol X in $\Gamma \cup \{U\}$, $\delta_1(q, \$, X) = \{(q_e, X)\}$. Finally, for every $X \in \Gamma \cup \{U\}$, $\delta_1(q_e, \Lambda, X) = \{(q_e, \Lambda)\}$.

It is clear that M_1 is deterministic since M is. For any $x \in L$, there is a sequence of moves of M that lead to an accepting state q; from q there is a Λ-transition to q_e on input \$; and from q_e there is a sequence of Λ-transitions that causes the stack to empty. Therefore, $x\$$ is accepted by M_1, by empty stack. Conversely, it is clear that the only way M_1 can empty its stack is by entering q_e first, and the only strings that cause it to do this are strings $x\$$, where x is accepted by M. Therefore, M_1 accepts the language $L\{\$\}$.

7.17. The property of *pal* that is used in the proof is the one established in Theorem 3.3: any two distinct strings in Σ^* are distinguishable with respect to *pal*. The argument in Theorem 3.3 can be adapted easily to show that the languages of even-length and odd-length palindromes both have this property. We show that the language in (d) also does.

Let $x, y \in \{0,1\}^*$ with $x \neq y$. If $|x| = |y|$, let $z = 0x0$. Then $xz = x0x0$ is obviously in the language. The string $yz = y0x0$, however, cannot be in the language: It cannot be of the form $ww^\sim$ because of the 0's, and it can't be of the form ww, since $x \neq y$. If the lengths are not equal, say $|x| < |y|$, then if $|y| - |x|$ is odd, the string x distinguishes x and y relative to the language, since $|yx|$ is odd. If $|y| - |x| = 2m$, then let $y = y_1\alpha\beta$, where $|y_1| = |x|$ and $|\alpha| = |\beta| = m$, and let $z = \beta\beta^\sim x\beta\beta^\sim$. Then $xz = (x\beta\beta^\sim)(x\beta\beta^\sim)$ is in the language. However, $yz = (y_1\alpha\beta\beta)(\beta^\sim x\beta\beta^\sim)$. This string cannot be of the form ww, because the first half ends with β and the second with $\beta^\sim$; and it can't be $ww^\sim$, because the preceding portions of the two halves agree. Therefore, z distinguishes x and y.

7.18. (a) The PDA works as follows. Instead of saving excess 0's or excess 1's on the stack, we save $*$'s, and use two different states to indicate which symbol there is currently a surplus of. The state q_0 is the initial state and the only accepting state.

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	0	Z_0	$(p_0, *Z_0)$
2	q_0	1	Z_0	$(p_1, *Z_0)$
3	p_0	0	$*$	$(p_0, **)$
4	p_1	1	$*$	$(p_1, **)$
5	p_0	1	$*$	(p_0, Λ)
6	p_1	0	$*$	(p_1, Λ)
7	p_0	Λ	Z_0	(q_0, Z_0)
8	p_1	Λ	Z_0	(q_0, Z_0)
(all other combinations)				none

(b) For $x \in \{0,1\}^*$, let $d(x) = 2n_1(x) - n_0(x)$. We use the three states q_0, q_+, and q_- to indicate that the quantity $d(x)$ is currently 0, positive, or negative, respectively, and the number of $*$'s on the stack indicates the current absolute value of $d(x)$. q_+ is the accepting state.

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	0	Z_0	$(q_-, *Z_0)$
2	q_0	1	Z_0	$(q_+, * * Z_0)$
3	q_-	0	$*$	$(q_-, **)$
4	q_-	1	$*$	(q_t, Λ)
5	q_t	Λ	Z_0	$(q_+, *Z_0)$
6	q_t	Λ	$*$	(q_-, Λ)
7	q_-	Λ	Z_0	(q_0, Z_0)
8	q_+	1	$*$	$(q_+, * * *)$
9	q_+	0	$*$	(q_0, Λ)
10	q_0	Λ	$*$	$(q_+, *)$
(all other combinations)				none

Lines 4-7 of the table are the moves required in the case where $d(x) < 0$ and we read the symbol 1. If there are at least two $*$'s on the stack, we pop two off, and if this empties the stack we go to q_0. If there is only one $*$ on the stack, then we go to q_-, ending up with one $*$ on the stack. The state q_t is a temporary state used in this procedure.

Note that it's possible for the PDA to be temporarily in state q_- when the stack is empty, but if this happens, the machine makes a Λ-transition to q_0. In move 9, we are forced to go to some state other than q_+, in case popping the $*$ would cause us accidentally to accept a string x for which $d(x) = 0$. This means that the PDA can also be temporarily in q_0 when there is at least one $*$ on the stack; but again, if this happens, we immediately make a Λ-transition back to q_+.

7.19. The PDA in Example 7.3 (Table 7.3) is such an example. The string }, for example, causes the machine to crash immediately.

7.20. Let us use {} and () as our two types of parentheses. One definition is to say that a string is balanced if three conditions hold: i) When {'s and }'s are ignored, the remainder

is a balanced string, as in Definition 6.5; ii) when ('s and)'s are ignored, the remainder is a balanced string; and iii) For any substring starting with a left parenthesis of either type and ending with its mate, no prefix has more right parentheses of the other type than left. Condition iii) is equivalent to saying that for each left parenthesis a of either type, the string of parentheses of the other type appearing between a and its mate is balanced.

7.21. (b)

$$
\begin{array}{lll}
 & (q_0, (a*a+a), Z_0) & \\
\vdash & (q_1, (a*a+a), SZ_0) & \qquad S \\
\vdash & (q_1, (a*a+a), (S)Z_0) & \Rightarrow \ (S) \\
\vdash & (q_1, a*a+a), S)Z_0) & \\
\vdash & (q_1, a*a+a), S+S)Z_0) & \Rightarrow \ (S+S) \\
\vdash & (q_1, a*a+a), S*S+S)Z_0) & \Rightarrow \ (S*S+S) \\
\vdash & (q_1, a*a+a), a*S+S)Z_0) & \Rightarrow \ (a*S+S) \\
\vdash & (q_1, *a+a), *S+S)Z_0) & \\
\vdash & (q_1, a+a), S+S)Z_0) & \\
\vdash & (q_1, a+a), a+S)Z_0) & \Rightarrow \ (a*a+S) \\
\vdash & (q_1, +a), +S)Z_0) & \\
\vdash & (q_1, a), S)Z_0) & \\
\vdash & (q_1, a), a)Z_0) & \Rightarrow \ (a*a+a) \\
\vdash & (q_1,),)Z_0) & \\
\vdash & (q_1, \Lambda, Z_0) & \\
\vdash & (q_2, \Lambda, Z_0) &
\end{array}
$$

7.22. A sequence of moves causing aba to be accepted is

$$
\begin{array}{lll}
(q_0, ababa, Z_0) & \vdash & (q_0, baba, aZ_0) \\
 & \vdash & (q_0, aba, baZ_0) \\
 & \vdash & (q_1, ba, baZ_0) \\
 & \vdash & (q_1, a, aZ_0) \\
 & \vdash & (q_1, \Lambda, Z_0) \\
 & \vdash & (q_2, \Lambda, \Lambda)
\end{array}
$$

The corresponding derivation in the CFG is as follows.

$$
\begin{array}{lll}
S \Rightarrow [q_0, Z_0, q_2] & \Rightarrow & a[q_0, A, q_1][q_1, Z_0, q_2] \\
 & \Rightarrow & ab[q_0, B, q_1][q_1, A, q_1][q_1, Z_0, q_2] \\
 & \Rightarrow & aba[q_1, B, q_1][q_1, A, q_1][q_1, Z_0, q_2] \\
 & \Rightarrow & abab[q_1, A, q_1][q_1, Z_0, q_2] \\
 & \Rightarrow & ababa[q_1, Z_0, q_2] \\
 & \Rightarrow & ababa
\end{array}
$$

(As in Example 7.7, we use upper-case letters as stack symbols.)

7.23. The PDA is deterministic only if, for each variable A, there is at most one production with left side A. In this case, there is no choice whatsoever in a leftmost derivation; in other words, there is only one leftmost derivation possible, and only one string in the language.

7.24. (a)

Type of Move	State	Stack	Unread Input	Production used
(initial)	q	Z_0	[]$[]$	
reduce	q	SZ_0	[]$[]$	$\Rightarrow$ []$[]$
shift	q	$[SZ_0$	]$[]$	
reduce	q	$S[SZ_0$	]$[]$	$\Rightarrow S[][]$
shift	q	$]S[SZ_0$	$[]$	
reduce	$q_{1,1}$	$S[SZ_0$	$[]$	$\Rightarrow S[S][]$
	$q_{1,2}$	$[SZ_0$	$[]$	
	$q_{1,3}$	SZ_0	$[]$	
	q	SZ_0	$[]$	
shift	q	$[SZ_0$	$]$	
reduce	q	$S[SZ_0$	$]$	$\Rightarrow S[]$
shift	q	$[S[SZ_0$	$]]$	
reduce	q	$S[S[SZ_0$	$]]$	$\Rightarrow S[S[]]$
shift	q	$]S[S[SZ_0$	$]$	
reduce	$q_{1,1}$	$S[S[SZ_0$	$]$	$\Rightarrow S[S[S]]$
	$q_{1,2}$	$[S[SZ_0$	$]$	
	$q_{1,3}$	$S[SZ_0$	$]$	
	q	$S[SZ_0$	$]$	
shift	q	$]S[SZ_0$	Λ	
reduce	$q_{1,1}$	$S[SZ_0$	Λ	$S \Rightarrow S[S]$
	$q_{1,2}$	$[SZ_0$	Λ	
	$q_{1,3}$	SZ_0	Λ	
	q	SZ_0	Λ	
(pop S)	q_1	Z_0	Λ	
(accept)	q_2	Z_0	Λ	

7.25. If the PDA is deterministic, the resulting grammar is unambiguous. This could happen even with a nondeterministic PDA, however; it could happen that the PDA has a choice of moves at some point but that only one choice leads to acceptance.

7.27. (a) If we follow Example 7.8 and make the PDA operate by replacing any variable V on the stack by the right side of a production $V \rightarrow \alpha$, we obtain this PDA.

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	$(q_1, S_1\$)$
3	q_1	a	S_1	(q_a, AS_1)
4	q_1	b	S_1	(q_b, AS_1)
5	q_1	\$	S_1	$(q_\$, \Lambda)$
6	q_1	a	A	(q_a, aA)
7	q_1	b	A	(q_b, b)
8	q_1	a	a	(q_1, Λ)
9	q_1	b	b	(q_1, Λ)
10	q_1	\$	\$	(q_1, Λ)
11	q_a	Λ	A	(q_a, aA)
12	q_a	Λ	a	(q_1, Λ)
13	q_b	Λ	A	(q_b, b)
14	q_b	Λ	b	(q_1, Λ)
15	$q_\$$	Λ	\$	(q_1, Λ)
16	q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)				none

For example, in state q_1, if S_1 is on the stack and the next input is a, we first replace S_1 by AS_1, then (without reading any more input) replace A by aA, and finally pop the a, which matches the input symbol already read. The intermediate state q_a is used to accomplish these last two steps. It is obvious that the PDA could be made simpler and more efficient by compressing these steps into one move: S_1 is replaced on the stack by AS_1.

7.28. (c)

$$S \to S_1\$ \qquad S_1 \to abX \qquad X \to TX \mid \Lambda \qquad T \to aU \qquad U \to Tbb \mid b$$

7.29. The resulting grammar would be the same except that the last production $U \to b$ would be replaced by $U \to \Lambda$. Consider the string $abaabbaa\$$. The moves by which the nondeterministic PDA accepts this string are given below.

$$\begin{array}{lll}
(q_0, abaabbaa\$, S) & \vdash (q_1, abaabbaa\$, S_1\$Z_0) & \vdash (q_1, abaabbaa\$, abX\$Z_0) \\
\vdash (q_1, baabbaa\$, bX\$Z_0) & \vdash (q_1, aabbaa\$, X\$Z_0) & \vdash (q_1, aabbaa\$, TX\$Z_0) \\
\vdash (q_1, aabbaa\$, aUX\$Z_0) & \vdash \underline{(q_1, abbaa\$, UX\$Z_0)} & \vdash (q_1, abbaa\$, TbbX\$Z_0) \\
\vdash (q_1, abbaa\$, aUbbX\$Z_0) & \vdash \overline{(q_1, bbaa\$, UbbX\$Z_0)} & \vdash (q_1, bbaa\$, bbX\$Z_0) \\
\vdash (q_1, baa\$, bX\$Z_0) & \vdash (q_1, aa\$, X\$Z_0) & \vdash (q_1, aa\$, TX\$Z_0) \\
\vdash \underline{(q_1, aa\$, aUX\$Z_0)} & \vdash \underline{(q_1, a\$, UX\$Z_0)} & \vdash (q_1, a\$, X\$Z_0) \\
\vdash \overline{(q_1, a\$, TX\$Z_0)} & \vdash \overline{(q_1, a\$, aUX\$Z_0)} & \vdash (q_1, \$, UX\$Z_0) \\
\vdash (q_1, \$, X\$Z_0) & \vdash (q_1, \$, \$Z_0) & \vdash (q_1, \Lambda, Z_0) \\
\vdash (q_2, \Lambda, Z_0) & &
\end{array}$$

At both places that are underlined, U is on top of the stack and a is the next input symbol, but the moves are different. This shows that the grammar is not LL(1).

7.32. The PDA executes a reduction whenever \$, a, or) is on top of the stack. In the case of), the reduction replaces either $(S_1 + S_1)$ or $(S_1 * S_1)$ by S_1; which one is determined entirely by what is on the stack. For any other stack symbol (either S_1 or Z_0), the correct move is to shift.

7.33. (a)

Move no.	State	Input	Stack symbol	Move(s)
Shift moves				
1	q	σ	X	$(q, \sigma X)$
(X any stack symbol other than] or \$)				
Moves to reduce S_1\$ to S				
2	q	Λ	\$	$(q_{0,1}, \Lambda)$
3	$q_{0,1}$	Λ	S_1	(q, S)
Moves to reduce some string to S_1				
4	q	Λ	]	$(q_{1,1}, \Lambda)$
5	$q_{1,1}$	Λ	[	$(q_{1,2}, \Lambda)$
6	$q_{1,2}$	Λ	S_1	(q, S_1)
7	$q_{1,2}$	Λ	X	(q, S_1X)
(X any stack symbol other than S_1)				
8	$q_{1,1}$	Λ	S_1	$(q_{1,3}, \Lambda)$
9	$q_{1,3}$	Λ	[	$(q_{1,4}, \Lambda)$
10	$q_{1,4}$	Λ	S_1	(q, S_1)
11	$q_{1,4}$	Λ	X	(q, S_1X)
(X any stack symbol other than S_1)				
Move to accept				
12	q	Λ	S	(q_{acc}, Λ)
(all other combinations)				none

(b) One reason is that if S_1 is on the stack and the next input symbol is], the next correct move might be a shift (if the S_1 is the one on the right side of the production $S \rightarrow [S_1]$, for example) or a reduction (if the S_1 is the one on the right side of the production $S \rightarrow []S_1$, for example). A string can easily be found so that both these moves occur in the course of accepting the string.

7.36. (a) The pairs (X, σ) for which it is correct to reduce when the top stack symbol is X and the next input is σ are the following: $(\$, \sigma)$ (for any input σ), (a, σ) (for any input σ), $(T, +)$, and $(T, \$)$.

7.37. (a)

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	$(q_1, aZ_0), (q_1, aaZ_0)$
2	q_1	a	a	$(q_1, aa), (q_1, aaa)$
3	q_1	b	a	(q_2, Λ)
4	q_2	b	a	(q_2, Λ)
5	q_2	Λ	Z_0	(q_3, Z_0)
(all other combinations)				none

The initial state is q_0 and the accepting states are q_0 and q_3. Each a is used to cancel either one or two b's.

(b)

Move no.	State	Input	Stack symbol	Move(s)
1	q_0	a	Z_0	(q_1, aZ_0)
2	q_0	b	Z_0	(q_0, bZ_0)
3	q_0	a	b	(q_1, Λ)
4	q_0	b	b	(q_0, bb)
5	q_1	a	Z_0	(q_2, aaZ_0)
6	q_1	b	Z_0	(q_1, bZ_0)
7	q_1	a	a	(q_2, aaa)
8	q_1	a	b	(q_2, a)
9	q_1	b	a	(q_1, Λ)
10	q_1	b	b	(q_1, bb)
11	q_2	a	Z_0	$(q_2, aZ_0), (q_2, aaZ_0)$
12	q_2	a	a	$(q_2, aa), (q_2, aaa)$
13	q_2	a	b	$(q_2, \Lambda), (q_2, a)$
14	q_2	b	a	(q_2, Λ)
15	q_2	b	b	(q_2, bb)
16	q_2	Λ	Z_0	(q_3, Z_0)
(all other combinations)				none

The initial state is q_0 and the accepting state q_3. The first a read will be used to cancel one b, the second will be used to cancel two b's, and subsequent a's can be used either way. (Note that every string in the language has at least two a's.)

The PDA stays in q_0 as long as no a's have been read. If and when the first a is read, it will be used to cancel a single b, either by removing b from the stack, or, if a is the first input symbol, saving it on the stack for a subsequent b to cancel. The state q_1 means that a single a has been read; if and when a second a is read, it will be used to cancel two b's (either by popping two b's from the stack, or by popping one from the stack and pushing an a to be canceled later, or by pushing two a's), and the PDA will go to state q_2. In q_2, each subsequent a will cancel either one or two b's, and the machine can enter q_3 if the stack is empty except for Z_0.

7.38. No. We can take any PDA that accepts a nonregular CFL, add a new nonaccepting state q_n, and allow the PDA to enter q_n immediately. Once it enters q_n, it simply reads the rest of the input and stays in q_n without changing the stack. The moves that allow the PDA to accepts strings are not changed. The effect is that for any input string, there is a rejecting sequence of moves that processes the string without adding any symbols to the stack.

7.39. Yes. We can carry out the construction in the solution to Exercise 7.10, with one addition. There is one additional state s to which any transition goes that would otherwise result in a pair (q, α) with $|\alpha| > k$. All transitions from s return to s. This guarantees that the NFA-Λ functions correctly, because for any input string x that is in L, there is a sequence of moves corresponding to x that never causes the NFA-Λ to enter s.

7.40. Let $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ be a DPDA accepting L. We construct a DPDA $M_1 = (Q_1, \Sigma, \Gamma, q_0', Z_0, A, \delta_1)$ accepting the new language as follows. $Q_1 = Q \cup Q_1$, where Q_1 is a set containing another copy q' of every $q \in Q$. (In particular, the initial state of M_1 is the primed version of q_0.) For each $q' \in Q'$, and each $a \in \Sigma \cup \{\Lambda\}$ and $X \in \Gamma$, $\delta_1(q', a, X) = \delta(q, a, X)'$. (In other words, for inputs that are Λ or ordinary alphabet symbols, the new PDA behaves the same way with the states q' that M does with the original states. In addition, if $q \in A$, then for every stack symbol X, $\delta_1(q', \#, X) = \{(q, X)\}$. Finally, δ_1 agrees with δ for elements of Q and inputs other than $\#$.

What this means is that M_1 acts the same way as M up to the point where the input $\#$ is encountered, except that the new states allow it to remember that it has not yet seen $\#$. Once this symbol is seen, if the current state is q' for some $q \in A$ (i.e., if M would have accepted the current string), then the machine switches over to the original states for the rest of the processing. It enters an accepting state subsequently if and only if both the substring preceding the $\#$ and the entire string except for the $\#$ would be accepted by M.

7.42. Let M be a PDA accepting L by empty stack. We wish to construct M_1 accepting L by final state. M_1 will have an extra state q_f, which will be the accepting state. There are no transitions from q_f. M_1 will also have a new initial state, and the first thing it will do from this state is to insert a new stack symbol U under Z_0 (necessary in order to allow M_1 to move to q_f after M's stack empties). It then enters the initial state of M.

The transitions of M_1 are exactly the same as those of M, for all the states in M, all inputs, and all stack symbols other than U. However, from every $q \in Q$, there is a Λ-transition to q_f if the top stack symbol is U. The new PDA M_1 accepts L, because for any string x that causes M's stack to empty, there is a sequence of moves by which x causes M_1 to move to q_f, and the only way M_1 can enter q_f is for it to have processed a string that causes M's stack to empty.

7.45. Suppose the two stack symbols other than Z_0 are 0 and 1. One approach is simply to encode the kth stack symbol by a string of k 1's, and to use 0's as separators. Auxiliary states can be used during the moves that pop the stack, in order to count the number of 1's, and during the moves that push a sequence of k 1's onto the stack.

Chapter 8

Context-free and Noncontext-free Languages

8.1. (a) Suppose L is a CFL, and let n be the integer in the pumping lemma. Let $u = a^n b^{n+1} c^{n+2}$. Then $u = vwxyz$ for some v, w, x, y, and z satisfying $|wy| > 0$, $|wxy| \leq n$, and $vw^i xy^i z \in L$ for every $i \geq 0$.

First we consider the case when wy contains at least one a. Then since $|wxy| \leq n$, wy can contain no c's. Therefore, vw^2xy^2z has at least $n+1$ a's and exactly $n+2$ c's, which is impossible if the string is in L.

If wy contains no a's, then it must contain either b or c. In this case, $vw^0xy^0z = vxz$ has either fewer than $n+1$ b's or fewer than $n+2$ c's, but in either case exactly n a's. This is also impossible if this string is in L. Thus in either case we have derived a contradiction, and we conclude that L cannot be a context-free language.

(b) Suppose L is a CFL, and let n be the integer in the pumping lemma. Let $u = a^n b^{n^2}$. Then $u = vwxyz$, where the same three conditions on v, w, x, y, and z hold. Let $n_a(wy) = p$ and $n_b(wy) = q$. Then for each i, we have

$$n_a(vw^{i+1}xy^{i+1}z) = n + ip \qquad n_b(vw^{i+1}xy^{i+1}z) = n^2 + iq$$

Using the definition of L, we may conclude that $n^2 + iq = (n + ip)^2$ for every $i \geq 0$. However, it is easy to see that this is impossible. The numbers p and q cannot both be 0, since $|wy| > 0$, and if one of them is 0 we have a contradiction. Suppose both are positive. Then using $i = 1$, we obtain $q = 2pn + p^2$, and using $i = 2$ we obtain $q = 2pn + 2p^2$. These two equations imply that $p = 0$, and we obtain a contradiction. Therefore, L is not a CFL.

(c) Suppose L is a CFL, and let n be the integer in the pumping lemma. Let $u = a^n b^{2n} a^n$. Then $u = vwxyz$, where the same three conditions on v, w, x, y, and z hold. If wy contains at least one a from the first group, then since $|wxy| \leq n$, wy can contain no a's from the second group. Then vxz contains n a's at the end and fewer than n at the beginning, so that it is not in L. Similarly if wy contains at least one a from the second group. The only remaining case is when wy contains no a's, and in this case vxz contains fewer than $2n$ b's. In each case we have a contradiction.

(d) Suppose L is a CFL, and let n be the integer in the pumping lemma. Let $u = a^n b^n c^n$. Then $u = vwxyz$, where the same three conditions on v, w, x, y, and z hold.

If wy contains at least one a, then since $|wxy| \leq n$, wy can contain no c's. Therefore, vw^0xy^0z contains fewer than n a's but exactly n c's, and so it is impossible for this string to be in L. If wy contains no a's, then vw^2xy^2z contains either more than n b's or more than n c's, but exactly n a's. In this case also, the string cannot be in L. Therefore, we have a contradiction.

8.2. Some choices that would not work include a^{2n}, $(ab)^n(ab)^n$, and $a^n b a^n b$.

8.4. Yes. In the application of Ogden's lemma, we can designate as distinguished all the positions of u except those containing semicolons. This eliminates the possibility that wy consists of only a a semicolon.

8.5. (a) Yes. A CFG generating L has productions $S \to aSb \mid T \qquad T \to bTa \mid \Lambda$.

(b) Yes. A CFG generating L has productions $S \to Tb \qquad T \to aTa \mid aTb \mid bTa \mid bTb \mid a$.

(c) No. A proof can be constructed involving the pumping lemma that is very similar to the proof in Example 8.2.

(d) No. Suppose L is a CFG, and let n be the integer in the pumping lemma. Let u be the string $a^nb^na^nb^n$ (i.e., $x = a^nb^n$ and $y = \Lambda$). Then $u = vwxyz$, where $|wy| > 0$, $|wxy| \leq n$, and $vw^ixy^iz \in L$ for every $i \geq 0$.

Suppose first that wy contains either only a's from the first group or only b's from the last group. Then vw^2xy^2z is either $a^{n+i}b^na^nb^n$ or $a^nb^na^nb^{n+i}$ for some $i > 0$, and in neither case can this string be of the form sts for any s with $|s| > 0$.

Otherwise, wy contains either a b from the first group or an a from the second. In this case vw^0xy^0z is either $a^ib^ja^kb^n$ or $a^nb^ia^jb^k$, where in either case i and k are positive and $j < n$. Neither of these strings can be of the form required for L either. Therefore, we have a contradiction, and L is not a CFG.

(e) Yes. See Exercise 7.37(b).

(f) Yes. A PDA (in fact, a counter automaton) can be constructed that accepts the language, following the general model in the solution to Exercise 7.18(b). Roughly speaking, the contents of the stack measures the absolute value of $d(x) = 10n_b(x) - n_a(x)$, and the current state indicates whether the actual value is positive or not.

(g) Yes. The set of balanced strings of parentheses is a DCFL, and it follows from the discussion at the end of Section 8.2 that its complement is also.

8.6. The generalization of Theorem 5.3 says that if L is an infinite CFL, then there are strings v, w, x, y, and z with $|wy| > 0$ and $vw^ixy^iz \in L$ for every $i \geq 0$. The generalization of Theorem 5.4 says that if L is an infinite CFL, the set of integers that are lengths of elements of L contains an infinite arithmetic progression.

Let $L_1 = \{x \in \{a,b,c\}^* \mid n_a(x) = n_b(x) = n_c(x)\}$, $L_2 = \{a^ib^ic^i \mid i \geq 0\}$, and $L_3 = \{a^{n^2} \mid n \geq 0\}$. Theorem 8.1a can be used to show that L_1 is not a CFL, but the generalization of Theorem 5.3 cannot. (There is no guarantee that the string wy doesn't have equal numbers of a's, b's, and c's.) The generalization of Theorem 5.3 can be used to show L_2 is not a CFL, but the generalization of Theorem 5.4 cannot. Finally, the generalization of Theorem 5.4 can be used to show that L_3 is not a CFL. (See Exercise 5.27(h).)

8.7. The proof of Theorem 8.4 actually proves that if M_1 is a DPDA accepting L_1 and M_2 is an FA recognizing L_2, then the PDA M that we constructed is a DPDA accepting $L_1 \cap L_2$.

8.8. (a) Call this language L. Then L is the union of the two languages $\{a^ib^jc^k \mid i \geq j\}$ and $\{a^ib^jc^k \mid i \geq k$, each of which is easily seen to be a CFL. The complement of L in $\{a,b,c\}^*$ is $L' = (\{a\}^*\{b\}^*\{c\}^*)' \cup \{a^ib^jc^k \mid i < j \text{ and } i < k\}$. From the formula $L' \cap \{a\}^*\{b\}^*\{c\}^* = \{a^ib^jc^k \mid i < j \text{ and } i < k\}$ and Theorem 8.4, it follows that if L is a CFL, then so is $\{a^ib^jc^k \mid i < j \text{ and } i < k\}$. However, we can show using the pumping lemma that this is not the case. Suppose $\{a^ib^jc^k \mid i < j \text{ and } i < k\}$ is a CFL and let n be the integer in the pumping lemma. Let $u = a^nb^{n+1}c^{n+1}$. Then $u = vwxyz$, where

the usual conditions hold. If wy contains a's, it can contain no c's, and we may obtain a contradiction by considering vw^2xy^2z. If wy contains no a's, we may obtain a contradiction by considering vxz.

(b) The proof is similar to the preceding proof.

8.9. (a) Suppose $L = \{a^ib^{i+k}a^k \mid k \neq i\}$ is a CFL. Let n be the integer in Ogden's lemma, let $u = a^nb^{2n+n!}a^{n+n!}$, and designate the first n positions of u as the distinguished positions. Then $u = vwxyz$, where wy contains at least one from the first group of a's and $vw^ixy^iz \in L$ for every $i \geq 0$. If either w or y contained both a's and b's, then clearly vw^2xy^2z would not be in L. Also, if neither w nor y contained any b's, then vw^2xy^2z would not preserve the balance between a's and b's required for membership in L. Therefore, w contains only a's from the first group, and y contains only b's, and in fact the lengths of these strings are equal, say p. Now, however, let $i = 1 + n!/p$. Then $vw^ixy^iz = a^{n+(i-1)p}b^{2n+n!+(i-1)p}a^{n+n!} = a^{n+n!}b^{2n+2n!}a^{n+n!}$, which is clearly not an element of L. This contradiction implies that L is not a CFL.

(c) Suppose $L = \{a^ib^ja^i \mid j \neq i\}$ is a CFL. Let n be the integer in Ogden's lemma, let $u = a^nb^{n+n!}a^n$, and designate the first n positions of u as the distinguished positions. Then $u = vwxyz$, where wy contains at least one from the first group of a's and $vw^ixy^iz \in L$ for every $i \geq 0$. The only case in which considering vw^2xy^2z does not easily produce a contradiction is that in which w contains k a's from the first group for some $k > 0$ and y contains k a's from the second group. In this case, however, we can let $i = 1+n!/k$, so that $vw^ixy^iz = a^{n+n!}b^{n+n!}a^{n+n!}$. This string is not in L, and so we also have a contradiction in this case.

8.10. (a) Since finite sets are regular, F is regular, and since the complement of a regular set is regular, F' is regular. Therefore, by Theorem 8.4, $L - F = L \cap F'$ is a CFL.

(b) We use the formula $(L - F) \cup (L \cap F) = L$. If $L - F$ were a CFL, then since $L \cap F$ is finite and therefore a CFL, L would be a CFL.

(c) Here we use the formula $L \cup F - (F - L) = L$. If $L \cup F$ were a CFL, then since $F - L$ is finite, it would follow from (a) that L is a CFL.

8.11. Part (a) is still true in the more general case, but (b) and (c) are not, since in each case we could take F to be Σ^*.

8.12. All three statements are true.

8.13. The DPDA given in the solution to Exercise 7.13(c) has this property. For the input string ab, for example, we have

$$(q_0, ab, Z_0) \vdash (q_0, b, aZ_0) \vdash (q_t, \Lambda, Z_0) \vdash (q_1, \Lambda, Z_0)$$

At the point when the PDA enters q_t, the string is accepted. However, ab is an element of L, not L'.

8.14. Yes. It is not difficult to show that if G is a CFG generating L, then the grammar obtained from G by reversing the right sides of all the productions generates $rev(L)$.

8.15. (a) No. Suppose L is a CFG, and let n be the integer in the pumping lemma. Let $u = b^p a^{2np}$, where p is an integer large enough so that $2n^2/p < 1$ (for reasons that will be clear shortly). Then $u = vwxyz$, where $|wy| > 0$, $|wxy| \leq n$, and $vw^ixy^iz \in L$ for every $i \geq 0$.

The string vw^2xy^2z has $2np + j$ a's and $p + k$ b's, where $0 \leq j \leq n$ and $0 \leq k \leq n$ Consider the ratio n_a/n_b for this string. It is

$$\frac{2np + j}{p + k} = 2n + \frac{j - 2nk}{p + k}$$

The fraction on the right side has absolute value no larger than $2n^2/p$, which by assumption is less than 1. However, it is nonzero. (j and k cannot both be 0; if one is zero, the numerator is clearly nonzero; and if both are nonzero, $j - 2nk \neq 0$ because $j < n$.) Therefore, the original ratio cannot be an integer, which means that the string vw^2xy^2z can't be in L. This contradiction implies that L is not a CFL.

(b) Yes. Given a PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ accepting L, we can construct another PDA $M_1 = (Q_1, \Sigma, \Gamma, q_1, Z_0, A_1, \delta_1)$ accepting the set of prefixes of elements of L as follows. Q_1 can be taken to be $Q \times \{0, 1\}$—in other words, a set containing two copies $(q, 0)$ and $(q, 1)$ of each element of Q. The initial state q_1 is $(q_0, 0)$, and the set A_1 is $A \times \{1\}$. For each combination $(q, 0)$ (where $q \in Q$), $a \in \Sigma$, and $X \in \Gamma$, the set $\delta_1((q, 0), a, X)$ is simply $\{((p, 0), \alpha) \mid (p, \alpha) \in \delta(q, a, X)\}$; however, $\delta_1((q, 0), \Lambda, X)$ contains not only the elements $((p, 0), \alpha)$ for which $(p, \alpha) \in \delta(q, \Lambda, X)$, but one additional element, $((q, 1), X)$. For each $q \in Q$, $a \in \Sigma$, and $X \in \Gamma$, $\delta_1((q, 1), a, X) = \emptyset$. Finally, for each $q \in Q$ and each $X \in \Gamma$, $\delta_1((q, 1), \Lambda, X)$ is the union of the sets $\{((p, 1), \alpha) \mid (p, \alpha) \in \delta(q, a, X)\}$ over all $a \in \Sigma \cup \{\Lambda\}$.

The idea here is that M_1 starts out in state $(q_0, 0)$ and acts on the states $(q, 0)$ exactly the same way M acts on the states q, until such time as M_1 takes a Λ-transition from its current state $(q, 0)$ to $(q, 1)$. From this point on, M_1 can make the same moves on the states $(q, 1)$ that M can make on the states q, changing the stack in the same way, but without reading any input. If $xy \in L$, then for any sequence of moves M makes corresponding to xy, ending in the state r, one possible sequence of moves M_1 can make on xy is to copy the moves of M while processing x, reaching some state $(q_x, 0)$, then make a Λ-transition to $(q_x, 1)$, then continue simulating the sequence on the states $(p, 1)$ but making only Λ-transitions, ending at the state $(r, 1)$ having processed the string x. Therefore, x is accepted by M_1. Conversely, if x is accepted by M_1, there is a sequence of moves corresponding to x that ends at a state $(p, 1)$, where $p \in A$. In this case, the moves of the sequence that involve states $(q, 0)$, ending at $(q_x, 0)$, correspond to moves M can make processing x, ending at the state q_x; and the remaining Λ-transitions that end at $(p, 1)$ correspond to transitions that M can make using the symbols of some string y. Therefore, if M_1 accepts x, then there is a string y so that M accepts xy.

(c) and (d) Yes. The argument in part (i) can be adapted for each of these parts.

(f) No. Suppose L is a CFG, and let n be the integer in the pumping lemma. Let $u = a^p b^p c^p$, where p is a prime larger than n. Then $u = vwxyz$, where $|wy| > 0$, $|wxy| \leq n$,

and $vw^ixy^iz \in L$ for every $i \geq 0$. These conditions imply that wy contains no more than two distinct symbols. Therefore, if $j = n_a(vw^0xy^0z = n_a(vxz)$, $k = n_b(vxz)$, and $m = n_c(vxz)$, at least one the three integers j, k, m is p, at least one is less than p, and all three are positive. Since p is prime, it is not possible for all three to have a nontrivial common factor. Therefore, L is not a CFG.

8.16. As suggested in the hint, let $L_1 = \{x\#y\#z \mid xyz \in L\}$. Then it is easy to see that if there is a PDA M accepting L, there is another one M_1 accepting L_1. (M_1 can simply ignore the # symbols during the processing, except that it enters an accepting state only if it has seen exactly two such symbols during the processing.)

Let n be the integer in Ogden's lemma applied to L_1, and let $u_1u_2u_3$ be any string in L, where $|u_2| \geq n$. Then consider the string $u = u_1\#u_2\#u_3 \in L_1$, and specify all the positions in the substring u_2 to be distinguished. Then according to Ogden's lemma, $u = v'w'x'y'z'$, where $w'y'$ contains at least one distinguished position and $v'(w')^ix'(y')^iz' \in L_1$ for every $i \geq 0$. Suppose that w' contains one of the distinguished positions. Then w' must be a substring of u_2, for otherwise we would have strings with more than two #'s in L_1; similarly if y' contains a distinguished position. Now let v, w, x, y, and z be v', w', x', y', z', respectively, except that any occurrences of # are omitted. Then the strings v, w, x, y, and z satisfy the desired conditions.

8.17. (c) Suppose $L = \{a^ib^ja^i \mid j \neq i\}$ is a CFL. Let n be the integer in Exercise 8.16, let $u = a^nb^{n+n!}a^n$, and let $u_1 = \Lambda$, $u_2 = a^n$, and $u_3 = b^{n+n!}a^n$. Then $u = vwxyz$, where w or y is a nonempty substring of the first group of a's and $vw^ixy^iz \in L$ for every $i \geq 0$. From this point on, the contradiction is obtained in almost the same way as in the solution to Exercise 8.9.

8.18. Let L be the language in Example 8.5, $\{a^ib^ic^j \mid i, j \geq 0 \text{ and } j \neq i\}$. Suppose L is a CFL, and let n be the integer in the statement in Exercise 8.16. Let $u = a^nb^nc^{n+n!}$, as in Example 8.5, and let the string u_2 be a^n. Then there are strings v, w, x, y, and z so that either w or y is a nonnull substring of a^n and $vw^ixy^iz \in L$ for every $i \geq 0$. (Note: the statement in Exercise 8.16 says only that w or y is a substring of u_2, but the proof given in the solution implies that we can require it to be nonnull.) From this point on, the argument follows that in Example 8.5.

Let L_1 be the language in Example 8.6, $\{a^pb^qc^rd^s \mid p = 0 \text{ or } q = r = s\}$. Let n be the integer in Exercise 8.16, let $u = ab^nc^nd^n$, and let u_2 be the substring c^n. Then $u = vwxyz$, where either w or y is a nonnull substring of c^n and $vw^ixy^iz \in L_1$ for every $i \geq 0$. It follows that wy cannot contain all three of the symbols b, c, and d. We can now finish the argument as in Example 8.6.

8.19. The class of DCFLs is not closed under any of these operations.

(a) The languages L_1 and L_2 defined in the proof of Theorem 8.3 are both DCFLs. The language $(L_1 \cup L_2)' \cap \{a\}^*\{b\}^*\{c\}^*$ is $\{a^ib^jc^k \mid i \geq j \text{ and } i \geq k\}$, which can be shown by the pumping lemma not to be a CFL. It follows from Theorem 8.4 that $(L_1 \cup L_2)'$ is not a CFL, and therefore that $L_1 \cup L_2$ is not a DCFL.

(b) We have the formula $L_1 \cup L_2 = (L_1' \cap L_2')'$. If the set of DCFLs were closed under intersection, it would follow from this formula that it was also closed under union.

(c) Again look at the DCFLs L_1 and L_2 in the proof of Theorem 8.3. Let $L_3 = \{d\}L_1 \cup L_2$. Then L_3 is a DCFL, because in order to accept it, the presence or absence of an initial d is all that needs to be checked before executing the appropriate DPDA to accept either L_1 or L_2. The language $\{d\}^*$ is also a DCFL. However, we can check that $\{d\}^*L_3$ is not a DCFL. The reason is that $\{d\}^*L_3 \cap \{d\}\{a\}^*\{b\}^*\{c\}^* = \{d\}L_1 \cup \{d\}L_2$. If $\{d\}^*L_3$ were a DCFL, then this intersection would be also (see Exercise 8.7), and it follows easily that $L_1 \cup L_2$ would be as well.

(d) Let $L_4 = \{d\} \cup \{d\}L_1 \cup L_2$, where L_1 and L_2 are as above. Then L_4 is a DCFL. It is not hard to see that

$$L_4^* \cap \{d\}\{a\}^*\{b\}^*\{c\}^* = \{d\}L_1 \cup \{d\}L_2$$

Therefore, the same argument used in (c) implies that L_4^* is not a DCFL.

(e) We have the formula $A \cap B = A - B'$. If the set of DCFLs were closed under the difference operation, then since it is closed under complements, it would be closed under intersections.

8.20. (a) Let $L_1 = \{x\#y \mid x \in \mathit{pal}$ and $xy \in \mathit{pal}\}$. Then if *pal* were a DCFL, L_1 would be, and therefore, by Exercise 8.7, $L_1 \cap \{0\}^*\{1\}^*\{0\}^*\{\#\}\{1\}^*\{0\}^*$ would be also. However, this intersection is $\{0^i1^j0^i\#1^j0^i \mid i, j \geq 0\}$, and it's easy to show using the pumping lemma that this language is not even a CFL.

(b) Let this language be L, and let $L_1 = \{x\#y \mid x \in L$ and $xy \in L\}$. If L were a DCFL, then L_1 would be, and so $L_1 \cap \{a\}^+\{b\}^*\{\#\}\{a\}^+$ would be. However, consider what it means for a string $a^ib^j\#a^k$ (with $i, j, k > 0$) to be in this intersection. Saying that the prefix up to the # is in L means that j is either i or $2i$, and saying that $a^ib^ja^k$ is in L means that j is either $i + k$ or $2(i + k)$. Since i, j, and k are all positive, the only way both conditions can be satisfied is for j to be equal to both $2i$ and $i + k$—in other words, for the string to be of the form $a^ib^{2i}\#a^i$. The set of all such strings is not a CFL, and so L cannot be a DCFL.

8.21. Every string accepted by M_1 is of the form $x\#y$, where x and xy are in L. If the PDA is nondeterministic, however, the converse may not be true. If x and xy are in L, M can make a sequence of moves on x that leads to an accepting state, and it can make a sequence of moves on xy that leads to an accepting state, but the second sequence may not cause M to be in an accepting state after x.

8.22. An example is the language *pal*. If $L_1 = \{x\#y \mid x, y \in L\}$ were a CFL, then $L_1 \cap \{0\}^*\{1\}^*\{0\}^*\{\#\}\{1\}^*\{0\}^*$ would also be a CFL, by Theorem 8.4. As we observed in the solution to Exercise 8.20, however, it is not.

8.23. To simplify things slightly, let $D = \{i \mid a^i \in L\}$. Then it will be sufficient to show that D is a finite union of (not necessarily infinite) arithmetic progressions of the form

$\{m + ip \mid i \geq 0\}$. (See Exercise 5.51.)

Let n be the integer in the pumping lemma applied to L. Then for any $m \in D$ with $m \geq n$, there is an integer p_m with $0 < p_m \leq n$ for which $m + ip_m \in D$ for every $i \geq 0$. There are only a finite number of distinct p_m's; let p be the least common multiple of all of them. Then for every $m \in D$ with $m \geq n$, $m + ip \in D$ for every $i \geq 0$.

Now, for every k satisfying $0 \leq k < p$, if there is an $m \in D$ with $m \geq n$ and $m \equiv k \bmod p$, then because of the preceding paragraph, the set of all such m form an infinite arithmetic progression. Whether there is or not, the set $\{m \in D \mid m < n \text{ and } m \equiv k \bmod p\}$ is a finite union of (finite) arithmetic progressions. Therefore, the set $\{m \in D \mid m \equiv k \bmod p\}$ is a finite union of arithmetic progressions. Therefore, by taking the union over all k with $0 \leq k < p$, the entire set D has this property.

8.24. We show the "only if" part. Suppose L is a CFL, and let n be the integer in the pumping lemma. For any $m \geq n$, let $u = a^{f(m)}b^m$. Then there are strings v, w, x, y, and z, with w and y not both null and $|wy| \leq n$, so that $vw^ixy^iz \in L$ for every $i \geq 0$. Suppose wy contains p_m b's and q_m a's. Then $vw^{i+1}xy^{i+1}z$ contains $iq_m + f(m)$ a's and $ip_m + m$ b's, and so, since this string is in L, $f(ip_m + m) = iq_m + f(m)$ for every $i \geq 0$. Since p_m and q_m can't both be 0, it follows that $p_m > 0$. Since each p_m satisfies $p_m \leq n$, there are only a finite number of them; let p be the least common multiple of these values. Then for each m, $p = k_m p_m$ for some k_m, so that $f(ip + m) = ik_m q_m + f(m)$.

Now let r be any integer with $0 \leq r < p$. Pick an integer α so that $m = \alpha p + r \geq n$. If we let $j' = j - \alpha$, then for $j' \geq 0$, $f(jp + r) = f(j'p + m) = j'k_m q_m + f(m) = cj + d$, where $c = k_m q_m$ and $d = f(m) - \alpha k_m q_m$.

Chapter 9

Turing Machines

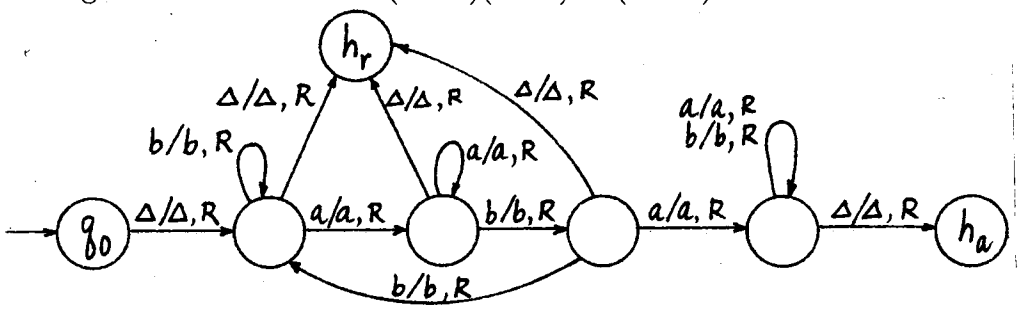
9.1.

$(q_0, \underline{\Delta}aaba) \vdash (q_1, \Delta\underline{a}aba) \vdash (q_2, \Delta A\underline{a}ba) \vdash (q_2, \Delta Aa\underline{b}a) \vdash (q_2, \Delta Aab\underline{a}) \vdash$
$(q_2, \Delta Aaba\underline{\Delta}) \vdash (q_3, \Delta Aab\underline{a}) \vdash (q_4, \Delta Aa\underline{b}A) \vdash (q_4, \Delta A\underline{a}bA) \vdash (q_4, \Delta\underline{A}abA) \vdash$
$(q_1, \Delta A\underline{a}bA) \vdash (q_2, \Delta AA\underline{b}A) \vdash (q_2, \Delta AAb\underline{A}) \vdash (q_3, \Delta AA\underline{b}A) \vdash (q_4, \Delta A\underline{A}BA) \vdash$
$(q_1, \Delta AA\underline{B}A) \vdash (q_5, \Delta A\underline{A}BA) \vdash (q_5, \Delta\underline{A}aBA) \vdash (q_5, \underline{\Delta}aaBA) \vdash (q_6, \Delta\underline{a}aBA) \vdash$
$(q_8, \Delta A\underline{a}BA) \vdash (q_8, \Delta Aa\underline{B}A) \vdash (h_r, \Delta AaB\underline{A})$

9.2. (c) Starting in the initial configuration $(q_0, \underline{\Delta}x)$, where x is an arbitrary string in $\{a, b\}^*$, this TM halts in the configuration $(h_a, \Delta\underline{\Delta}xx^r)$.

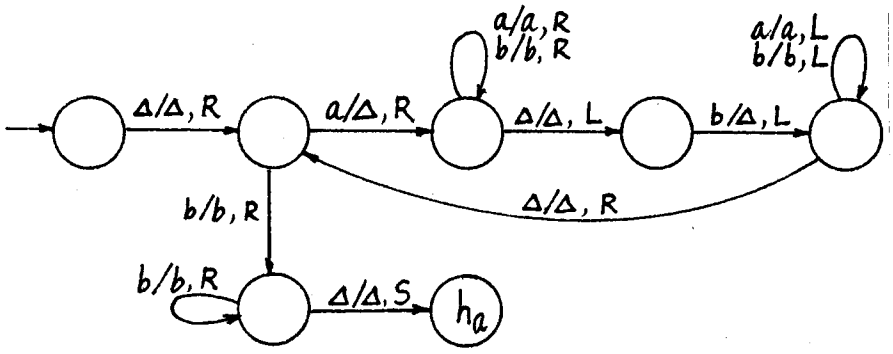
9.3. A configuration is determined by specifying the state, the tape contents, and the tape head position. There are $s + 2$ possibilities for the state, including h_a and h_r; $(t + 1)^{n+1}$ possibilities for the tape contents, since there are $n + 1$ squares, each of which can have an element of Σ or a blank; and $n + 1$ possibilities for the tape head position. The total number of configurations is therefore $(s + 2)(t + 1)^{n+1}(n + 1)$.

9.4.

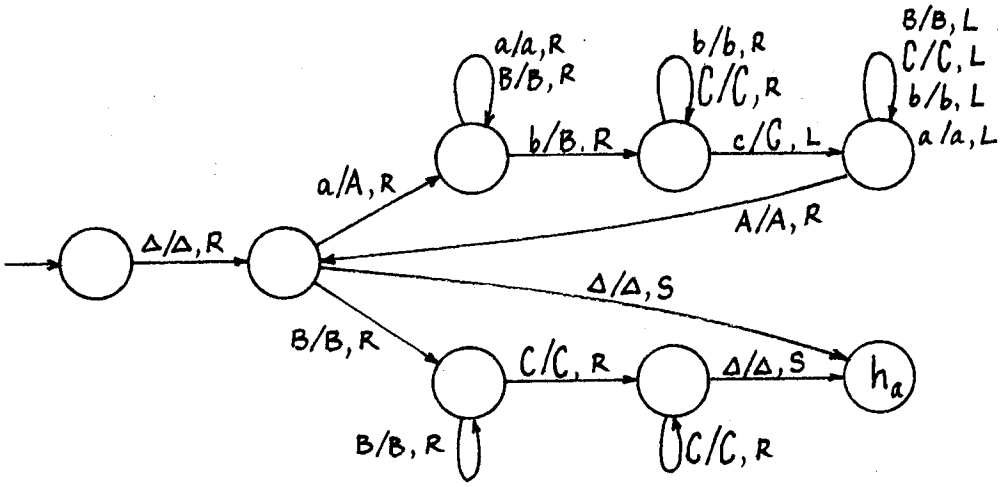


9.5. The technique used in Figure 9.3 can be used in general. There is an introductory move from q_0 to another state that represents the initial state of the FA, which leaves a blank on square 0 and moves the tape head right. The FA transitions are used without change, in the sense that any move on symbol a corresponds to a TM move that leaves a on the tape and moves the tape head right. Finally, from every state that is an accepting state in the FA, there is a transition on the symbol Δ to the state h_a that moves the tape head right.

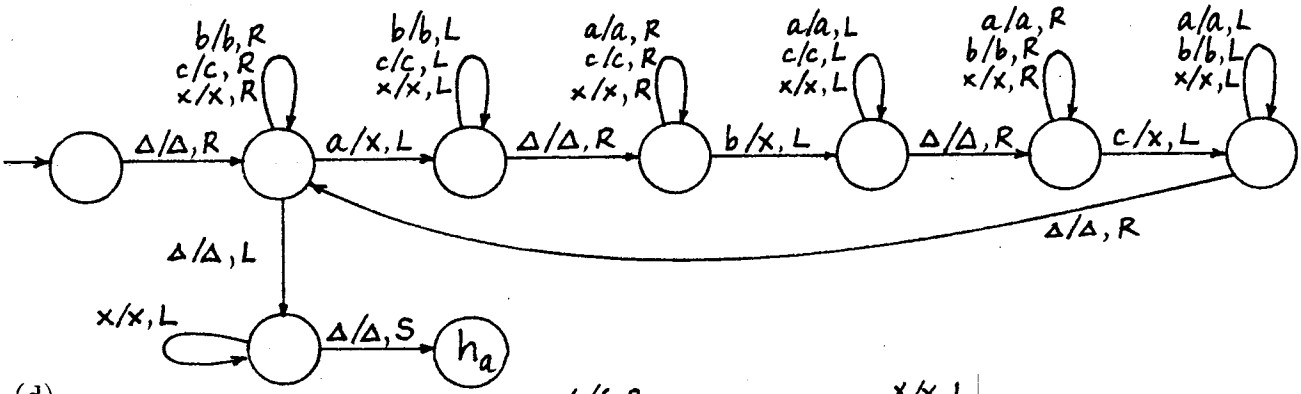
9.6. (a)



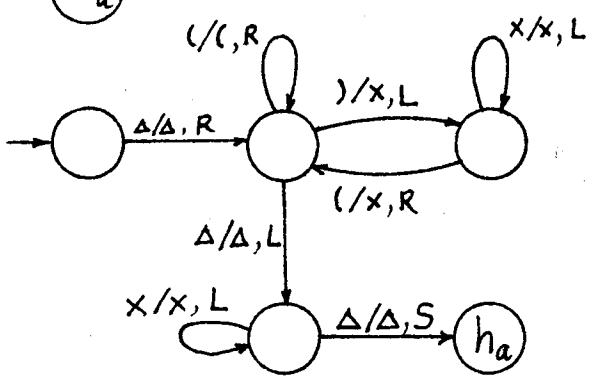
9.6 (b)



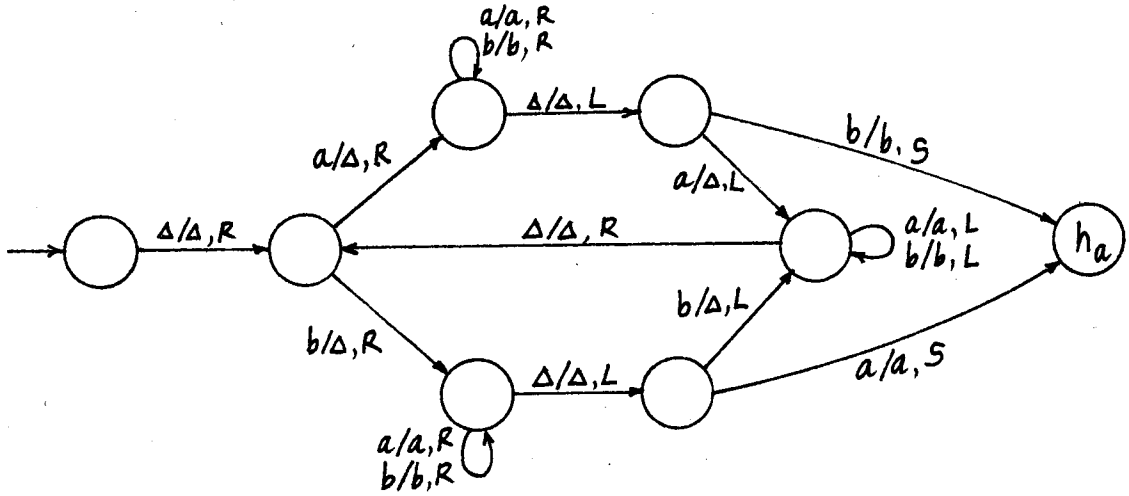
(c)



(d)



(e)



(f) The approach of Example 9.3 can be used, except that in the first phase the point one-third of the way through the string is located, by converting two symbols at the end of the string for every one at the beginning.

9.7. The language $\{1^{2^i} \mid i \geq 0\}$.

9.8. A Λ-transition in an NFA-Λ or a PDA makes it possible for the device to make a move without processing an input symbol. This is unnecessary in a TM because there is already the possibility of a move that leaves the tape head in the same position. (Moves that leave the tape head alone or move it left mean that a TM is not constrained to process the input left-to-right as the simpler machines are.)

9.9. The question is, what does "any obvious modification of it" mean? It is possible to consider a nondeterministic TM that arbitrarily picks a point in the middle of the input string, saves the second part while it processes the first part as T_1 would, and, if and when T_1 would accept, erases the portion of the tape used for that computation and then executes T_2 on the second portion of the original input. (This can also be done without nondeterminism, but it would be more complicated.) Short of this or something like this, however, the approach would not work, because T_1 might accept a string x but never reach the accepting state on a longer string of which x is a prefix, and it might fail to accept x by itself but accept some longer string beginning with x.

9.10. First we relabel states if necessary so that $Q_1 \cap Q_2 = \emptyset$. Then Q is the set $Q_1 \cup Q_2$, the initial state q_0 is q_1, the initial state of Q_1, Σ is Σ_1, and Γ is Γ_2. The transition function δ has the same values as δ_1 for any point of the form (q, a), where $q \in Q_1$ and $\delta_1(q, a) \neq h_a$. (In particular, if T_1 rejects, so does the composite machine.) For any point of the form (q, a) where $q \in Q_2$, $\delta(q, a) = \delta_2(q, a)$. Finally, if $q \in Q_1$ and $\delta_1(q, a) = h_a$, $\delta(q, a) = q_1$.

9.11. Let T_1 be a new TM, which differs from T as follows. It has a new state q not present in T, and for every symbol a, $\delta(q, a) = (q, a, \mathrm{S})$. (The effect is that once T_1 gets to q, it loops forever.) For every combination of state p and tape symbol σ (including Δ) for which T moves to h_r, $\delta(p, \sigma) = (q, \sigma, \mathrm{S})$.

The other way T might end up in h_r is to try to move its tape head off the left end of the tape. T_1 avoids this by executing a preliminary sequence of moves, during which it inserts a special tape symbol \$ in square 0, moving the input string and the preceding blank over by one square. After these preliminary moves, T_1 begins in the configuration $(q_0, \$\underline{\Delta}x)$, where x is the input string. If T_1 ever reads the symbol \$, it enters an infinite loop in which the tape head stays on square 0.

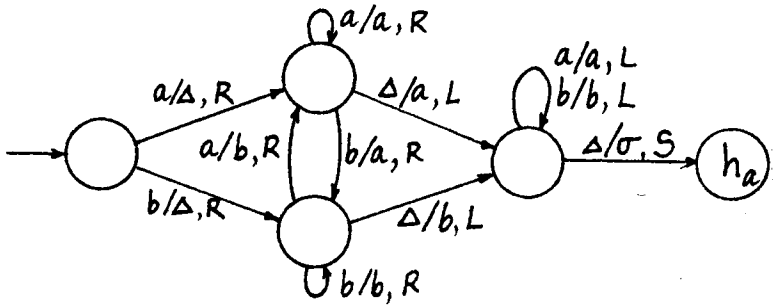
These modifications do not interfere with T's processing of any input string that it accepts, and they do not cause any additional strings to be accepted. Therefore, T_1 accepts the same language as T and never enters the state h_r.

9.12. (a) Suppose there is such a TM T_0, and consider a TM T_1 that halts in the configuration $(h_a, \underline{\Delta}1)$. Let n be the number of the highest-numbered tape square read by

T_0 in the composite machine T_1T_0. Now let T_2 be another TM, which halts in the configuration $(h_a, \underline{\Delta}1\Delta^n1)$. Since the first n tape squares are identical in the final configurations of T_1 and T_2, T_0 will not read the rightmost 1 left by T_2, and so T_2T_0 will not halt with the tape head positioned on the rightmost nonblank symbol left by T_2.

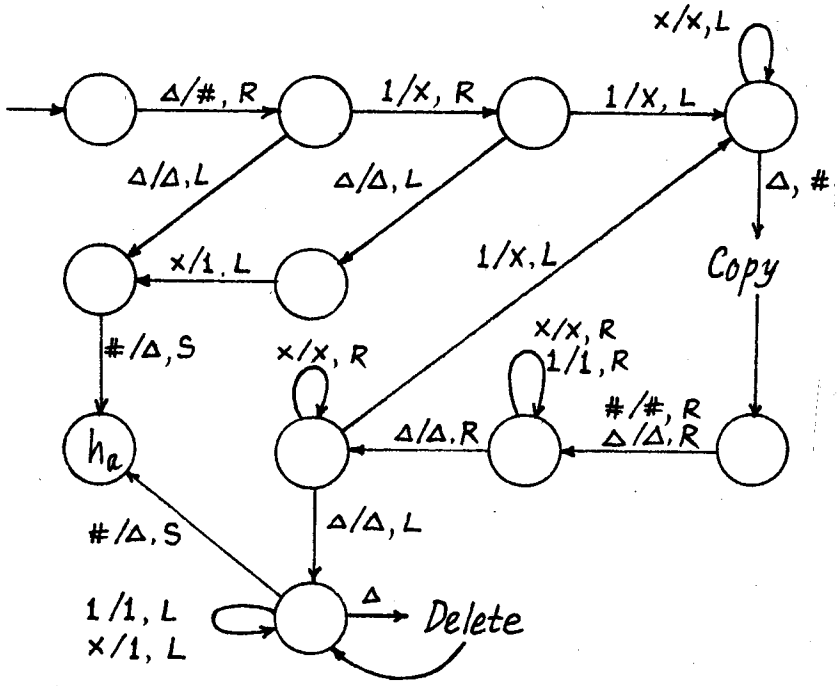
(b) R_T will simulate T, but in such a way that at the end of the processing, the rightmost nonblank symbol can be located. One way to do this is to start by placing a special marker symbol # at the right end of the input string and to return to square 0. During the processing, the marker # will be treated exactly like a blank, except that whenever the tape head reaches the square containing #, this symbol will be moved over one more square to the right. In this way, it continues to mark the rightmost edge of the portion of the tape that has been examined. Once the simulation of T is done, the marker can be erased and the tape head left at the rightmost nonblank symbol preceding it.

9.13.

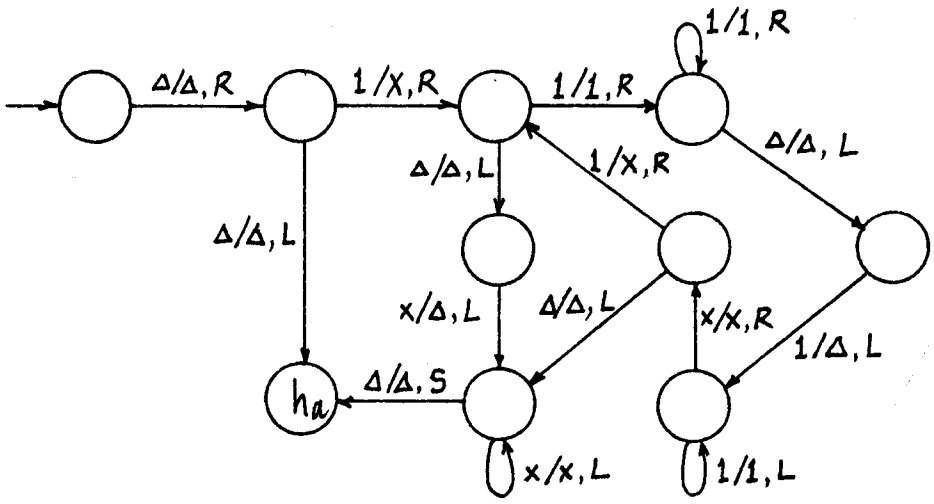


9.14. According to our definition, a TM that accepts an input string but does not leave the tape in the prescribed configuration does not compute a partial function.

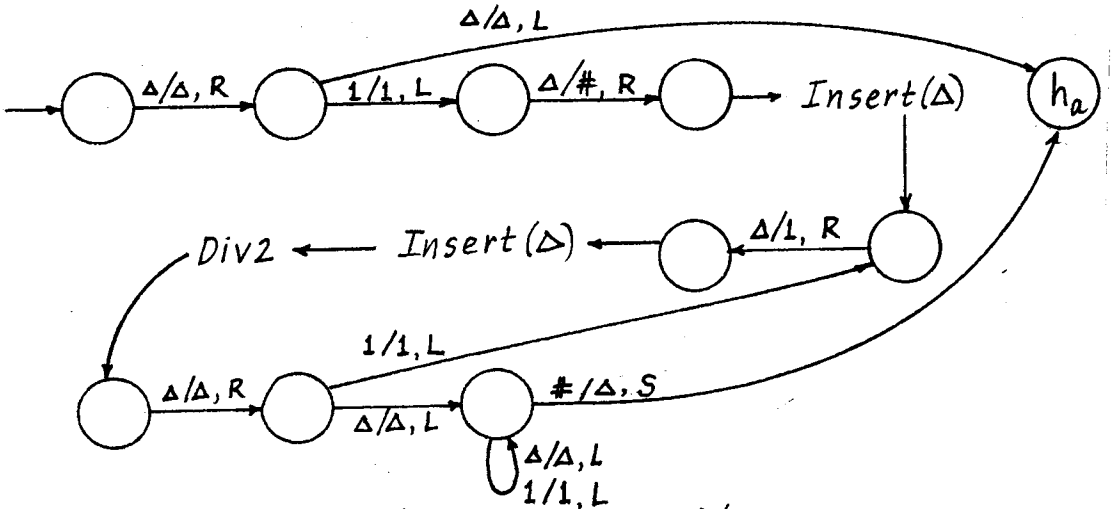
9.15. (c)



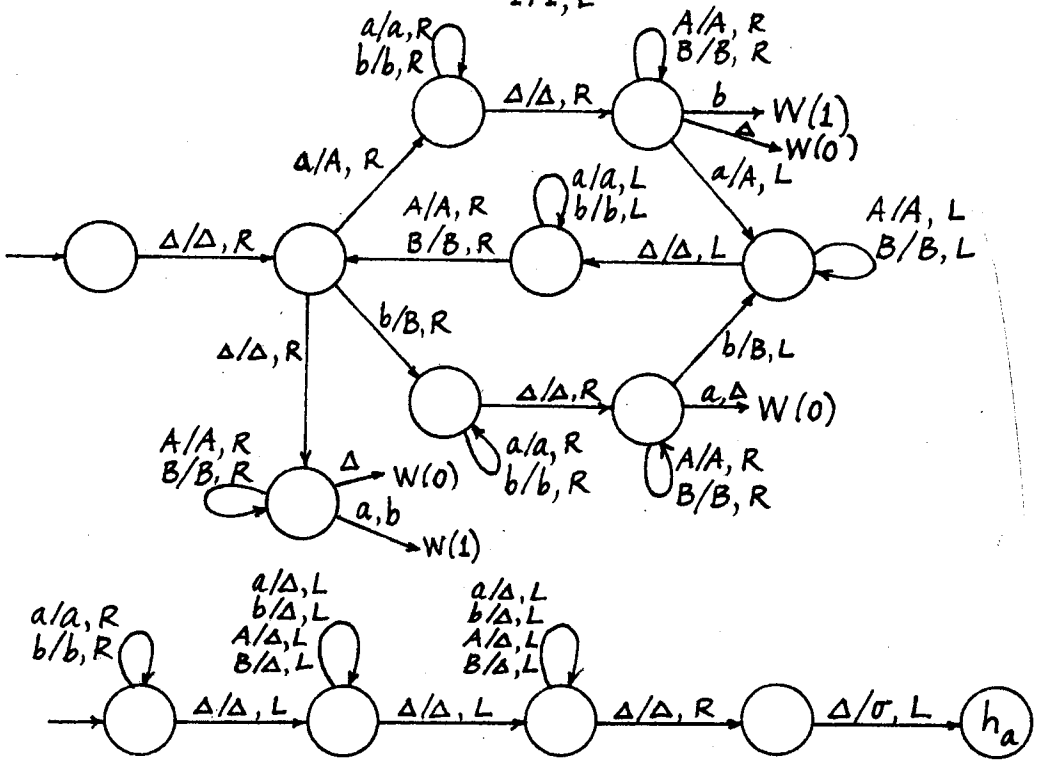
9.15 (d)



(e)



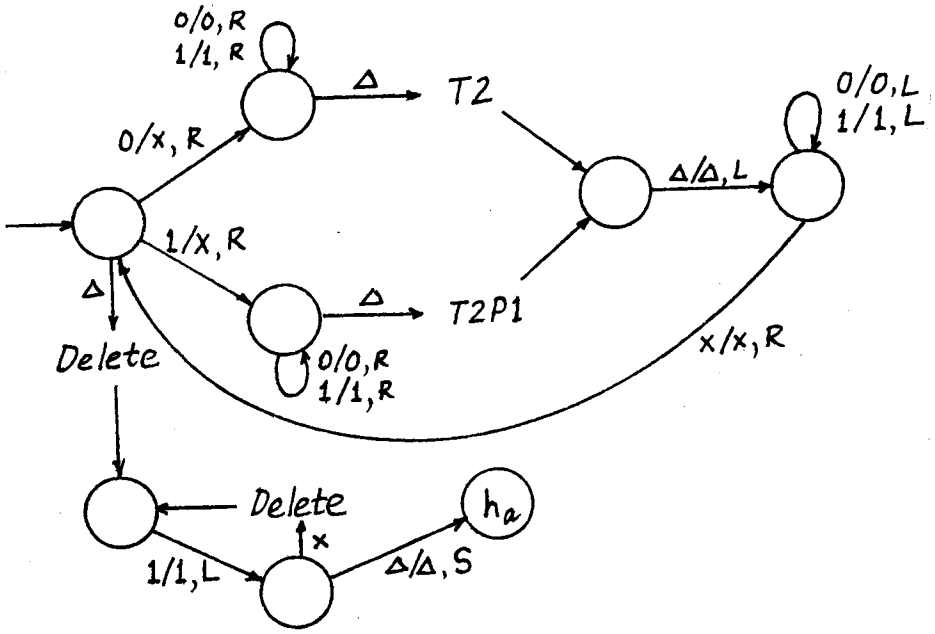
(g)



9.16. See the solution to Exercise 9.15(i).

9.17. In both (a) and (b), a composite TM can start by making a copy of the input, executing T_1 on the copy, interchanging the result with the original input string (which amounts to moving the blank that separates the two the appropriate distance left or right), and executing T_2 on the original input. The result is that the results of T_1 and T_2 are on the tape, separated by a blank. Now it is straightforward to calculate either their sum or their minimum.

9.18. *T2* and *T2P1* compute the functions $n \to 2n$ and $n \to 2n+1$, respectively. The TM below that uses these applies an algorithm that is essentially Horner's rule.



9.19. A pseudocode solution can be expressed as follows. Let `x` denote the input string (or the integer it represents), and `y` the answer.

```
y = 0 (i.e., the null string);
while x is not 0
{  insert the digit x mod 2 at the beginning of y;
   x = x / 2;
}
```

This can be adapted fairly easily to a TM: the string `y` is maintained at the beginning of the tape, and each iteration of the loop consists of a pass through the remaining input to see whether it is even or odd, the insertion of the appropriate symbol at the beginning of `y`, and the execution of the TM in Exercise 9.15(d) on the remaining input.

9.20. One way to modify the diagram is the following. (a) Have the TM insert the symbol # at the beginning of the tape (moving the other tape symbols over) as well as at the end, to simplify the problem of erasing the tape before writing the answer.

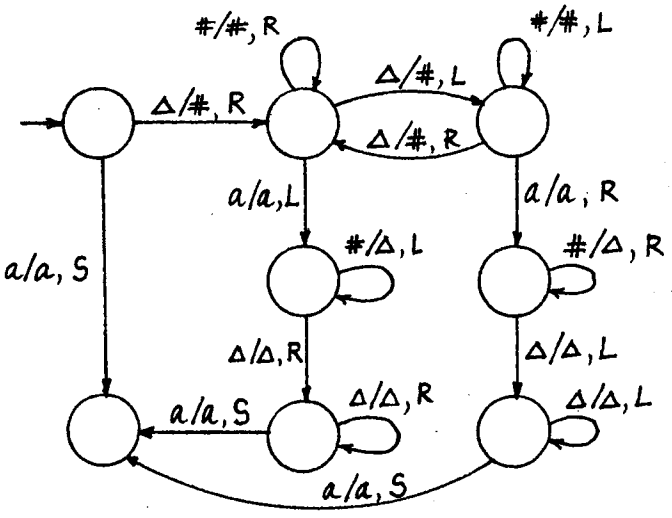
9.20. (b) Replacing the move to the halt state by a sequence of moves that erases the tape, prints 1 on square 1, and halts with the tape head on square 0.

(c) In five places in the diagram where the TM would crash for a string not in the language, add a sequence of moves that would erase the tape, print 0 on square 1, and halt with the tape head on square 0. These five places are: i) the state at the top of the diagram that is third from the right, if the symbol encountered is b or c when it should be a; ii) the state below this one, if the symbol is a or c when it should be b; iii) the state to the left of the one in i), if Δ is encountered instead of c; iv) the state just below this one, in the same situation; v) the state just above the halt state, if anything other than Δ or # is encountered in the loop.

9.21. Here is a verbal description of one solution. Start by inserting an additional blank at the beginning of the tape, to obtain $\Delta\underline{\Delta}x$ (where x is the input). Make a sequence of passes. The first moves the first symbol of x one square to the left and the last symbol one square to the right; the second moves the second symbol of x one square to the left and the next-to-last symbol one square to the right; etc. If x is of odd length, reject. The tape now looks like $\Delta x_1 \Delta\Delta x_2$, where x_1 and x_2 are the first and second halves of x. Now begin comparing symbols of x_1 and x_2, but replacing symbols that have been compared by blanks, rather than uppercase letters as in Example 9.3. In order for this to work, it is necessary to check before each pass to see whether the symbols that are about to be compared are the last symbols in the respective halves. (This prevents an attempt to perform another pass when there are no more nonblank symbols, which would attempt to move the tape head off the tape.)

9.22. The move $\delta(p, a) = (q, b, \mathrm{S})$ can be simulated by using the moves $\delta(p, a) = (p_1, b, R)$ and $\delta(p_1, \sigma) = (q, \sigma, L)$ for any $\sigma \in \Gamma \cup \{\Delta\}$. Here p_1 is a new state used only for this purpose, and the moves shown are the only ones in which it is involved.

9.24.



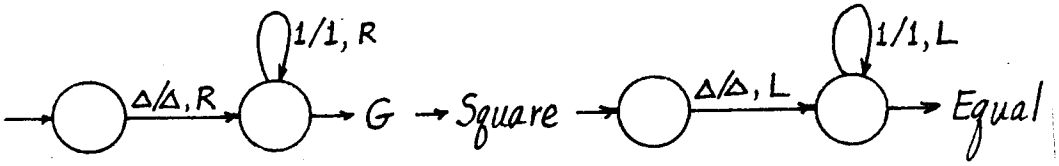
9.25. Given a TM T, a TM T_1 with a doubly infinite tape can be constructed to work as follows: starting in the initial configuration $(q_0, \underline{\Delta}x)$, with the tape head on square 0, it places a special symbol # in square -1, returns the tape head to square 0, and executes

T, except that if it ever reads the symbol # it rejects. It is easy to see that T_1 has the properties specified in the exercise.

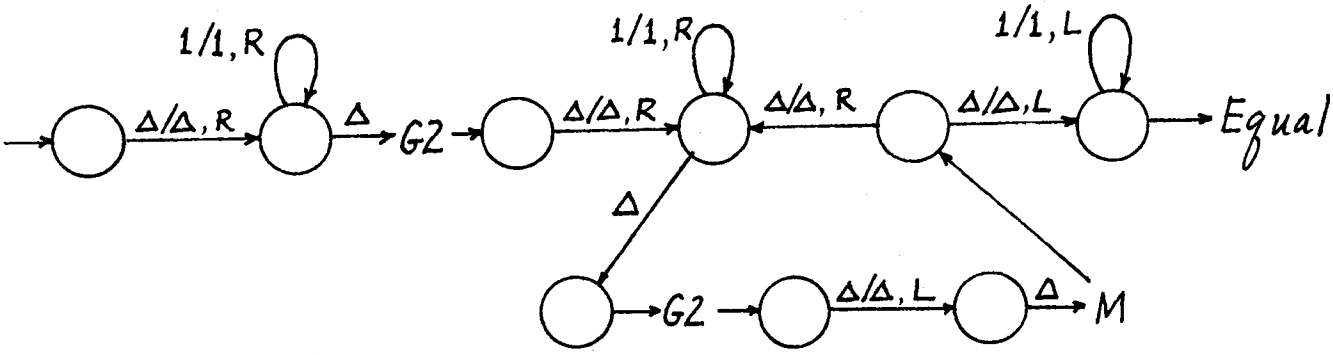
9.29. The TM begins moving its tape head to the right, writing a 0 or a 1 on each square as it goes. It either continues this forever or halts in the configuration $(h_a, \underline{\Delta}x)$, where x is some element of $\{0,1\}^*$. Here x is arbitrary (i.e., for any x, some computation of the TM causes it to halt with x on the tape).

9.30. The language $\{xx \mid x \in \{0,1\}^\}$.

9.31. In the diagram, G is the TM described in Exercise 9.29, *Square* is the TM of Exercise 9.15(c). and *Equal* is as in Exercise 9.30.

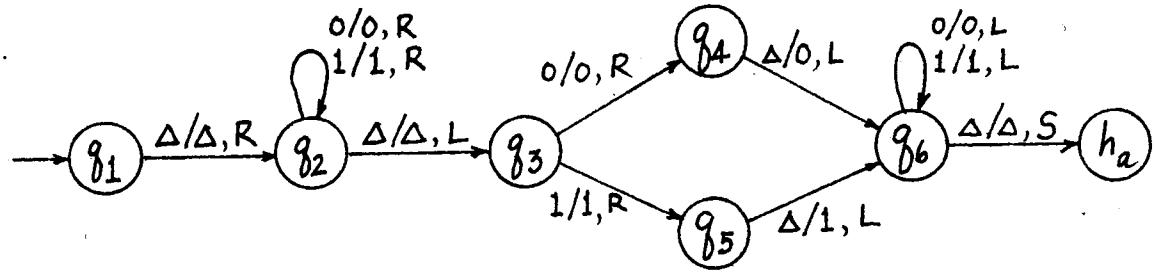


9.32. In the diagram, $G2$ is the same as G in Exercise 9.29 except that it always leaves at least two nonblank symbols on the tape. The TM M computes the multiplication function: it transforms the tape $\underline{\Delta}1^i\Delta 1^j$ to the tape $\underline{\Delta}1^{i*j}$. The TM shown is nondeterministic in that $G2$ is, but also because the steps shown on the lower level can be executed an arbitrary number of times after the first time.

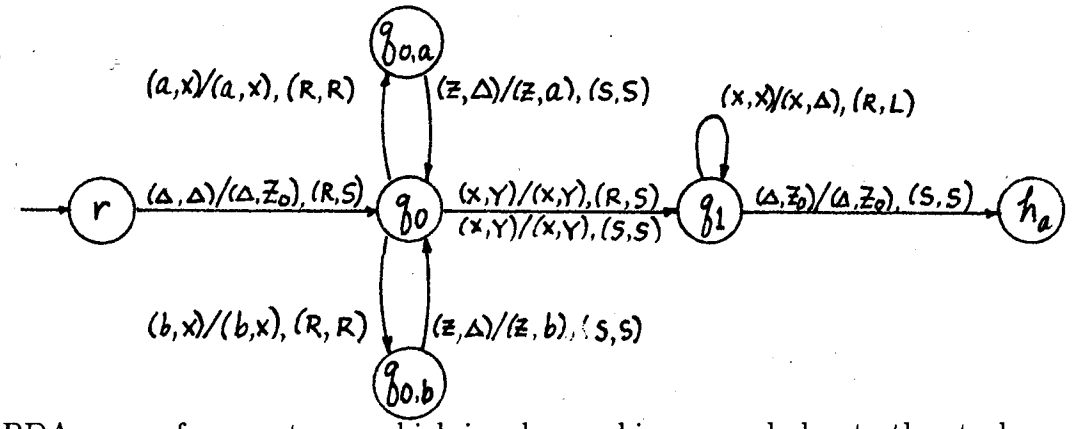


9.33. To accept the set of prefixes of elements of L, for example, a TM could begin by executing the nondeterministic TM G in Exercise 9.29, concatenating the result onto the end of the input string, and executing T on the result. A similar approach works for parts (b) and (c).

9.35.



9.37.



The PDA moves from q_0 to q_0, which involve pushing a symbol onto the stack, require two TM moves, since the TM needs to change a symbol other than the current one on the tape representing the stack. The TM begins by moving to the state corresponding to the initial PDA state q_0, moving the tape head on tape 1 one square to the right, and replacing Δ by Z_0 in the first square of tape 2. The labels on the transitions from q_0 to $q_{0,a}$ and $q_{0,b}$ actually each represent three labels: in both cases, the symbol x can be either a, b, or Z_0 (but is the same in both occurrences within the label). Similarly, on the labels from these two states back to q_0, z can be either a or b but is the same in both occurrences within the label. Each label on the transition from q_0 to q_1 represents six transitions: x can be either a or b, and y can be either a, b, or Z_0 (but all occurrences of x within a label represent the same symbol, and the same for y). Finally, the label on the transition from q_1 to q_1 represents two labels: x can be either a or b.

For example, the moves by which the input string aba is accepted are shown below.

$$(r, \underline{\Delta}aba, \underline{\Delta}) \vdash (q_0, \Delta\underline{a}ba, \underline{Z_0}) \vdash (q_{0,a}, \Delta a\underline{b}a, Z_0\underline{\Delta}) \vdash (q_0, \Delta a\underline{b}a, Z_0\underline{a})$$
$$(q_1, \Delta ab\underline{a}, Z_0\underline{a}) \vdash (q_1, \Delta aba\underline{\Delta}, \underline{Z_0}) \vdash (h_a, \Delta aba\underline{\Delta}, \underline{Z_0})$$

The moves by which aa is accepted are as follows.

$$(r, \underline{\Delta}aa, \underline{\Delta}) \vdash (q_0, \Delta\underline{a}a, \underline{Z_0}) \vdash (q_{0,a}, \Delta a\underline{a}, Z_0\underline{\Delta}) \vdash (q_0, \Delta a\underline{a}, Z_0\underline{a})$$
$$(q_1, \Delta a\underline{a}, Z_0\underline{a}) \vdash (q_1, \Delta aa\underline{\Delta}, \underline{Z_0}) \vdash (h_a, \Delta aa\underline{\Delta}, \underline{Z_0})$$

9.38. The iterative step can be described as follows. If x is the last palindrome written on the tape, then x is copied, so that the last part of the nonblank portion of the tape looks like $\Delta x \Delta x$. If x consists entirely of 1's, then the second copy is then changed to the string $0^{|x|+1}$. Otherwise, the last 0 in the first half of the copy is located. (We take the "first half" to include the middle symbol if $|x|$ is odd.) It is changed to 1, and all the 1's after it within the first half are changed to 0. The second half is then modified so as to make the string a palindrome.

9.39. (a) If the tape head does not move past square n, the possible number of nonhalting configurations is $s(t+1)^{n+1}(n+1)$, where s and t are the sizes of Q and Γ, respectively (see the solution to Exercise 9.3). Within that many moves, therefore, the TM will have

either halted or repeated a nonhalting configuration, and so if it has not halted you may conclude it is in an infinite loop.

(b) As in part (a), let s be the size of the set Q. Suppose you watch the TM until it moves to the first square after the input string, and continue to watch for s more moves. If the tape head has moved right each time, then either it has encountered the combination (q, Δ) twice, for some state q, or the tape head moves right in every possible combination (q, Δ). In either case, the TM is in an infinite loop.

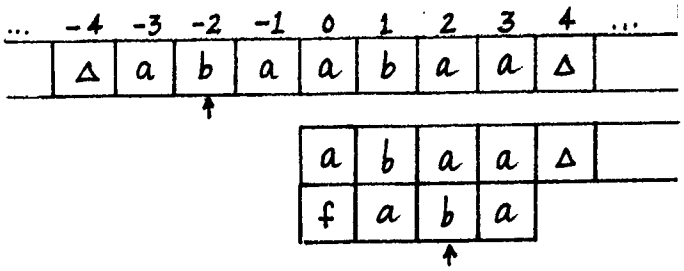
9.40. (a) In this case, the result for an input string x depends only on the first $n+1$ symbols of x. This means that if $S_1 = \{x \in L(T) \mid |x| < n+1\}$ and $S_2 = \{x \in L(T) \mid |x| = n+1\}$, then $L(T) = S_1 \cup S_2\Sigma^*$, which is a regular language.

(b) Once the tape head has passed the last symbol of the input, the TM will never examine the input string again. This means that whether it ever reaches the accepting state depends only on the state it is in when it reaches the first blank square after the input; i.e., there is a set A of states (possibly including h_a) so that for any x, x is accepted if and only if the TM is in a state in A when it reaches the first blank square after the input. Therefore, we may consider the finite automaton $M = (Q, \Sigma, q_1, A, \rho)$, where q_1 is the state T is in after the first move, and for any state $q \in Q$ and $a \in \Sigma$, $\rho(q, a)$ is the state to which T moves on state-symbol combination (q, a). (If $h_a \in A$, then we may add this state to Q and let $\rho(h_a, \sigma) = h_a$ for every $\sigma \in \Sigma$.) The strings accepted by T are precisely those accepted by M, and thus $L(T)$ is accepted by an FA.

9.41. Yes. It follows from the assumption on T that there is a fixed integer n so that for every i, once the tape head reaches square i, it cannot move farther to the left than square $i-n$. This means that we can build another TM T_1 that mimics T but never moves its tape head to the left, by allowing T_1 always to remember the contents of the n squares preceding the current one. Furthermore, it is unnecessary for T_1 to make a move that leaves the tape head stationary, because by examining the transition table for T we can predict the contents of the current tape square and the state to which it goes when it finally moves the tape head right. The conclusion is that there is a TM that accepts the same language as T but always moves its tape head to the right. Exercise 9.40(b) then implies that $L(T)$ is regular.

9.42. We use the first approach. At any point where T has moved its head k squares and no further beyond square 0, either to the left or to the right, T_1's tape will show the first k negative squares as "folded over," in the sense that the portion of the tape from 1 to k will contain two tracks, the first representing the squares from 1 to k and the second those from -1 to $-k$. Square 0 will also contain an extra symbol representing the fold, so that any time T's head crosses from one side of the tape to the other, T_1's head can change from one track to the other.

The following diagram shows what we imagine the two tapes to look like at some point. If T is currently scanning square -2, T_1 will be scanning square 2, paying attention to the lower part only.



The alphabet of T_1 contains not only symbols of Γ but pairs of symbols of Γ, as well as pairs of the form (σ, f), where $\sigma \in \Gamma$ and f represents the fold. (In fact, the alphabet will be even a little larger, as we will see.) The states will be of the form (q, r) and (q, l), where $q \in Q$. The second entries indicate whether the current square of T's tape is on the right side or the left.

T_1 begins by placing the fold marker f in square 0 and moving to state (q_0, r), where it begins to simulate the moves of T. During this simulation, whenever T would move the tape head, T_1 does also, except that if it is currently in a state (q, l) it moves in the opposite direction, and whenever it encounters f it may change tracks. Each time T_1 moves farther from square 0 than it has gone up to that point, it sees a single symbol rather than a pair; at this point it extends the folded portion by converting the single symbol to a pair before continuing.

T_1 starts with the move $\delta(q_1, \Delta) = ((q_0, \mathrm{r}), (\Delta, \mathrm{f}), \mathrm{S})$. For each move $\delta(p, X) = (q, Y, D)$ made by T, we must allow the following moves of T_1. To simplify the notation, let $-D$ denote the direction opposite to D. First, for every $Z \in \Gamma \cup \{\Delta\}$,

$$\begin{aligned} \delta_1((p, \mathrm{r}), (X, Z)) &= ((q, \mathrm{r}), (Y, Z), D) \\ \delta_1((p, \mathrm{l}), (Z, X)) &= ((q, \mathrm{l}), (Z, Y), -D) \end{aligned}$$

Second, $\delta_1((p, \mathrm{r}), (X, \mathrm{f}))$ and $delta_1((p, \mathrm{l}), (X, \mathrm{f})$ are both $((q, \mathrm{l}), (Y, \mathrm{f}), \mathrm{R})$ if $D = \mathrm{L}$ and $((q, \mathrm{r}), (Y, \mathrm{f}), D)$ otherwise. In addition, for each move $\delta(p, X) = (q, Y, D)$, T_1 has the moves $\delta_1((p, \mathrm{r}), X) = ((q, \mathrm{r}), (Y, \Delta), D)$ and (if $X = \Delta$) $\delta_1((p, \mathrm{l}), \Delta) = ((q, \mathrm{l}, (\Delta, Y), -D)$. These moves allow T_1 to simulate faithfully the moves of T up to the point where it halts.

If and when T accepts, T_1 moves instead to some state that still allows it to remember which track of the tape it is supposed to be looking at. Before it can halt, it must fix up its tape so that it is really a duplicate of T's, with the obvious exception that the nonblank symbols will begin in square 0 or somewhere to the right. This essentially means "unfolding" the tape. We can do this by adding extra alphabet symbols of the form (σ, γ'), (σ', γ), and (σ', f), where $\sigma, \gamma \in \Gamma \cup \{\Delta\}$. The primes indicate the current head position and simplify the problem of finding this position later. The unfolding can be done in a sequence of passes, where each pass deletes one symbol from the second track of the tape and inserts it at the beginning of the first. When the second track has been eliminated, the f is deleted, the final position of the tape head is located, and the symbol there is changed back to its original form.

9.44. If we assume that M_1 reads the entire input string, then we can assume that there will be no nonblank symbols farther to the right than the nth square. This means that the

preliminary insertion and final deletion of the symbol # can be done in a total of approximately $4n$ moves, the final pass converting symbol pairs to single symbols can be done in approximately $2n$ moves, and moving to the final tape position requires no more than n moves. The total so far is therefore about $7n$. To simulate a single move in the way that's described in the proof requires three iterations of the following process: move the tape head to the current location on one of the tracks, make some fixed number of moves, and move the tape head back. Since by the ith move, the current location on either track can't be farther right than square i, these three iterations require at most $6i$ moves, plus some fixed number independent of i. The total of these is therefore bounded by

$$\sum_{i=1}^{n}(6i + C) = 3n(n + 1) + Cn$$

The overall total is therefore a quadratic function of n: $3n^2 + 7n + Cn$, where C is a fairly small constant.

9.46. See the proof of Lemma 11.1.

9.47. See the proof of Theorem 10.2.

9.48. For each string of digits of length k that specifies a sequence of moves of the nondeterministic TM to be tried, the number of moves made by T_2 is roughly proportional to $n+k$: n because of copying the input string, k because of executing the sequence of moves, erasing the tape, and calculating the next sequence. For each k, there are 2^k sequences of k moves. So the total number of moves of T_2 is, very approximately, $\sum_{k=1}^{n_x}(n + k)2^k$. This is roughly proportional to $(n + n_x)2^{n_x+1}$.

9.49. We give an informal description of one such machine. For each initial a that it reads, it pushes an x onto the first stack. When it encounters a b, it processes each b by removing an x from stack 1 and placing it on stack 2. When it encounters a c, it checks that stack 1 is empty and transfers the contents of stack 2 to stack 1. It then processes c's the same way it processed b's. This approach can obviously be generalized to any number of distinct symbols.

9.51. The machine makes a sequence of passes. On the first pass, it moves symbols from the front to the rear, except that the first a, b, and c are replaced by A, B, and C, respectively. It crashes if it detects an illegal pair of symbols (ba, cb, ac, or ca), or if it finds one but not all three of the lowercase symbols. On each subsequent pass, it processes the first remaining a, b, and c similarly, crashing if it finds one but not all of the three. It accepts if on some pass there are no remaining a's, b's, or c's.

9.52. The idea that allows a Post machine to simulate a move to the left on the tape is to remember the last two symbols that have been removed from the front of the queue. We illustrate by an example. Suppose that the initial contents of the queue are $abaab$, with the

a at the front. The machine starts by placing a marker X at the rear, to produce $abaabX$. It proceeds to remove symbols from the front, remembering the last two, and placing the symbols at the rear but lagging behind by one. Thus after two steps, the contents are $aabXa$, and the machine remembers that the last symbol removed was b. After three more steps, the contents are $Xabaa$, and the machine remembers that the last symbol was b. When it reads X, it places X on the rear immediately, followed by b, to produce $abaaXb$. At this point, it begins simply moving symbols from the front to the rear, and continues until it has removed the X. If it fails to replace it, the contents of the queue are now $babaa$, and the move to the left has been effectively simulated.

Chapter 9

Turing Machines

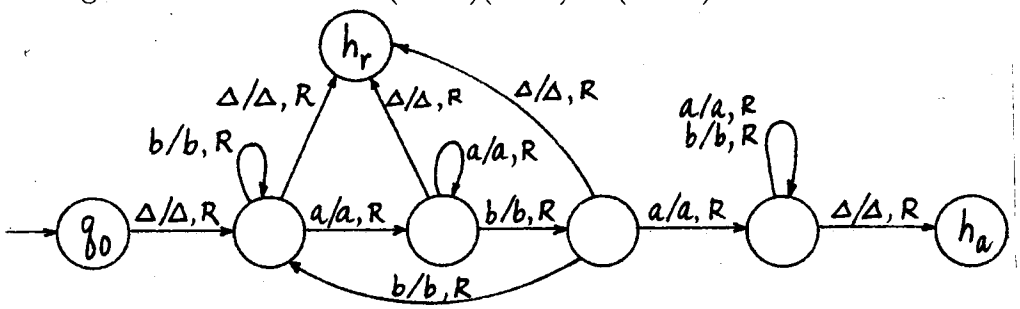
9.1.

$(q_0, \underline{\Delta}aaba) \vdash (q_1, \Delta\underline{a}aba) \vdash (q_2, \Delta A\underline{a}ba) \vdash (q_2, \Delta Aa\underline{b}a) \vdash (q_2, \Delta Aab\underline{a}) \vdash$
$(q_2, \Delta Aaba\underline{\Delta}) \vdash (q_3, \Delta Aab\underline{a}) \vdash (q_4, \Delta Aa\underline{b}A) \vdash (q_4, \Delta A\underline{a}bA) \vdash (q_4, \Delta\underline{A}abA) \vdash$
$(q_1, \Delta A\underline{a}bA) \vdash (q_2, \Delta AA\underline{b}A) \vdash (q_2, \Delta AAb\underline{A}) \vdash (q_3, \Delta AA\underline{b}A) \vdash (q_4, \Delta A\underline{A}BA) \vdash$
$(q_1, \Delta AA\underline{B}A) \vdash (q_5, \Delta A\underline{A}BA) \vdash (q_5, \Delta\underline{A}aBA) \vdash (q_5, \underline{\Delta}aaBA) \vdash (q_6, \Delta\underline{a}aBA) \vdash$
$(q_8, \Delta A\underline{a}BA) \vdash (q_8, \Delta Aa\underline{B}A) \vdash (h_r, \Delta AaB\underline{A})$

9.2. (c) Starting in the initial configuration $(q_0, \underline{\Delta}x)$, where x is an arbitrary string in $\{a, b\}^*$, this TM halts in the configuration $(h_a, \Delta\underline{\Delta}xx^r)$.

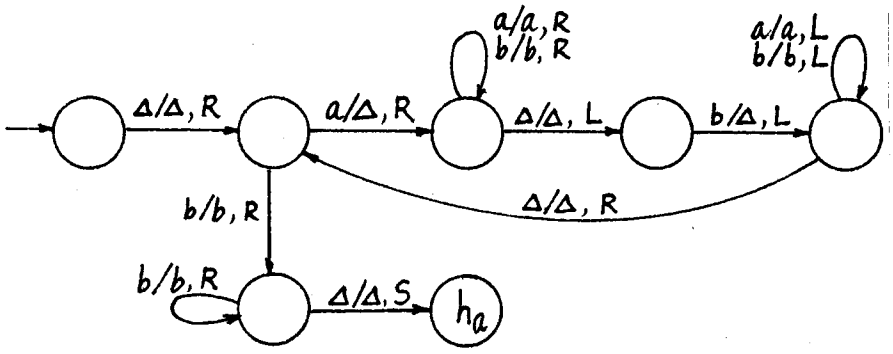
9.3. A configuration is determined by specifying the state, the tape contents, and the tape head position. There are $s + 2$ possibilities for the state, including h_a and h_r; $(t + 1)^{n+1}$ possibilities for the tape contents, since there are $n + 1$ squares, each of which can have an element of Σ or a blank; and $n + 1$ possibilities for the tape head position. The total number of configurations is therefore $(s + 2)(t + 1)^{n+1}(n + 1)$.

9.4.

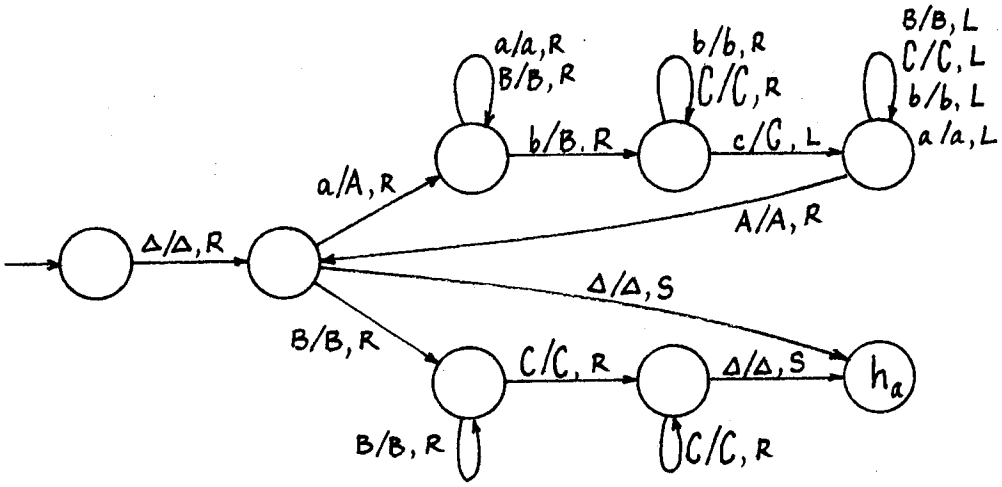


9.5. The technique used in Figure 9.3 can be used in general. There is an introductory move from q_0 to another state that represents the initial state of the FA, which leaves a blank on square 0 and moves the tape head right. The FA transitions are used without change, in the sense that any move on symbol a corresponds to a TM move that leaves a on the tape and moves the tape head right. Finally, from every state that is an accepting state in the FA, there is a transition on the symbol Δ to the state h_a that moves the tape head right.

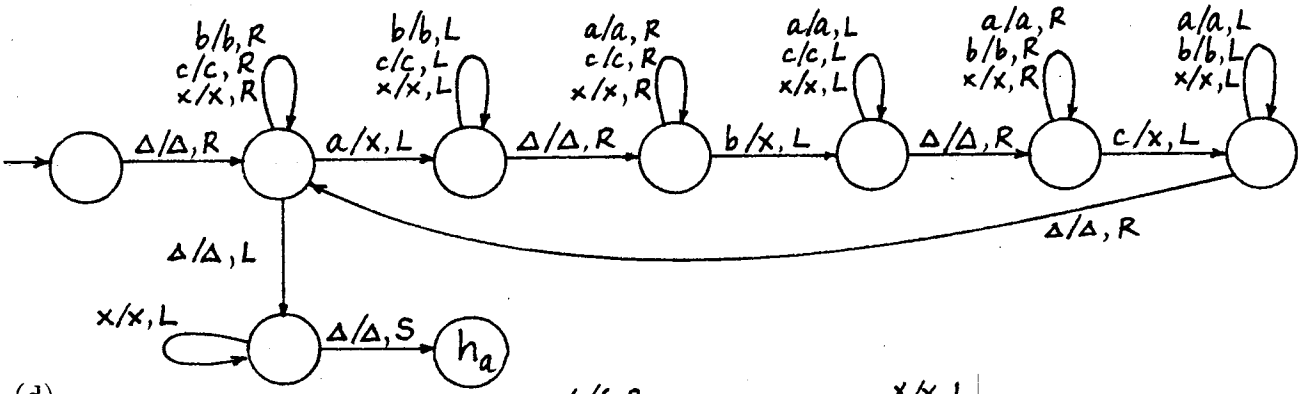
9.6. (a)



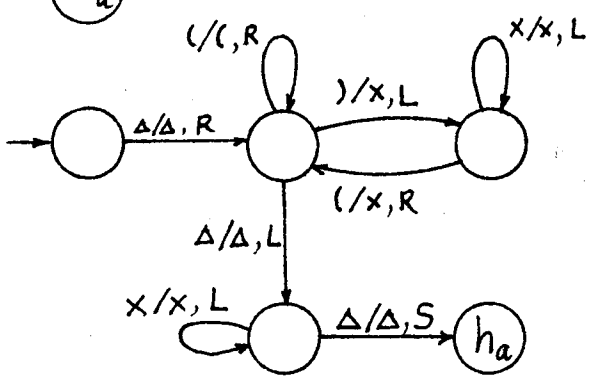
9.6 (b)



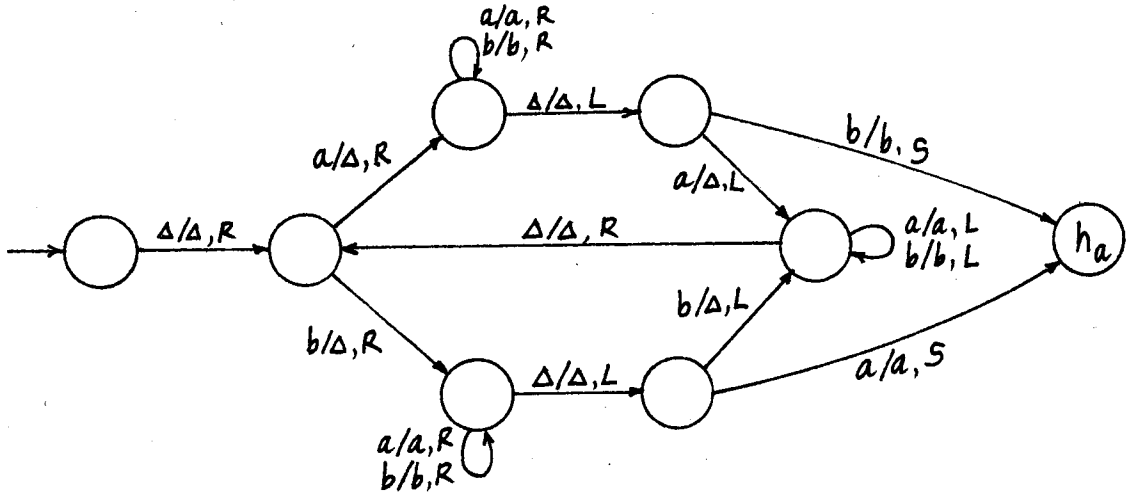
(c)



(d)



(e)



(f) The approach of Example 9.3 can be used, except that in the first phase the point one-third of the way through the string is located, by converting two symbols at the end of the string for every one at the beginning.

9.7. The language $\{1^{2^i} \mid i \geq 0\}$.

9.8. A Λ-transition in an NFA-Λ or a PDA makes it possible for the device to make a move without processing an input symbol. This is unnecessary in a TM because there is already the possibility of a move that leaves the tape head in the same position. (Moves that leave the tape head alone or move it left mean that a TM is not constrained to process the input left-to-right as the simpler machines are.)

9.9. The question is, what does "any obvious modification of it" mean? It is possible to consider a nondeterministic TM that arbitrarily picks a point in the middle of the input string, saves the second part while it processes the first part as T_1 would, and, if and when T_1 would accept, erases the portion of the tape used for that computation and then executes T_2 on the second portion of the original input. (This can also be done without nondeterminism, but it would be more complicated.) Short of this or something like this, however, the approach would not work, because T_1 might accept a string x but never reach the accepting state on a longer string of which x is a prefix, and it might fail to accept x by itself but accept some longer string beginning with x.

9.10. First we relabel states if necessary so that $Q_1 \cap Q_2 = \emptyset$. Then Q is the set $Q_1 \cup Q_2$, the initial state q_0 is q_1, the initial state of Q_1, Σ is Σ_1, and Γ is Γ_2. The transition function δ has the same values as δ_1 for any point of the form (q, a), where $q \in Q_1$ and $\delta_1(q, a) \neq h_a$. (In particular, if T_1 rejects, so does the composite machine.) For any point of the form (q, a) where $q \in Q_2$, $\delta(q, a) = \delta_2(q, a)$. Finally, if $q \in Q_1$ and $\delta_1(q, a) = h_a$, $\delta(q, a) = q_1$.

9.11. Let T_1 be a new TM, which differs from T as follows. It has a new state q not present in T, and for every symbol a, $\delta(q, a) = (q, a, \mathrm{S})$. (The effect is that once T_1 gets to q, it loops forever.) For every combination of state p and tape symbol σ (including Δ) for which T moves to h_r, $\delta(p, \sigma) = (q, \sigma, \mathrm{S})$.

The other way T might end up in h_r is to try to move its tape head off the left end of the tape. T_1 avoids this by executing a preliminary sequence of moves, during which it inserts a special tape symbol \$ in square 0, moving the input string and the preceding blank over by one square. After these preliminary moves, T_1 begins in the configuration $(q_0, \$\underline{\Delta}x)$, where x is the input string. If T_1 ever reads the symbol \$, it enters an infinite loop in which the tape head stays on square 0.

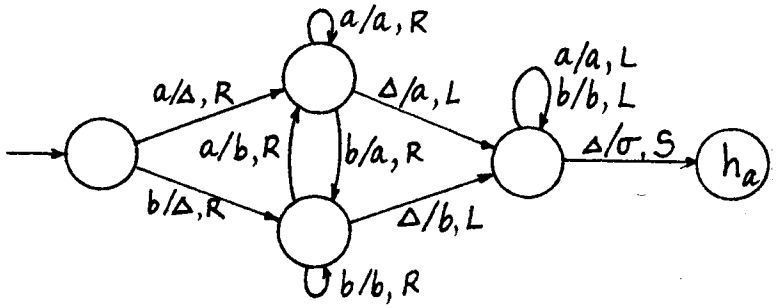
These modifications do not interfere with T's processing of any input string that it accepts, and they do not cause any additional strings to be accepted. Therefore, T_1 accepts the same language as T and never enters the state h_r.

9.12. (a) Suppose there is such a TM T_0, and consider a TM T_1 that halts in the configuration $(h_a, \underline{\Delta}1)$. Let n be the number of the highest-numbered tape square read by

T_0 in the composite machine T_1T_0. Now let T_2 be another TM, which halts in the configuration $(h_a, \underline{\Delta}1\Delta^n1)$. Since the first n tape squares are identical in the final configurations of T_1 and T_2, T_0 will not read the rightmost 1 left by T_2, and so T_2T_0 will not halt with the tape head positioned on the rightmost nonblank symbol left by T_2.

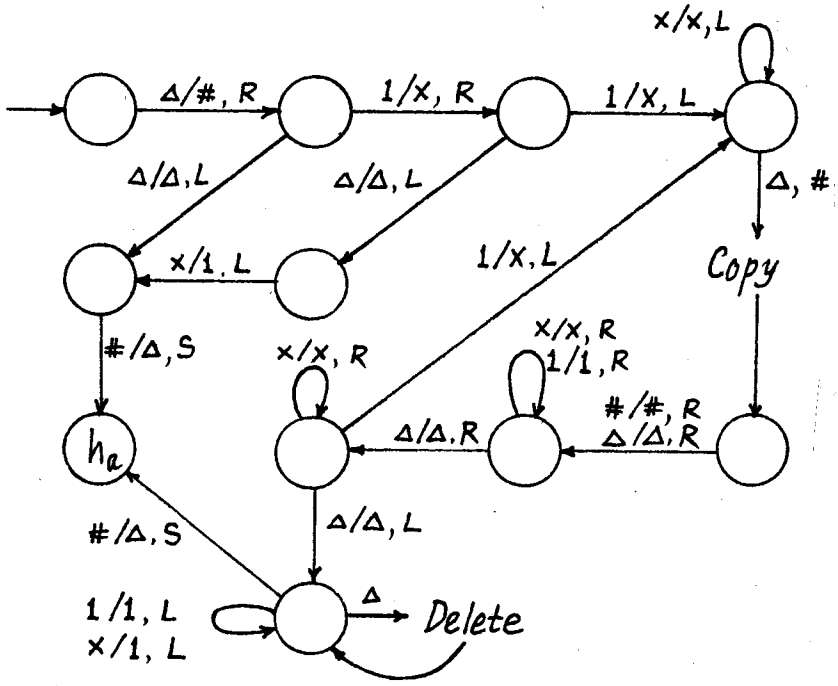
(b) R_T will simulate T, but in such a way that at the end of the processing, the rightmost nonblank symbol can be located. One way to do this is to start by placing a special marker symbol # at the right end of the input string and to return to square 0. During the processing, the marker # will be treated exactly like a blank, except that whenever the tape head reaches the square containing #, this symbol will be moved over one more square to the right. In this way, it continues to mark the rightmost edge of the portion of the tape that has been examined. Once the simulation of T is done, the marker can be erased and the tape head left at the rightmost nonblank symbol preceding it.

9.13.

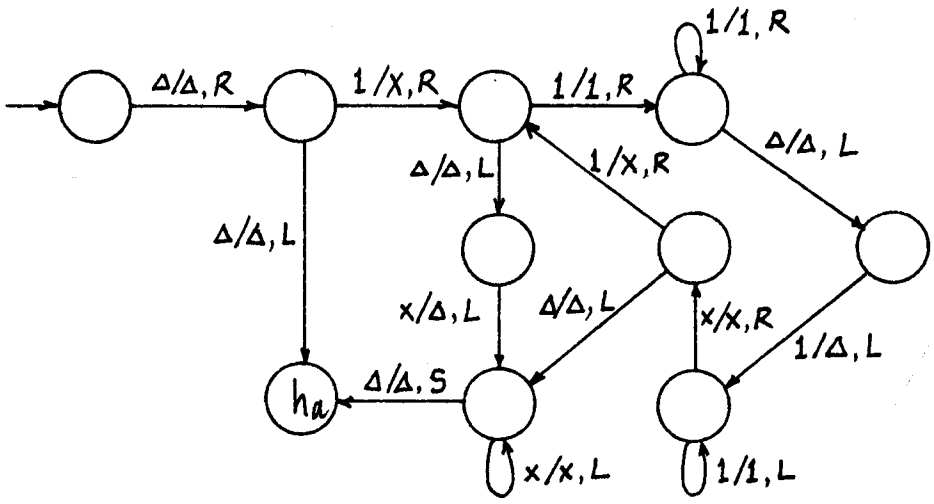


9.14. According to our definition, a TM that accepts an input string but does not leave the tape in the prescribed configuration does not compute a partial function.

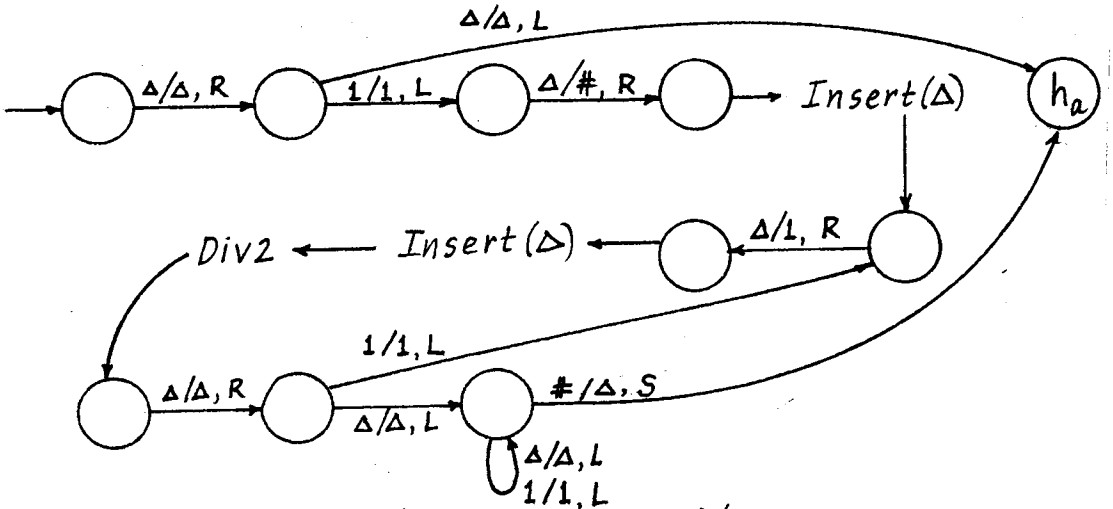
9.15. (c)



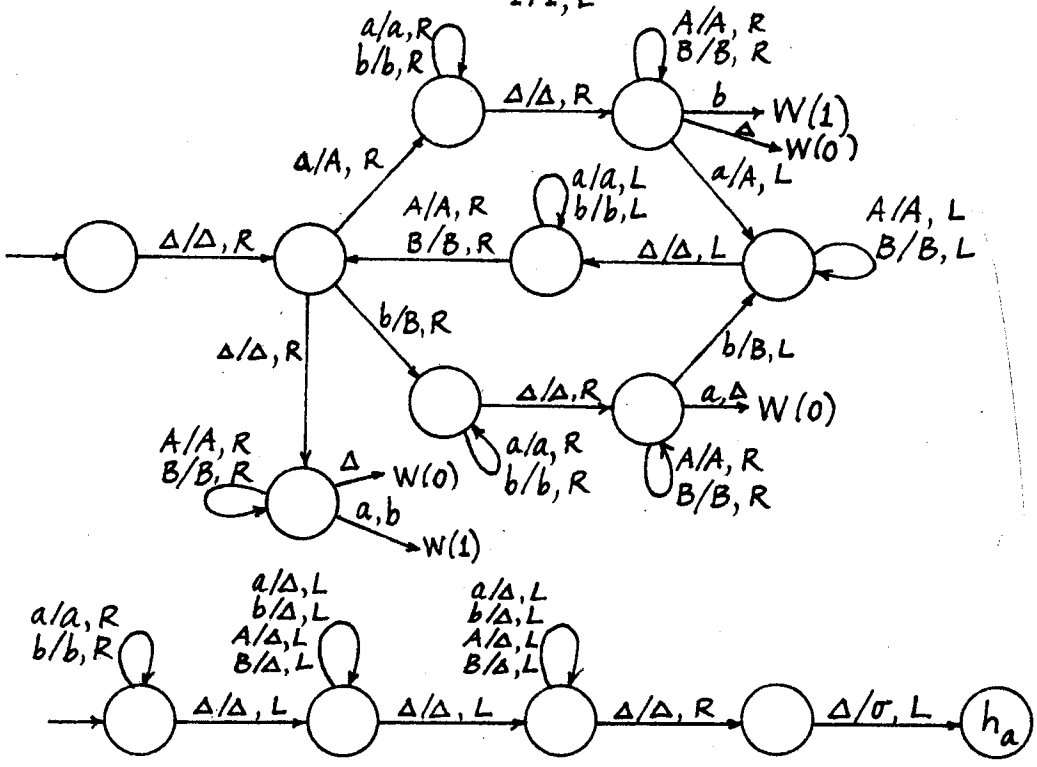
9.15 (d)



(e)



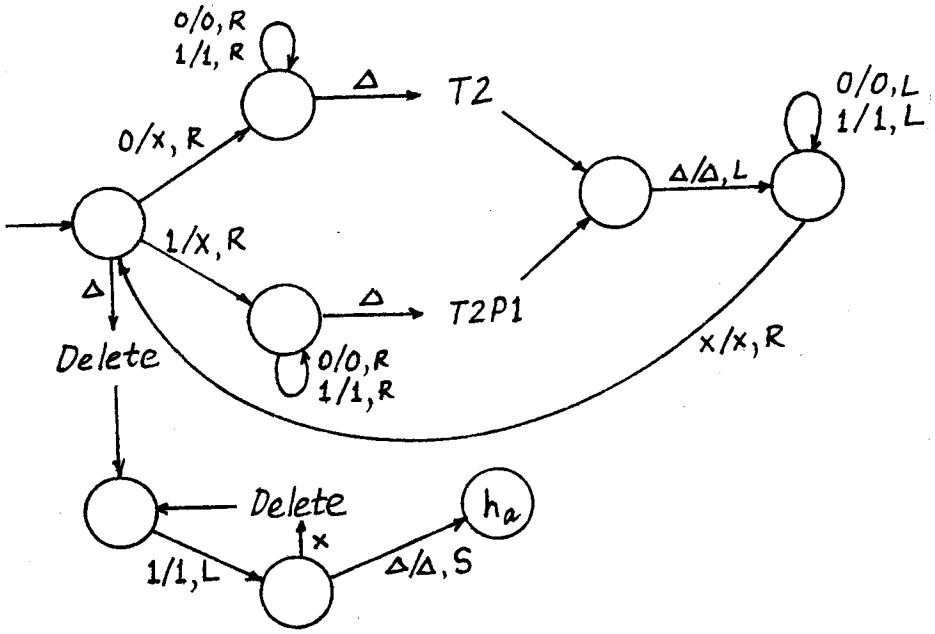
(g)



9.16. See the solution to Exercise 9.15(i).

9.17. In both (a) and (b), a composite TM can start by making a copy of the input, executing T_1 on the copy, interchanging the result with the original input string (which amounts to moving the blank that separates the two the appropriate distance left or right), and executing T_2 on the original input. The result is that the results of T_1 and T_2 are on the tape, separated by a blank. Now it is straightforward to calculate either their sum or their minimum.

9.18. *T2* and *T2P1* compute the functions $n \to 2n$ and $n \to 2n+1$, respectively. The TM below that uses these applies an algorithm that is essentially Horner's rule.



9.19. A pseudocode solution can be expressed as follows. Let `x` denote the input string (or the integer it represents), and `y` the answer.

```
y = 0 (i.e., the null string);
while x is not 0
{  insert the digit x mod 2 at the beginning of y;
   x = x / 2;
}
```

This can be adapted fairly easily to a TM: the string `y` is maintained at the beginning of the tape, and each iteration of the loop consists of a pass through the remaining input to see whether it is even or odd, the insertion of the appropriate symbol at the beginning of `y`, and the execution of the TM in Exercise 9.15(d) on the remaining input.

9.20. One way to modify the diagram is the following. (a) Have the TM insert the symbol # at the beginning of the tape (moving the other tape symbols over) as well as at the end, to simplify the problem of erasing the tape before writing the answer.

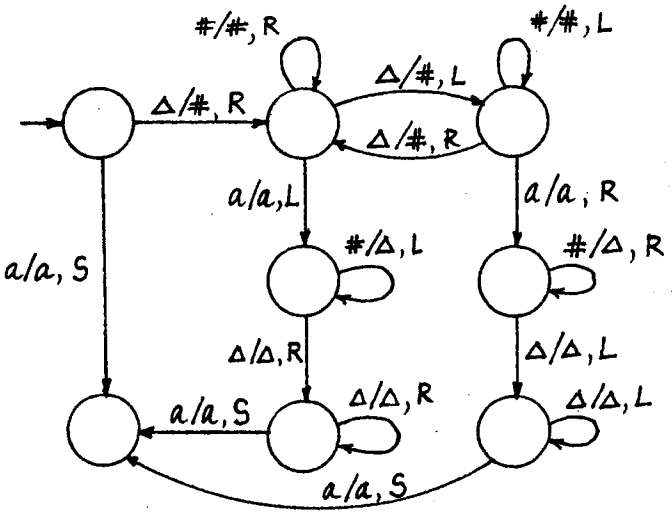
9.20. (b) Replacing the move to the halt state by a sequence of moves that erases the tape, prints 1 on square 1, and halts with the tape head on square 0.

(c) In five places in the diagram where the TM would crash for a string not in the language, add a sequence of moves that would erase the tape, print 0 on square 1, and halt with the tape head on square 0. These five places are: i) the state at the top of the diagram that is third from the right, if the symbol encountered is b or c when it should be a; ii) the state below this one, if the symbol is a or c when it should be b; iii) the state to the left of the one in i), if Δ is encountered instead of c; iv) the state just below this one, in the same situation; v) the state just above the halt state, if anything other than Δ or # is encountered in the loop.

9.21. Here is a verbal description of one solution. Start by inserting an additional blank at the beginning of the tape, to obtain $\Delta\underline{\Delta}x$ (where x is the input). Make a sequence of passes. The first moves the first symbol of x one square to the left and the last symbol one square to the right; the second moves the second symbol of x one square to the left and the next-to-last symbol one square to the right; etc. If x is of odd length, reject. The tape now looks like $\Delta x_1 \Delta\Delta x_2$, where x_1 and x_2 are the first and second halves of x. Now begin comparing symbols of x_1 and x_2, but replacing symbols that have been compared by blanks, rather than uppercase letters as in Example 9.3. In order for this to work, it is necessary to check before each pass to see whether the symbols that are about to be compared are the last symbols in the respective halves. (This prevents an attempt to perform another pass when there are no more nonblank symbols, which would attempt to move the tape head off the tape.)

9.22. The move $\delta(p, a) = (q, b, \mathrm{S})$ can be simulated by using the moves $\delta(p, a) = (p_1, b, R)$ and $\delta(p_1, \sigma) = (q, \sigma, L)$ for any $\sigma \in \Gamma \cup \{\Delta\}$. Here p_1 is a new state used only for this purpose, and the moves shown are the only ones in which it is involved.

9.24.



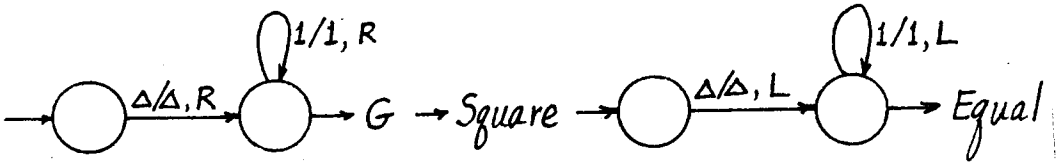
9.25. Given a TM T, a TM T_1 with a doubly infinite tape can be constructed to work as follows: starting in the initial configuration $(q_0, \underline{\Delta}x)$, with the tape head on square 0, it places a special symbol # in square -1, returns the tape head to square 0, and executes

T, except that if it ever reads the symbol # it rejects. It is easy to see that T_1 has the properties specified in the exercise.

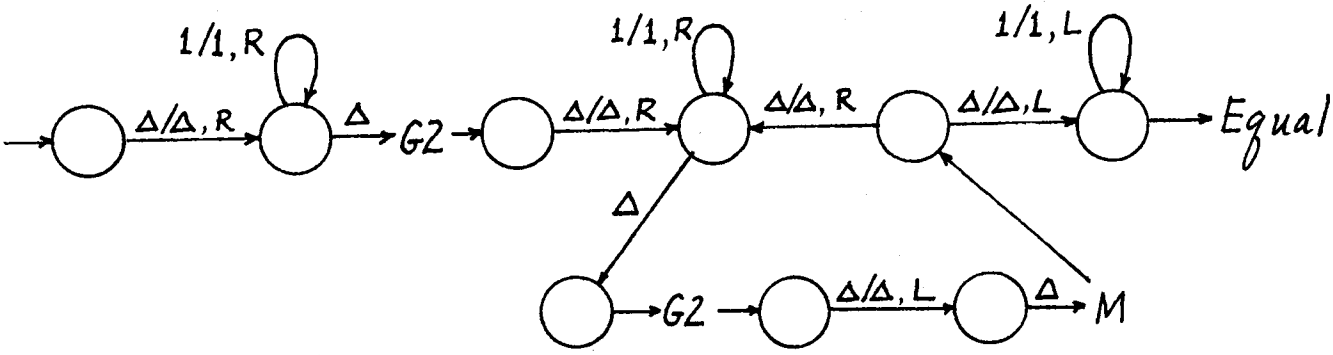
9.29. The TM begins moving its tape head to the right, writing a 0 or a 1 on each square as it goes. It either continues this forever or halts in the configuration $(h_a, \underline{\Delta}x)$, where x is some element of $\{0,1\}^*$. Here x is arbitrary (i.e., for any x, some computation of the TM causes it to halt with x on the tape).

9.30. The language $\{xx \mid x \in \{0,1\}^\}$.

9.31. In the diagram, G is the TM described in Exercise 9.29, *Square* is the TM of Exercise 9.15(c). and *Equal* is as in Exercise 9.30.

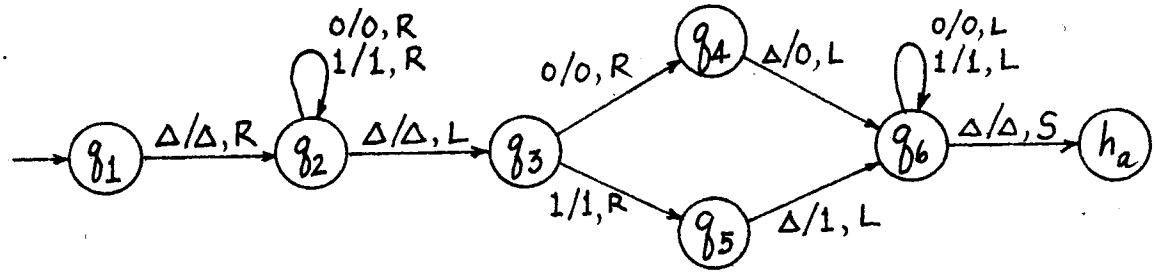


9.32. In the diagram, $G2$ is the same as G in Exercise 9.29 except that it always leaves at least two nonblank symbols on the tape. The TM M computes the multiplication function: it transforms the tape $\underline{\Delta}1^i\Delta 1^j$ to the tape $\underline{\Delta}1^{i*j}$. The TM shown is nondeterministic in that $G2$ is, but also because the steps shown on the lower level can be executed an arbitrary number of times after the first time.

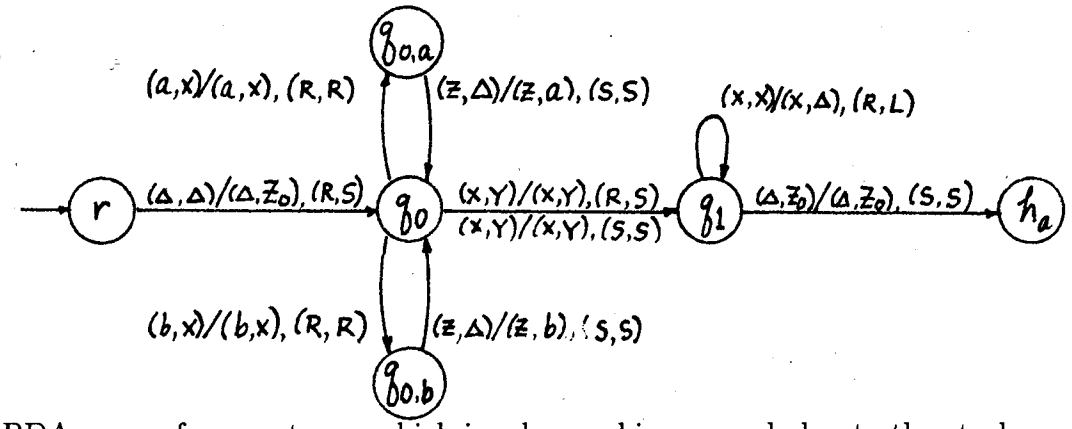


9.33. To accept the set of prefixes of elements of L, for example, a TM could begin by executing the nondeterministic TM G in Exercise 9.29, concatenating the result onto the end of the input string, and executing T on the result. A similar approach works for parts (b) and (c).

9.35.



9.37.



The PDA moves from q_0 to q_0, which involve pushing a symbol onto the stack, require two TM moves, since the TM needs to change a symbol other than the current one on the tape representing the stack. The TM begins by moving to the state corresponding to the initial PDA state q_0, moving the tape head on tape 1 one square to the right, and replacing Δ by Z_0 in the first square of tape 2. The labels on the transitions from q_0 to $q_{0,a}$ and $q_{0,b}$ actually each represent three labels: in both cases, the symbol x can be either a, b, or Z_0 (but is the same in both occurrences within the label). Similarly, on the labels from these two states back to q_0, z can be either a or b but is the same in both occurrences within the label. Each label on the transition from q_0 to q_1 represents six transitions: x can be either a or b, and y can be either a, b, or Z_0 (but all occurrences of x within a label represent the same symbol, and the same for y). Finally, the label on the transition from q_1 to q_1 represents two labels: x can be either a or b.

For example, the moves by which the input string aba is accepted are shown below.

$$(r, \underline{\Delta}aba, \underline{\Delta}) \vdash (q_0, \Delta\underline{a}ba, \underline{Z_0}) \vdash (q_{0,a}, \Delta a\underline{b}a, Z_0\underline{\Delta}) \vdash (q_0, \Delta a\underline{b}a, Z_0\underline{a})$$
$$(q_1, \Delta ab\underline{a}, Z_0\underline{a}) \vdash (q_1, \Delta aba\underline{\Delta}, \underline{Z_0}) \vdash (h_a, \Delta aba\underline{\Delta}, \underline{Z_0})$$

The moves by which aa is accepted are as follows.

$$(r, \underline{\Delta}aa, \underline{\Delta}) \vdash (q_0, \Delta\underline{a}a, \underline{Z_0}) \vdash (q_{0,a}, \Delta a\underline{a}, Z_0\underline{\Delta}) \vdash (q_0, \Delta a\underline{a}, Z_0\underline{a})$$
$$(q_1, \Delta a\underline{a}, Z_0\underline{a}) \vdash (q_1, \Delta aa\underline{\Delta}, \underline{Z_0}) \vdash (h_a, \Delta aa\underline{\Delta}, \underline{Z_0})$$

9.38. The iterative step can be described as follows. If x is the last palindrome written on the tape, then x is copied, so that the last part of the nonblank portion of the tape looks like $\Delta x \Delta x$. If x consists entirely of 1's, then the second copy is then changed to the string $0^{|x|+1}$. Otherwise, the last 0 in the first half of the copy is located. (We take the "first half" to include the middle symbol if $|x|$ is odd.) It is changed to 1, and all the 1's after it within the first half are changed to 0. The second half is then modified so as to make the string a palindrome.

9.39. (a) If the tape head does not move past square n, the possible number of nonhalting configurations is $s(t+1)^{n+1}(n+1)$, where s and t are the sizes of Q and Γ, respectively (see the solution to Exercise 9.3). Within that many moves, therefore, the TM will have

either halted or repeated a nonhalting configuration, and so if it has not halted you may conclude it is in an infinite loop.

(b) As in part (a), let s be the size of the set Q. Suppose you watch the TM until it moves to the first square after the input string, and continue to watch for s more moves. If the tape head has moved right each time, then either it has encountered the combination (q, Δ) twice, for some state q, or the tape head moves right in every possible combination (q, Δ). In either case, the TM is in an infinite loop.

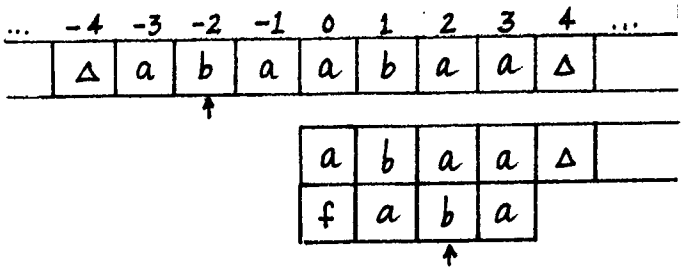
9.40. (a) In this case, the result for an input string x depends only on the first $n+1$ symbols of x. This means that if $S_1 = \{x \in L(T) \mid |x| < n+1\}$ and $S_2 = \{x \in L(T) \mid |x| = n+1\}$, then $L(T) = S_1 \cup S_2\Sigma^*$, which is a regular language.

(b) Once the tape head has passed the last symbol of the input, the TM will never examine the input string again. This means that whether it ever reaches the accepting state depends only on the state it is in when it reaches the first blank square after the input; i.e., there is a set A of states (possibly including h_a) so that for any x, x is accepted if and only if the TM is in a state in A when it reaches the first blank square after the input. Therefore, we may consider the finite automaton $M = (Q, \Sigma, q_1, A, \rho)$, where q_1 is the state T is in after the first move, and for any state $q \in Q$ and $a \in \Sigma$, $\rho(q, a)$ is the state to which T moves on state-symbol combination (q, a). (If $h_a \in A$, then we may add this state to Q and let $\rho(h_a, \sigma) = h_a$ for every $\sigma \in \Sigma$.) The strings accepted by T are precisely those accepted by M, and thus $L(T)$ is accepted by an FA.

9.41. Yes. It follows from the assumption on T that there is a fixed integer n so that for every i, once the tape head reaches square i, it cannot move farther to the left than square $i-n$. This means that we can build another TM T_1 that mimics T but never moves its tape head to the left, by allowing T_1 always to remember the contents of the n squares preceding the current one. Furthermore, it is unnecessary for T_1 to make a move that leaves the tape head stationary, because by examining the transition table for T we can predict the contents of the current tape square and the state to which it goes when it finally moves the tape head right. The conclusion is that there is a TM that accepts the same language as T but always moves its tape head to the right. Exercise 9.40(b) then implies that $L(T)$ is regular.

9.42. We use the first approach. At any point where T has moved its head k squares and no further beyond square 0, either to the left or to the right, T_1's tape will show the first k negative squares as "folded over," in the sense that the portion of the tape from 1 to k will contain two tracks, the first representing the squares from 1 to k and the second those from -1 to $-k$. Square 0 will also contain an extra symbol representing the fold, so that any time T's head crosses from one side of the tape to the other, T_1's head can change from one track to the other.

The following diagram shows what we imagine the two tapes to look like at some point. If T is currently scanning square -2, T_1 will be scanning square 2, paying attention to the lower part only.



The alphabet of T_1 contains not only symbols of Γ but pairs of symbols of Γ, as well as pairs of the form (σ, f), where $\sigma \in \Gamma$ and f represents the fold. (In fact, the alphabet will be even a little larger, as we will see.) The states will be of the form (q, r) and (q, l), where $q \in Q$. The second entries indicate whether the current square of T's tape is on the right side or the left.

T_1 begins by placing the fold marker f in square 0 and moving to state (q_0, r), where it begins to simulate the moves of T. During this simulation, whenever T would move the tape head, T_1 does also, except that if it is currently in a state (q, l) it moves in the opposite direction, and whenever it encounters f it may change tracks. Each time T_1 moves farther from square 0 than it has gone up to that point, it sees a single symbol rather than a pair; at this point it extends the folded portion by converting the single symbol to a pair before continuing.

T_1 starts with the move $\delta(q_1, \Delta) = ((q_0, \mathrm{r}), (\Delta, \mathrm{f}), \mathrm{S})$. For each move $\delta(p, X) = (q, Y, D)$ made by T, we must allow the following moves of T_1. To simplify the notation, let $-D$ denote the direction opposite to D. First, for every $Z \in \Gamma \cup \{\Delta\}$,

$$\begin{aligned} \delta_1((p, \mathrm{r}), (X, Z)) &= ((q, \mathrm{r}), (Y, Z), D) \\ \delta_1((p, \mathrm{l}), (Z, X)) &= ((q, \mathrm{l}), (Z, Y), -D) \end{aligned}$$

Second, $\delta_1((p, \mathrm{r}), (X, \mathrm{f}))$ and $delta_1((p, \mathrm{l}), (X, \mathrm{f})$ are both $((q, \mathrm{l}), (Y, \mathrm{f}), \mathrm{R})$ if $D = \mathrm{L}$ and $((q, \mathrm{r}), (Y, \mathrm{f}), D)$ otherwise. In addition, for each move $\delta(p, X) = (q, Y, D)$, T_1 has the moves $\delta_1((p, \mathrm{r}), X) = ((q, \mathrm{r}), (Y, \Delta), D)$ and (if $X = \Delta$) $\delta_1((p, \mathrm{l}), \Delta) = ((q, \mathrm{l}, (\Delta, Y), -D)$. These moves allow T_1 to simulate faithfully the moves of T up to the point where it halts.

If and when T accepts, T_1 moves instead to some state that still allows it to remember which track of the tape it is supposed to be looking at. Before it can halt, it must fix up its tape so that it is really a duplicate of T's, with the obvious exception that the nonblank symbols will begin in square 0 or somewhere to the right. This essentially means "unfolding" the tape. We can do this by adding extra alphabet symbols of the form (σ, γ'), (σ', γ), and (σ', f), where $\sigma, \gamma \in \Gamma \cup \{\Delta\}$. The primes indicate the current head position and simplify the problem of finding this position later. The unfolding can be done in a sequence of passes, where each pass deletes one symbol from the second track of the tape and inserts it at the beginning of the first. When the second track has been eliminated, the f is deleted, the final position of the tape head is located, and the symbol there is changed back to its original form.

9.44. If we assume that M_1 reads the entire input string, then we can assume that there will be no nonblank symbols farther to the right than the nth square. This means that the

preliminary insertion and final deletion of the symbol # can be done in a total of approximately $4n$ moves, the final pass converting symbol pairs to single symbols can be done in approximately $2n$ moves, and moving to the final tape position requires no more than n moves. The total so far is therefore about $7n$. To simulate a single move in the way that's described in the proof requires three iterations of the following process: move the tape head to the current location on one of the tracks, make some fixed number of moves, and move the tape head back. Since by the ith move, the current location on either track can't be farther right than square i, these three iterations require at most $6i$ moves, plus some fixed number independent of i. The total of these is therefore bounded by

$$\sum_{i=1}^{n}(6i + C) = 3n(n + 1) + Cn$$

The overall total is therefore a quadratic function of n: $3n^2 + 7n + Cn$, where C is a fairly small constant.

9.46. See the proof of Lemma 11.1.

9.47. See the proof of Theorem 10.2.

9.48. For each string of digits of length k that specifies a sequence of moves of the nondeterministic TM to be tried, the number of moves made by T_2 is roughly proportional to $n+k$: n because of copying the input string, k because of executing the sequence of moves, erasing the tape, and calculating the next sequence. For each k, there are 2^k sequences of k moves. So the total number of moves of T_2 is, very approximately, $\sum_{k=1}^{n_x}(n + k)2^k$. This is roughly proportional to $(n + n_x)2^{n_x+1}$.

9.49. We give an informal description of one such machine. For each initial a that it reads, it pushes an x onto the first stack. When it encounters a b, it processes each b by removing an x from stack 1 and placing it on stack 2. When it encounters a c, it checks that stack 1 is empty and transfers the contents of stack 2 to stack 1. It then processes c's the same way it processed b's. This approach can obviously be generalized to any number of distinct symbols.

9.51. The machine makes a sequence of passes. On the first pass, it moves symbols from the front to the rear, except that the first a, b, and c are replaced by A, B, and C, respectively. It crashes if it detects an illegal pair of symbols (ba, cb, ac, or ca), or if it finds one but not all three of the lowercase symbols. On each subsequent pass, it processes the first remaining a, b, and c similarly, crashing if it finds one but not all of the three. It accepts if on some pass there are no remaining a's, b's, or c's.

9.52. The idea that allows a Post machine to simulate a move to the left on the tape is to remember the last two symbols that have been removed from the front of the queue. We illustrate by an example. Suppose that the initial contents of the queue are $abaab$, with the

a at the front. The machine starts by placing a marker X at the rear, to produce $abaabX$. It proceeds to remove symbols from the front, remembering the last two, and placing the symbols at the rear but lagging behind by one. Thus after two steps, the contents are $aabXa$, and the machine remembers that the last symbol removed was b. After three more steps, the contents are $Xabaa$, and the machine remembers that the last symbol was b. When it reads X, it places X on the rear immediately, followed by b, to produce $abaaXb$. At this point, it begins simply moving symbols from the front to the rear, and continues until it has removed the X. If it fails to replace it, the contents of the queue are now $babaa$, and the move to the left has been effectively simulated.

Chapter 10

Recursively Enumerable Languages

10.2. (a) This approach would not work for recognizing $L_1 \cup L_2$. The reason is that if the input string is in L_2 but not in L_1, the TM might enter an infinite loop while simulating T_1 and thus never begin the simulation of T_2.

(b) This approach could be used to recognize $L_1 \cap L_2$. If the input string is in the language, the TM will accept it, since the simulations of T_1 and T_2 will both terminate successfully. In all other cases, one of the simulations will reject or enter an infinite loop, and in any such case the composite TM will fail to accept.

10.3. False. There is a language L that is not recursively enumerable, and if $L = \{x_1, x_2, \ldots\}$, then $L = \{x_1\} \cup \{x_2\} \cup \ldots$.

10.4. For each i, L_i' is the union of the L_j's with $j \neq i$. Therefore, L_i' is recursively enumerable. Since L_i is also, L_i is recursive by Theorem 10.5.

10.5. We sketch the proof that if there is a TM T enumerating L in canonical order, then L is recursive. First, in the case where L is finite, L is obviously recursive. Otherwise, given a string x, a TM T_1 can determine whether x is in L by executing T until one of the strings left on the tape of T is either x or a string that comes after x in canonical order. In the first case, $x \in L$, and in the second case $x \notin L$.

10.6. One direction here is simple: if there is a computable partial function f that is defined precisely at the points of L, then by definition of computable, a TM computing f accepts precisely the strings in L.

Suppose that T accepts L. We may simply modify T by making the TM erase the tape and leave the tape head on square 0 before accepting. Then the modified TM computes the partial function f, defined to be Λ at every point in L and undefined otherwise.

10.7. Suppose there is no longest path. Then the root node N_0 must have infinitely many descendants. Since N_0 has only a finite number of children, then one of them, say N_1, must have infinitely many descendants. Similarly, one of N_1's children, say N_2, must have infinitely many descendants. This reasoning can be repeated indefinitely, and it follows that the tree has the infinite path N_0, N_1, ..., contrary to the assumption.

10.8. One approach to both (a) and (b), which uses the fact that these languages are both recursive, is to choose a method of enumerating $\{0,1\}^*$ and for each string in the enumeration, list it if it is in the language we want. However, we give alternate algorithms, which are presumably at least a little more efficient.

(a) The set is

$$101, 101^2, 1^201, 101^3, 1^201^3, 1^301, 101^4, 1^201^5, 1^301^2, 1^401, \ldots$$

The "algorithm" (though of course it never terminates) is the following:

For $i = 1$ to ∞
  For $j = 1$ to i
    List the string 1^j01^k, where k is the smallest integer relatively prime to j for which 1^j01^k has not yet been listed

(b) For each nonnull string w, let $x_{1,w}$, $x_{2,w}$, ... be an enumeration, possibly including repetitions, of all strings containing the substring www. (For example, we could for each i list every string of length i $i+1$ times, and insert www in each of the $i+1$ positions.) Now let w_1, w_2, ... be an enumeration of the strings w in canonical order. Then use the algorithm

For $i = 1$ to ∞
  For $j = 1$ to i
    List the string $x_{j,w_{i+1-j}}$ *if* it has not previously been listed.

(c) Fermat's last theorem, of which a proof was published by Andrew Wiles in 1995, says that there are no such n's bigger than 2. If we pretend that we don't know this, we can select some enumeration of the 4-tuples (n, x, y, z), and for each such 4-tuple, list n if it is not already listed and $x^n + y^n = z^n$. If we do know the theorem, then we write the numbers 1 and 2, and we're done.

10.10. Given a number y, we can determine whether y is in the range of f by computing $f(0)$, $f(1)$, ..., until we find a j for which $f(j) \geq y$. If $f(j) > y$ for the first such j, y is not in the range.

10.11. (a) The variable D can be thought of as a "doubling" operator. The language is $\{a^n \mid n = 2^k \text{ for some } k \geq 0\}$.

(b) $\{a^n \mid n = 2^j3^k \text{ for some } j, k \geq 0\}$

(c) $\{x \in \{a, b, c\}^* \mid n_a(x) = n_b(x) = n_c(x)\}$

(d) $\{a^n \mid n = k^2 \text{ for some } k \geq 1\}$. The way to understand this is to consider the string $\alpha_k = LA^k * A^k * \ldots * A^k * R$, in which there are k groups of k A's, each group followed by $*$. I can be viewed as an "incrementing" variable, which operates on the string to produce α_{k+1}. After using the productions $S \to LA * R$, we have α_1. When an additional I is created in α_k, the two productions $I* \to A * IJ$ and $IR \to A * R$ allow I to add an extra A to each of the k groups and add an extra group (i.e., an extra $*$) at the end with one A in it. In addition, the I creates a J in each of the original groups, and once each J has worked its way to the end of the string, it leaves an extra A in the final group. The result is therefore $k+1$ groups of $k+1$ A's. The variable E is an "erasing" variable that destroys all the symbols except the A's, which can be replaced by a's.

10.12. (a) $\{a^nb^na^n \mid n \geq 0\}$

(b) $B \to b$

10.13. (a)

$$S \to FS_1 \mid \Lambda \qquad S_1 \to ABCDS_1 \mid ABCD$$
$$BA \to AB \qquad CA \to AC \qquad DA \to AD \qquad CB \to BC \qquad DB \to BD \qquad DC \to CD$$
$$FA \to A_1 \qquad A_1A \to A_1A_1 \qquad A_1B \to A_1B_1 \qquad B_1B \to B_1B_1$$
$$B_1C \to B_1C_1 \qquad C_1C \to C_1C_1 \qquad C_1D \to C_1D_1 \qquad D_1D \to D_1D_1$$
$$A_1 \to a \qquad B_1 \to b \qquad C_1 \to a \qquad D_1 \to b$$

The idea here is that the symbols A, B, C, and D rearrange themselves into the form $A^nB^nC^nD^n$. The auxiliary variables A_1, B_1, C_1, and D_1 are introduced in order to force this to happen: The original variables can be converted into the new ones only when they are in the right order, and the terminals can be produced only by the new variables. (If the original variables went directly to terminal symbols, in the same way as in Example 10.1, there would be the possibility of errors, because both a's and b's are allowed to appear in two different parts of the string.)

(b)

$$S \to FS_1 \mid \Lambda \qquad S_1 \to ABCS_1 \mid ABC$$
$$BA \to AB \qquad CA \to AC \qquad CB \to BC$$
$$FA \to A_1 \qquad A_1A \to A_1A_1 \qquad A_1B \to A_1B_1$$
$$B_1B \to B_1B_1 \qquad B_1C \to B_1C_1 \qquad C_1C \to C_1C_1$$
$$A_1 \to a \qquad B_1 \to a \mid b \qquad C_1 \to b$$

The principle in this grammar is the same as in part (a), except that this time the symbol B_1 can be replaced by either a or b.

(c)

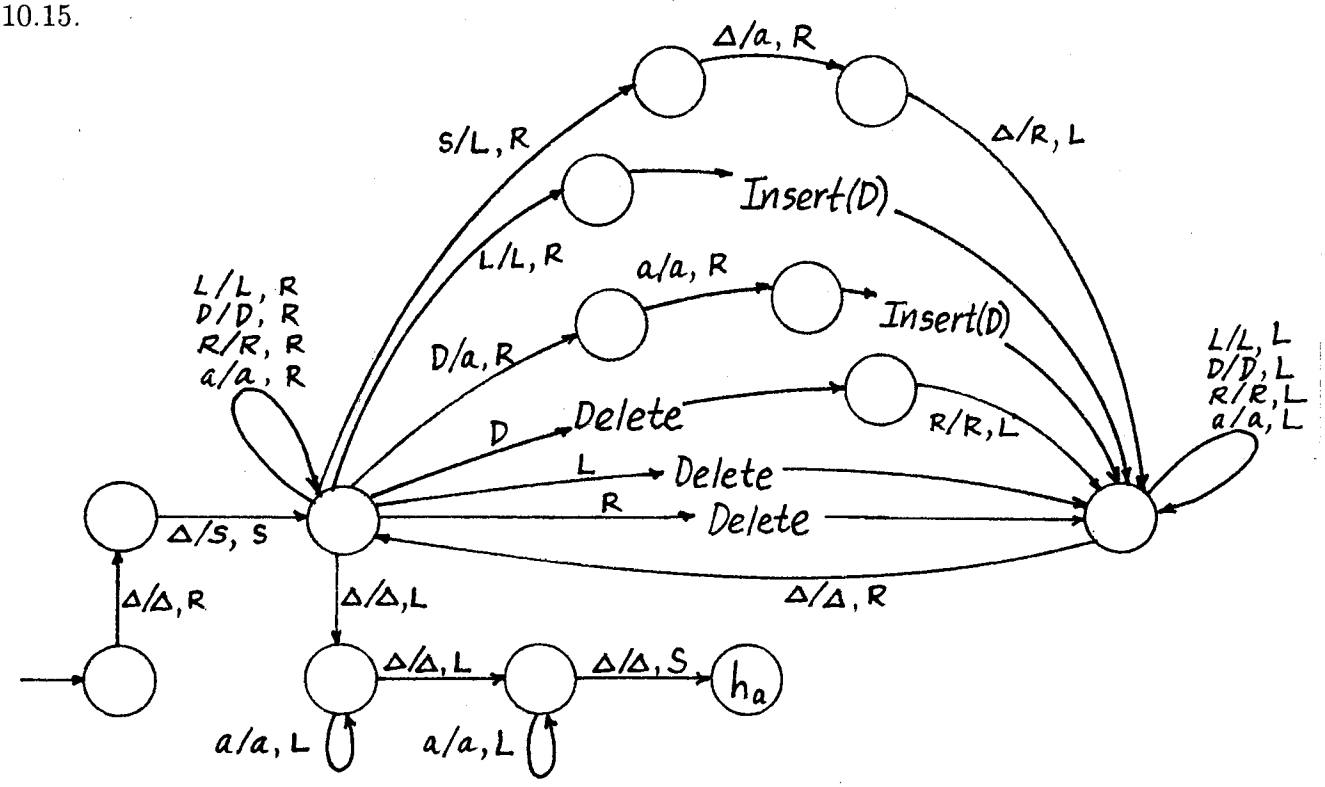
$$S \to LMN \qquad L \to LaA \mid LbB \mid \Lambda \qquad M \to \Lambda \qquad N \to \Lambda$$
$$Aa \to aA \qquad Ab \to bA \qquad Ba \to aB \qquad Bb \to bB$$
$$AM \to MaA \qquad BM \to MbB \qquad AN \to Na \qquad BN \to Nb$$

(d)

$$S \to LMN \qquad L \to LaA \mid LbB \mid \Lambda \qquad M \to \Lambda \qquad N \to \Lambda$$
$$Aa \to aA \qquad Ab \to bA \qquad Ba \to aB \qquad Bb \to bB$$
$$AM \to MA \qquad BM \to MB \qquad AN \to aNa \qquad BN \to bNb$$

Both parts (c) and (d) work on the principle of a variable (either A or B) migrating across the string and depositing the corresponding terminal in each of the three portions of the string. The difference is that in (c) the terminal is deposited at the beginning of each portion and in (d) it is deposited at the beginning of the first portion, the end of the second, and the beginning of the third.

10.15.



10.16.

$$
\begin{aligned}
S &\Rightarrow S(\Delta\Delta) \Rightarrow T(\Delta\Delta) \Rightarrow T(aa)(\Delta\Delta) \\
&\Rightarrow T(bb)(aa)(\Delta\Delta) \Rightarrow T(bb)(bb)(aa)(\Delta\Delta) \\
&\Rightarrow T(aa)(bb)(bb)(aa)(\Delta\Delta) \\
&\Rightarrow q_0(\Delta\Delta)(aa)(bb)(bb)(aa)(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)q_1(aa)(bb)(bb)(aa)(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)(a\Delta)q_2(bb)(bb)(aa)(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)(a\Delta)(bb)q_2(bb)(aa)(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)(a\Delta)(bb)(bb)q_2(aa)(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)(a\Delta)(bb)(bb)(aa)q_2(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)(a\Delta)(bb)(bb)q_3(aa)(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)(a\Delta)(bb)q_4(bb)(a\Delta)(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)(a\Delta)q_4(bb)(bb)(a\Delta)(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)q_4(a\Delta)(bb)(bb)(a\Delta)(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)(a\Delta)q_1(bb)(bb)(a\Delta)(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)(a\Delta)(b\Delta)q_5(bb)(a\Delta)(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)(a\Delta)(b\Delta)(bb)q_5(a\Delta)(\Delta\Delta) \\
&\Rightarrow (\Delta\Delta)(a\Delta)(b\Delta)q_6(bb)(a\Delta)(\Delta\Delta)
\end{aligned}
$$

$$
\begin{aligned}
&\Rightarrow (\Delta\Delta)(a\Delta)q_4(b\Delta)(b\Delta)(a\Delta)(\Delta\Delta)\\
&\Rightarrow (\Delta\Delta)(a\Delta)(b\Delta)q_1(b\Delta)(a\Delta)(\Delta\Delta)\\
&\Rightarrow (\Delta\Delta)(a\Delta)h(b\Delta)(b\Delta)(a\Delta)(\Delta\Delta)\\
&\Rightarrow (\Delta\Delta)h(a\Delta)h(b\Delta)(b\Delta)(a\Delta)(\Delta\Delta)\\
&\Rightarrow h(\Delta\Delta)h(a\Delta)h(b\Delta)(b\Delta)(a\Delta)(\Delta\Delta)\\
&\Rightarrow h(\Delta\Delta)h(a\Delta)h(b\Delta)h(b\Delta)(a\Delta)(\Delta\Delta)\\
&\Rightarrow h(\Delta\Delta)h(a\Delta)h(b\Delta)h(b\Delta)h(a\Delta)(\Delta\Delta)\\
&\Rightarrow h(\Delta\Delta)h(a\Delta)h(b\Delta)h(b\Delta)h(a\Delta)h(\Delta\Delta)\\
&\Rightarrow \Lambda h(a\Delta)h(b\Delta)h(b\Delta)h(a\Delta)h(\Delta\Delta)\\
&\Rightarrow \Lambda a h(b\Delta)h(b\Delta)h(a\Delta)h(\Delta\Delta)\\
&\Rightarrow \ldots \Rightarrow abba
\end{aligned}
$$

10.17. In the grammar in Example 10.2, F will be replaced by either A_1 or B_1, representing the first symbol (either a or b) in the first half. Similarly, M will be replaced by A_2 or B_2.

$$
\begin{gathered}
S \to A_1A_2 \mid B_1B_2\\
A_1 \to A_1aA \mid B_1aB \qquad B_1 \to A_1bA \mid B_1bB\\
Aa \to aA \qquad Ab \to bA \qquad Ba \to aB \qquad Bb \to bB\\
AA_2 \to A_2a \qquad AB_2 \to A_2b \qquad BA_2 \to B_2a \qquad BB_2 \to B_2b\\
A_1 \to a \qquad A_2 \to a \qquad B_1 \to b \qquad B_2 \to b
\end{gathered}
$$

10.18. (a) $S \to a \mid A_1A_2 \qquad A_1 \to A_1A \mid a \qquad A_2 \to a \qquad Aa \to aaA \qquad AA_2 \to aaA_2$
(d)

$$
\begin{gathered}
S \to a \mid A_LA_2aA_R \qquad A_L \to A_LI \mid E\\
Ia \to aI \mid IA_2 \to aA_2JI \qquad IA_R \to aA_2aA_R\\
Ja \to aJ \qquad JA_2 \to A_2J \qquad JA_R \to aA_R\\
Ea \to aE \qquad EA_2 \to aE \qquad EA_R \to aa
\end{gathered}
$$

The idea here is that at some stage, when the current string consists of n groups of n a's, each group be represented as $aa\ldots aA_2$, so that the variable A_2 represents the last a in the group. However, the first such group needs to begin with a special variable A_L to represent the first a in the entire string, and the last such group needs to end with a special variable A_R (instead of A_2) to represent the last a in the string. The case of only one a is taken care of separately. The variables I and J are used more or less as before, except that now each of them actually represents an a, not just a location in the string of a's. The production that is the most different, and perhaps requires a little more explanation, is $IA_R \to aA_2aA_R$. In the grammar in Exercise 10.11(d), the corresponding production was $IR \to A * R$; the

interpretation was that after I had already passed the last $*$, it formed a new group by creating a $*$ and leaving A before it. Here, A_R is within the last group; thus I must first add an a to this group (and at the same time change the A_R to A_2, since this group will no longer be the last one), then create the last group by creating a new A_R.

10.20. If G has the productions $S_1 \to BaB$, $aB \to Ba$, and $B \to b$, then $L(G) = \{bab, bba\}$. Either the grammar used in Chapter 6 to generate $L(G)L(G)$ or the one used to generate $L(G)^*$ would allow S_1S_1 to be generated, and would therefore allow the derivation $S_1S_1 \Rightarrow^* BaBBaB \Rightarrow^* BBBBaa \Rightarrow^* bbbbaa$. However, this string is in neither $L(G)L(G)$ not $L(G)^*$.

10.21. The basic idea is that if T can be restricted to the portion of the tape between square 0 and square $k|x|$, then by using several tracks on the tape it is possible to use another TM that is restricted to the portion between 0 and $|x|$. Therefore, L can be accepted by an LBA.

10.22. A nondeterministic TM can simulate derivations of G as in the proof of Theorem 10.8 but with the following modifications. It can mark a portion of the tape, whose length is $k|x|$, where x is the input string; this portion of the tape will be the one used for simulating the derivation. In addition, farther to the right on the tape, it can keep a record of every string that appears in a derivation. It rejects whenever either the string produced in the simulation is either longer than $k|x|$ or a string that has appeared at some previous step. Eventually, if it does not accept, it must reject in one of these two ways. Therefore, by Theorem 10.2, the language it accepts is recursive; however, it is easy to see that it accepts $L(G)$, since any string in $L(G)$ can be generated by a derivation in which no string is obtained for the second time.

10.23. The productions corresponding to $\delta(p, a) = (q, b, \mathrm{L})$ are

$$\begin{array}{rr} (\sigma_1\sigma_2)p(\sigma_3 a) \to q(\sigma_1\sigma_2)(\sigma_3 b) & (\sigma_1\langle\sigma_2)p(\sigma_3 a) \to q(\sigma_1\langle\sigma_2)(\sigma_3 b) \\ (\sigma_1\sigma_2)p(\sigma_3 a\rangle) \to q(\sigma_1\sigma_2)(\sigma_3 b\rangle) & (\sigma_1\langle\sigma_2)p(\sigma_3 a\rangle) \to q(\sigma_1\langle\sigma_2)(\sigma_3 b\rangle) \\ p(\sigma\langle a) \to q(\sigma\langle b)^L & p(\sigma\langle a\rangle) \to q(\sigma\langle b\rangle)^L \end{array}$$

Those corresponding to $\delta(p, a) = (q, b, S)$ are

$$p(\sigma_1 a) \to q(\sigma_1 b) \qquad p(\sigma_1\langle a) \to q(\sigma_1\langle b) \qquad p(\sigma_1 a\rangle) \to q(\sigma_1 b\rangle) \qquad p(\sigma_1\langle a\rangle) \to q(\sigma_1\langle b\rangle)$$

10.24. Let S be countable and $T \subseteq S$. If T is infinite, then S is certainly infinite also. Since S is countable, we may write $S = \{x_0, x_1, \ldots\}$, where $x_i \neq x_j$ if $i \neq j$. Let i_0 be the first subscript for which $x_{i_0} \in T$; let i_1 be the second such subscript; etc. Then $T = \{x_{i_1}, x_{i_2}, \ldots\}$. It follows that T is countably infinite, since the function that takes j to x_{i_j} is a bijection from $\mathcal{N}$ to T.

10.25. Suppose S is infinite. S must contain an element i_0, since otherwise there would be

a bijection from $\emptyset$ to S and S would be finite. In general, if $i_0, i_1, \ldots, i_k$ have been defined to be distinct elements of S, then there must be an element $i_{k+1} \in S$ that is not i_j for any $j \leq k$—otherwise there would be a bijection from $\{0, 1, \ldots, k\}$ to S. Therefore, S has a countably infinite subset $I = \{i_0, i_1, i_2, \ldots\}$. Now define a function $f : S \to S$ as follows: $f(x) = x$ for every $x \in S - I$, and for every $j \geq 0$, $f(i_j) = i_{j+1}$. Then f is a bijection from S to the proper subset $I \cup \{i_1, i_2, \ldots\}$ of S.

Conversely, if S is finite, then for some n, $S = \{i_0, i_1, \ldots, i_n\}$. It is not difficult to show using induction on n that there cannot be a bijection from S to a proper subset of S.

10.26. Suppose $f : S \to T$ is a bijection. If S is finite, then T is finite and therefore countable. If S is countably infinite, then there is a bijection $i : \mathcal{N} \to S$; the function $f \circ i : \mathcal{N} \to T$ is therefore a bijection, and T is countable. Finally, if T is countable, then since $f^{-1} : T \to S$ is a bijection, S is countable. Therefore, if S is uncountable, so is T.

10.27. The set $S \cap T$ is countable, since it is a subset of T. If $S - T$ were countable, then $(S - T) \cup (S \cap T) = S$ would be also.

10.28. First we can list the set $\mathcal{Q}_0$ of rational numbers x satisfying $0 \leq x < 1$ as follows:

$$0, 1/2, 1/3, 2/3, 1/4, 3/4, 1/5, 2/5, 3/5, 4/5, 1/6, 5/6, 1/7, \ldots$$

(The rule is to list the numbers in nondecreasing order of denominator, leaving out a fraction if it is numerically equal to one that has already appeared.)

Next we construct a list L that contains every integer infinitely often. One such list is

$$0, -1, 0, 1, -2, -1, 0, 1, 2, -3, -2, -1, 0, 1, 2, 3, -4, -3, -2, -1, 0, 1, 2, 3, 4, -5, \ldots$$

Finally, for an integer i, we let $\mathcal{Q}_i = \{i + x \mid x \in \mathcal{Q}_0\} = \{z \in \mathcal{Q} \mid i \leq z < i + 1\}$. Since $\mathcal{Q} = \cup_{i=-\infty}^{\infty} \mathcal{Q}_i$, we can create a list of the elements of $\mathcal{Q}$ by starting with the list L, and for each entry, if it is the jth occurrence of the integer i in L, replacing it by the jth element of $\mathcal{Q}_i$, where we use our listing of $\mathcal{Q}_0$ and the natural bijection from $\mathcal{Q}_0$ to $\mathcal{Q}_i$ determined by adding i.

Although it would be hard to write a formula for the nth element of the list, it is easy enough to find the nth element for a given value of n. Here are the first few:

$$0, -1, 1/2, 1, -2, -1/2, 1/3, 3/2, 2, -3, -3/2, -2/3, 2/3, 4/3, 5/2, 3, -4, \ldots$$

10.29. Recall the convention introduced in Chapter 9, by which there are fixed countably infinite sets $\mathcal{Q}$ and $\mathcal{S}$ from which we choose the states and the tape alphabet, respectively, of any Turing machine. For a given finite set of states, input alphabet, and tape alphabet, the number of Turing machines is finite. Since the set of finite subsets of $\mathcal{S}$ is countably infinite (see Exercise 10.31(b)), and once we fix the tape alphabet there are only a finite number of possible choices for the input alphabet, one application of Theorem 10.13 implies that there are only a countable number of Turing machines having a given set of states.

Since there are only a countably infinite number of finite subsets of $\mathcal{Q}$, another application implies that the set of Turing machines is finite.

10.30. (a) Define $f : S \to 2^{\mathcal{N}}$ as follows: for any $x \in S$, say $x = a_0, a_1, \ldots$, let $f(x)$ be the set $\{n \mid a_n = 1\}$. It is easy to see that f is a bijection.

(b) Suppose $S = \{s_0, s_1, \ldots\}$, and for each $j \geq 0$, let $s_{i,j}$ be the jth term of the sequence s_i. Then define x to be the sequence x_0, x_1, $\ldots$, where for each i, $x_i = 1 - s_{i,i}$. In other words, $x_i = 0$ if $s_{i,i} = 1$ and $x_i = 1$ if $s_{i,i} = 0$. Then for any i, since $x_i \neq s_{i,i}$, it follows that $x \neq s_i$. This shows that there can be no enumeration of the elements of S.

The reason this proof is essentially the same as the proof of Theorem 10.15 is that if we let $A_i = f(s_i)$ and $A = f(x)$, where $f : S \to 2^{\mathcal{N}}$ is the function described in the solution to (a), then

$$A = \{i \mid x_i = 1\} = \{i \mid s_{i,i} \neq 1\} = \{i \mid i \notin A_i\}$$

10.31. (a) Countable. This set is a subset of the set in part (b).

(b) Countable. The set is $\cup_{i=0}^{\infty} T_i$, where T_i is the set of all subsets of $\{0, 1, \ldots, i\}$. Each T_i is finite and therefore countable; therefore, their union is countable.

(c) Consider the following sets.

i) The set of nonempty subsets of $\mathcal{N}$
ii) The set of nonempty subsets of $\mathcal{N} - \{0\}$
ii) The set of partitions of $\mathcal{N}$ with two subsets

The first is uncountable, since $2^{\mathcal{N}}$ is. There is a bijection from the first to the second, since there is a bijection from $\mathcal{N}$ to $\mathcal{N} - \{0\}$, and so the second is uncountable. Finally, there is a bijection f from the second to the third. For any nonempty subset A of $\mathcal{N} - \{0\}$, let $f(A)$ be the partition consisting of A and $\mathcal{N} - A$. Then f is onto, because for any two-set partition of $\mathcal{N}$, one of the two sets is a nonempty set not containing 0. Also, f is 1-1, because if the two sets of a partition are A and $\mathcal{N} - A$, only one of them fails to contain 0, so that the partition cannot be both $f(A)$ and $f(\mathcal{N} - A)$.

It follows that the set of two-set partitions of $\mathcal{N}$ is uncountable. This is a subset of the set of all finite partitions, and thus the larger set is uncountable.

(d) A function from $\mathcal{N}$ to $\{0, 1\}$ is nothing more than a sequence a_0, a_1, $\ldots$ of 0's and 1's. More precisely, there is a natural bijection from the set $\mathcal{F}$ of functions from $\mathcal{N}$ to $\{0, 1\}$ to the set S in Exercise 10.30. It follows from that exercise that $\mathcal{F}$ is uncountable.

(e) There is an obvious bijection f from this set to the set $\mathcal{N} \times \mathcal{N}$: if t is a function from $\{0, 1\}$ to $\mathcal{N}$, take $f(t)$ to be the pair $(t(0), t(1))$. Since $\mathcal{N} \times \mathcal{N}$ is countable, this set is.

(f) This contains the set in (d) as a subset and is therefore uncountable.

(g) Suppose $n \geq 0$, $A = \{a_0, a_1, \ldots, a_n\}$ is a nonempty finite subset of $\mathcal{N}$, and the elements are listed in decreasing order. Then a nonincreasing function f from $\mathcal{N}$ to $\mathcal{N}$ with range A determines a finite set $S_{A,f} = \{j_1, j_2, \ldots, j_n\}$, where $f(i) = a_0$ for $0 \leq i < j_1$, $f(i) = a_1$ for $j_1 \leq i < j_2$, $\ldots$, $f(i) = a_n$ for $j_n \leq i$. (If $n = 0$, the set $S_{A,f}$ is empty; otherwise, $j_1 > 0$.) Two different nonincreasing functions with range A determine two

different finite sets. In other words, there is a bijection from the set of nonincreasing functions with range A to a subset of the set of finite subsets of $\mathcal{N}$. It follows from the solution to Exercise 10.31(b) that the set of nonincreasing functions with range A is countable. Since the set of all nonempty finite sets A is also countable (by the same exercise), the set of all nonincreasing functions is a countable union of countable sets and is therefore countable.

(h) and (i) Both countable, since both are subsets of the set of recursively enumerable languages over $\{0, 1\}$.

10.32. The set of all subsets of the even integers has this property. It is uncountable because there is a bijection from $\mathcal{N}$ to the set of even integers, and $2^{\mathcal{N}}$ is uncountable. Its complement is uncountable, because it contains every nonempty subset of the odd integers, and the set of all such subsets is uncountable for the same reason that the set of subsets of the even integers is.

10.33. We know that the set of recursively enumerable languages is countable, and therefore the set of languages whose complement is recursively enumerable is countable. Therefore, the set of languages L so that either L or L' is recursively enumerable is countable. However, the set of languages is uncountable. Therefore, the set of languages L so that neither L nor L' is recursively enumerable is uncountable.

10.34. We consider the contrapositive of the statement. Consider a TM T that halts except on the input strings $x_1, x_2, \ldots, x_n$. We may construct another TM T_1 as follows. T_1 first operates like an FA in order to determine whether the input string is one of the x_i's. (The set $\{x_1, x_2, \ldots, x_n\}$ is a regular language.) If it is, T_1 rejects. Otherwise, T_1 returns the tape head to square 0 and begins to execute T on the original input. Clearly, T_1 accepts precisely the same strings that T does, but T_1 halts on every input. The conclusion is that any language that can be accepted by a TM that enters an infinite loop on only finitely many input strings is recursive.

10.35. Here is a sketch of the construction of a nondeterministic TM T to accept L_1L_2, given TMs T_1 and T_2 accepting L_1 and L_2, respectively. T nondeterministically inserts a blank into the input string at some point, so that the resulting tape is of the form $\Delta x_1 \underline{\Delta} x_2$. T simulates T_2 on the string x_2, and, if and when that simulation accepts, erases the portion of the tape beginning at the end of x_1 and simulates T_1 on x_1. If the input string is of the form x_1x_2, where each $x_i \in L_i$, then the choice of moves that causes the blank to be inserted between them will cause T to accept. If the input is not of this form, then no matter where the blank is inserted, at least one of the simulations will fail to accept.

10.36. The statement $E(f)$ is true if and only if f is Turing-computable.

Suppose first that f is computable. Then the argument sketched just before Theorem 10.6 can be carried out almost without any change to show that if L is recursive then L can be enumerated in order f. In addition, the argument given in the solution to Exercise 10.5 can be adapted to show the converse: the only requirement is that given two strings,

we need to be able to determine which comes first in the ordering f, and we can do this provided f is computable.

Now suppose that for any language L, L is recursive if and only if L can be enumerated in order f. Then in particular, since Σ^* is recursive, Σ^* can be enumerated in order f. This obviously implies that f is computable, since $f(i)$ is simply the ith string in the enumeration of Σ^*.

10.37. Suppose first that f is computable, say by the TM T_f. Then we can construct a TM T to accept $g(f)$ as follows. On an input string $x\#y$, T saves the string y and executes T_f on the string x. If this computation halts, T compares the resulting output to y and halts if and only if they agree. Then if $y = f(x)$, clearly T accepts $x\#y$, and otherwise, T doesn't—either because f is undefined on x, which means that T_f fails to halt on input x, or because $f(x)$ is defined but different from y.

Suppose on the other hand that $g(f)$ is recursively enumerable. Then there is a TM T accepting this set. If $g(f)$ were actually recursive, then in order to compute $f(x)$, our strategy would be to consider the strings $x\#y_1$, $x\#y_2$, ..., where y_1, y_2, ...is an enumeration of Σ^* in canonical order; we would try these strings in order as inputs to T until we found one that was in $g(f)$, and the second part of it would then be $f(x)$. (If we never found one in $g(f)$, the process would never halt, and this would be appropriate because $f(x)$ is undefined in this case.) Since $g(f)$ is assumed only to be recursively enumerable, we must refine this strategy in a way similar to the proof of Theorem 10.6. We execute one move of T on $x\#y_1$; then two moves of T on this string and one on $x\#y_2$; then three moves of T on $x\#y_1$, two on $x\#y_2$, and one on $x\#y_3$; and so forth. If $x\#y_i \in g(f)$ for some i, then we will eventually complete the computation by which T accepts this string, and we will have computed $f(x) = y_i$; otherwise, the process will never halt.

10.38. If L is the range of a computable partial function f, then let T be a TM computing f. If f were a total function, then a strategy to accept L would be, given some input x, to successively compute $f(y_1)$, $f(y_2)$, ...(where y_1, y_2, etc. is an enumeration of Σ^* in canonical order) until one of the values is x, and to accept x if and only if that happened. Since we can't be sure this strategy will work, we make the usual modification: execute T for one step on y_1; then execute T for two steps on y_1 and for one step on y_2; then execute T for three steps on y_1, two steps on y_2, and one step on y_3; etc. This will allow us to compare $f(y_i)$ to x for every i for which the computation of T halts, which means that if x is in the range of f the process will halt on the input string x. If x is not in the range of f, then we will never find an i for which $f(y_i) = x$, and the process will not halt.

Suppose on the other hand that L is recursively enumerable, and let T be a TM accepting L. L is the range of the partial function f, defined by $f(x) = x$ if $x \in L$ and undefined otherwise. We can attempt to compute f on an input x by executing T on x, halting with output x if this computation halts and failing to halt otherwise. Then this process computes f.

10.39. Suppose a virus tester IsSafe exists. Then there is a program D that, on input P, evaluates IsSafe(PP), and either prints out "XXX" (if the value is "NO") or alters the

operating system otherwise. Now consider program D acting on input D. If D is safe on input D, it is because IsSafe(DD) is "NO", which makes IsSafe incorrect. If input D causes D to alter the operating system, there are two possibilities: i) IsSafe(DD) is "YES" (which also means IsSafe is incorrect); or ii) the alteration of the operating system occurs as a result of invoking the program IsSafe, in which IsSafe is not safe. In any case, IsSafe cannot be both safe and correct. (Both the exercise and the solution are taken from the article "There Are No Safe Virus Tests," by William F. Dowling, found in the *American Mathematical Monthly*, November 1989, pp. 835-836.)

10.40. Suppose L is infinite and recursively enumerable. Then there is an algorithm for listing the elements of L. We modify the algorithm so that it examines the strings in the same order but lists each one only if it follows the most recently listed one in canonical order. The modified algorithm obviously lists a subset L_1 of L in canonical order. Moreover, L_1 is infinite, because for each string x, the set of elements of L that precede x in canonical order is finite. Therefore, by Theorem 10.7, L_1 is recursive.

10.41. (a)

$$S \to SB \mid SC \mid SABC \mid BC$$
$$AB \to BA \qquad AC \to CA \qquad BC \to CB$$
$$BA \to AB \qquad CA \to AC \qquad CB \to BC$$
$$A \to a \qquad B \to b \qquad C \to c$$

(b)

$$S \to ABCS \mid ABBCS \mid AABBCS \mid$$
$$BCS \mid BBCS \mid CS \mid BC$$
$$AB \to BA \qquad AC \to CA \qquad BC \to CB$$
$$BA \to AB \qquad CA \to AC \qquad CB \to BC$$
$$A \to a \qquad B \to b \qquad C \to c$$

It is fairly easy to see that any string generated by this grammar satisfies the desired inequalities. The other direction is less obvious. If we let n_A, n_B, and n_C be the number of A's, B's, and C's, respectively, in a string obtained by the first seven productions, and we let $x = n_A$, $y = n_B - n_A - 1$, and $z = 2n_C - n_B - 1$, then we may write the equations

$$\begin{array}{lllllllllll} x & = & n_1 & + & n_2 & + & 2n_3 & & & & \\ y & = & & & n_2 & & & + & n_4 & + & 2n_5 \\ z & = & n_1 & & & & & + & n_4 & + & 2n_6 \end{array}$$

where, for each $i \leq 6$, n_i is the number of times the ith production is used. It is sufficient to show that for any choice of x, y, and z whose sum is even, there are nonnegative integer values for all the x_i's that satisfy the equations. This can be checked by considering cases.

When x, y, and z are all even, for example, it is easy to see that there is a solution in which n_1 is the minimum of x and z and $n_2 = n_4 = 0$.

(c)

$$S \to LA*R \qquad A \to a$$
$$L \to LA*I \qquad IA \to AI \qquad I* \to A*I \qquad IR \to R$$
$$L \to E \qquad EA \to AE \qquad E* \to E \qquad ER \to \Lambda$$

10.42. We will show that each production $\alpha \to \beta$, where $|\beta| \geq |\alpha|$, can be replaced by a set of productions of the desired form, so the new grammar generates the same language.

Suppose $\alpha = \gamma_1 A_1 \gamma_2 A_2 \gamma_3 \dots \gamma_n A_n \gamma_{n+1}$, where the A_i's are variables and each γ_i is a string of terminals. The first step is to introduce new variables $X_1, \dots, X_n$, and productions of the desired type that allow the string $\gamma_1 X_1 \gamma_2 \dots \gamma_n X_n \gamma_{n+1}$ to be generated. These new variables can appear only beginning with the string α, and will be used only to obtain the string β; the effect of this first step is to guarantee that none of the productions we are adding will cause strings not in the language to be generated. The first production is $\gamma_1 A_1 \gamma_2 \dots \gamma_n A_n \gamma_{n+1} \to \gamma_1 X_1 \gamma_2 \dots \gamma_n A_n \gamma_{n+1}$, the next allows A_2 to be replaced by X_2, and so forth; the nth production allows A_n to be replaced by X_n.

The second step is to introduce additional variables, and productions that allow us to generate a string $Y_1 Y_2 \dots Y_k$, where each Y_i is a variable and $k = |\alpha|$. For example, if $\gamma_1 = abc$, we can start with the productions $cX_1 \to Y_3 X_1$, $bY_3 \to Y_2 Y_3$, and $aY_2 \to Y_1 Y_2$. The variable Y_4 will be the same as X_1. Similarly, if $\gamma_2 = defg$, we would use $X_1 d \to X_1 Y_5$, $Y_5 e \to Y_5 Y_6$, etc. All these productions are of the right form, and they obviously allow us to obtain the string $Y_1 \dots Y_k$. The Y_i's, like the X_i's, are reserved for this purpose and are used nowhere else in the grammar.

The third step is to add still more productions that produce the string β. Let $\beta = Z_1 Z_2 \dots Z_k Z_{k+1} \dots Z_m$, where $m \geq k$ and each Z_i is either a variable or a terminal. The productions we need are $Y_1 Y_2 \to Z_1 Y_2$, $Y_2 Y_3 \to Z_2 Y_3$, $\dots$, $Y_{k-1} Y_k \to Z_{k-1} Y_k$, and $Z_{k-1} Y_k \to Z_{k-1} Z_k Z_{k+1} \dots Z_m$.

It is clear that all the productions we have added are of the proper form, and that they permit β to be derived from α. Furthermore, each of the new productions has a new variable on the right side except the last production mentioned above; and the left side of this last production can have been obtained only starting from the string α. The conclusion is that only the strings derivable in the original grammar can be derived in the new one.

10.45. (a) Let S_1 and S_2 be the start symbols of G_1 and G_2, respectively. Let G have new start symbol S, plus all the variables in G_1 and G_2, plus three other new variables L, M, and R. G will have all the productions in G_1 and G_2, together with the new productions

$$S \to LS_1 M S_2 R \qquad L\sigma \to \sigma L \qquad LM \to L \qquad LR \to \Lambda$$

(for every $\sigma \in \Sigma$). The idea is that M prevents any interaction between the productions of G_1 and those of G_2. L must eventually move to the right side of the string, in order to be eliminated. This means that it must first pass M, and once this has happened (using

the production $LM \to L$), there are never any variables preceding M in the string. This implies that the derivation in G_1 must be completed before $LM \to L$ is applied, and it is impossible for any production to be used whose left side involves both terminals produced by G_1 and variables in G_2.

(b) If S_1 is the start symbol of G_1, G will have the variables of G_1 as well as S (the new start symbol), S', L, M, and R. The productions of G will be

$$S \to \Lambda \mid LS' \qquad S' \to S_1MS' \mid S_1R$$
$$L\sigma \to \sigma L \qquad LM \to L \qquad LR \to \Lambda$$

The principle is similar to that in (a). From S we can get strings of the form Λ, LS_1R, LS_1MS_1R, $LS_1MS_1MS_1R$, etc. The derivation from one S_1 can't interfere with the derivation from another because of the presence of the M's.

10.46. The crucial aspect of the formula $A = \{i \mid i \notin A_i\}$ is that it specifies a function $f : \mathcal{N} \to \mathcal{N}$ so that for every i, $f(i) \in A_i$ if and only if $f(i) \notin A$. (The function is $f(i) = i$.) This allows us to conclude that for every i, $A \neq A_i$. The proof would work just as well with any other one-to-one function f. For example, we could let A be $\{2i \mid 2i \notin A_i\}$, or $\{3^i \mid 3^i \notin A_i\}$. (If f is not 1-1, this is not guaranteed to work. If $f(i) = f(j)$, for example, we might want $f(i)$ to be in A because it is not in A_i and $f(j)$ not to be in A because it is in A_j.)

10.47. (a) The function that takes s to $\{s\}$ is a bijection from S to a subset of 2^S. (b) To show that there is no bijection from S to 2^S, suppose f is such a bijection, and let $A = \{x \in S \mid x \notin f(x)\}$. Then since f is onto and $A \in 2^S$, $A = f(x_0)$ for some $x_0 \in S$. We ask whether x_0 can be an element of $f(x_0)$. On the one hand, if $x_0 \in f(x_0)$, then x_0 does not satisfy the defining condition for A, and so $x_0 \notin A = f(x_0)$. On the other hand, if $x_0 \notin f(x_0)$, then by definition of A, $x_0 \in A = f(x_0)$. Thus we have a contradiction, and we may conclude that the bijection f is impossible.

10.48. (a) For each n, there is a bijection between the $(n+1)$-fold Cartesian product $\mathcal{N} \times \mathcal{N} \times \ldots \times \mathcal{N}$ and the set of all integer polynomials of degree n. Therefore, the set of polynomials of degree n is countable, and thus the set of real numbers that are roots of such polynomials is countable. The set of all such numbers is the union of these sets, from $n = 0$ to ∞, and is therefore countable.

(b) For every subset $S = \{n_0, n_1, \ldots\}$ of $\mathcal{N}$ (assume the n_i's are listed in increasing order) we may consider the function f which is 0 for all i with $0 \leq i < n_0$, 1 for all i with $n_0 \leq i < n_1$, 2 for all i with $n_1 \leq i < n_2$, etc. This correspondence defines a bijection from $2^{\mathcal{N}}$ to a subset of the set of all nondecreasing functions from $\mathcal{N}$ to $\mathcal{N}$. Therefore, the set of such functions is uncountable.

(c) This set contains the one in Exercise 10.31(d) as a subset and is therefore uncountable.

(d) The set is finite, by an argument essentially the same as that in the solution to Exercise 10.31(g).

10.48. (e) Suppose f is periodic, and $P > 0$ is a *period* (i.e, $f(x + P) = f(x)$ for every x). Let $f_1 : \{0, 1, \ldots, P-1\} \to \mathcal{N}$ be defined by $f_1(n) = f(n)$ for every n with $0 \leq n \leq P-1$. Then if g is any periodic function other than f for which P is a period, the corresponding function g_1 is different from f_1. This means that there is a bijection from the set $\mathcal{F}_P$ of all functions having period P to the set of all functions from $\{0, 1, \ldots, P-1\}$ to $\mathcal{N}$. Since the second set is countable (see part (e)), the set $\mathcal{F}_P$ is. It follows that the set of all periodic functions is also, since this set is the union of all the $\mathcal{F}_P$'s for $P = 1, 2, \ldots$.

(f) We can think of an eventually periodic function as a periodic function which, for some n, has had its values at the first n integers changed arbitrarily. For a given periodic function f, there are only a countable number of n's for which this could happen, and for each n only a countable number of ways of changing the function at the first n integers. Therefore, the set of functions that are eventually equal to f is countable. Since the set of periodic functions is countable, the set of all eventually periodic functions is countable.

(g) This set is a subset of (f) and is therefore countable.

10.49. (a) If we define $f_0 : A_0 \to B$ by $f_0(x) = f(x)$, then f_0 is obviously one-to-one, since f is. For any $x \in A_0$, $f_0(x) \in B_1$, because the number of ancestors of $f_0(x)$ is one plus the number of ancestors of x. Finally, for any $y \in B_1$, $y = f(x)$ for some $x \in A_0$. Therefore, f_0 is a bijection from A_0 to B_1. Similarly, g_0 is a bijection from B_0 to A_1. It is also easy to see that the function f_∞ defined by $f_\infty(x) = f(x)$ is a bijection from A_∞ to B_∞.

Now we define $F : A \to B$ by letting $F(x) = f(x)$ if $x \in A_0$ or $x \in A_\infty$ and $F(x) = g^{-1}(x)$ if $x \in A_1(x)$. It follows that F is a bijection from A to B.

(b) Let $f : A \to B$ and $g : B \to C$ be the two bijections. Then $g \circ f$ is a bijection from A to a subset of C. If there were a bijection $h : A \to C$, then h^{-1} would be a bijection from C to A. Thus $h^{-1} \circ g$ would be a bijection from B to a subset of A. The Schröder-Bernstein Theorem would then imply that there was a bijection from B to A, but we are assuming this is not the case.

(c) This is an immediate consequence of the Schröder-Bernstein Theorem.

(d) If A is an infinite set, A has a countably infinite subset B by Lemma 10.1. It follows that if A is uncountable and C is countable, A is larger than C. If A is infinite and C is countable, A cannot be smaller than C: If there is a bijection from A to a subset S of C, since S cannot be finite, S must be countably infinite, and therefore A is also.

10.50. The function $f : I \to I \times I$ defined by $f(x) = (x, 0)$ is a bijection from I into a subset of $I \times I$. Now we define a bijection g from $I \times I$ into a subset of I. Given an element (x, y) of $I \times I$, let $0.x_0x_1x_2\ldots$ and $0.y_0y_1y_2\ldots$ be the decimal expansions of x and y, respectively (that do not end in infinite strings of 9's), and let $g(x, y)$ be the real number with decimal expansion $0.x_0y_0x_1y_1\ldots$. Then $g(x, y) \in I$, and g is a bijection from $I \times I$ to a subset of I. (In fact, g is not onto, since for example the number with expansion $0.1119191919\ldots$ is not in the range of g.) It follows that there is a bijection from I to $I \times I$.

10.51. Suppose there is a 1-1 function f from B to A and none from A to B. Then f can be interpreted as a bijection from B to its range, which is a subset of A. If there is no 1-1

function from A to B, then there can certainly not be a bijection from B to A. Therefore, A is bigger than B.

Now suppose A is bigger than B. Then there is a bijection from B to a subset of A, which can be interpreted as a 1-1 function from B to A. Suppose there is a 1-1 function from A to B. Then it determines a bijection from A to a subset of B, and it follows from the Schröder-Bernstein Theorem that there is a bijection from A to B, which is impossible.

Chapter 11

Unsolvable Problems

11.1. For any language L, the identity function from Σ^* to Σ^* reduces L to itself.

Suppose $f : \Sigma^* \to \Sigma^*$ and $g : \Sigma^* \to \Sigma^*$ have the property that for every x and y in Σ^*, $x \in L_1$ if and only if $f(x) \in L_2$, and $y \in L_2$ if and only if $g(y) \in L_3$. Then for any $x \in \Sigma^*$, $x \in L_1$ if and only if $g(f(x)) \in L_3$. In addition, if f and g are computable, their composition $g \circ f$ is also computable. Therefore, if f reduces L_1 to L_2 and g reduces L_2 to L_3, $g \circ f$ reduces L_1 to L_3.

We want an example of two languages L_1 and L_2 for which $L_1 \leq L_2$ but $L_2 \not\leq L_1$. A trivial example is for L_1 to be Σ^* and for L_2 to be $\{\Lambda\}$. The constant function $f : \Sigma^* \to \Sigma^*$ defined by $f(x) = \Lambda$ for every x reduces L_1 to L_2. However, for any $g : \Sigma^* \to \Sigma^*$, it can't be true that for every x, $x = \Lambda$ if and only if $g(x) \in \Sigma^*$.

11.2. (a) Given n, is evenly divisible by 10? (Reason: for any n, n is divisible by 2 if and only if $5n$ is divisible by 10.)

(b) One possibility is

$$g(n) = \begin{cases} k/5 & \text{if } k \text{ is divisible by 5} \\ 1 & \text{otherwise} \end{cases}$$

If n is divisible by 10, then $g(n) = k/5$, which is divisible by 2. If n is not divisible by 10, then either n is divisible by 5 and not by 2, so that $g(n)$ is not divisible by 2, or n is not divisible by 5, which also implies that $g(n)$ is not divisible by 2.

11.3. Let $f : \Sigma^* \to \Sigma^*$ be a function that reduces L_1 to L_2, let T_f be a TM computing f, and let T_2 be a TM accepting L_2. Then the composite TM $T_f T_2$ accepts L_1, because if the input string x is in L_1, $f(x) \in L_2$, so that T_2 accepts $f(x)$; and if $x \notin L_1$, then $f(x) \notin L_2$, so that T_2 does not accept $f(x)$.

11.4. Let x_0 be a string in L, and let y_0 be a string in L'. For any recursive language L_1, define $f : \Sigma^* \to \Sigma^*$ by $f(x) = x_0$ if $x \in L_1$ and $f(x) = y_0$ if $x \notin L_1$. Obviously, for any x, $x \in L_1$ if and only if $f(x) \in L$. Furthermore, f is computable, because L_1 is recursive.

11.5. (See the solution to Exercise 10.8.) Choose some algorithmic way to enumerate the 4-tuples (x, y, z, n) in $\mathcal{N} \times \mathcal{N} \times \mathcal{N} \times \mathcal{N}$ for which $n \geq 3$. It is then straightforward to build a TM that tests 4-tuples in this order, and halts if and only if it finds one satisfying $x^n + y^n = z^n$. Therefore, knowing whether this TM ever halts on the empty input is equivalent to knowing whether Fermat's last theorem is false.

11.6. Suppose L is recursively enumerable, and let T be a TM accepting L. Define $f : \Sigma^* \to \Sigma^*$ by $f(x) = e(T)e(x)$. Then if $x \in L$, T accepts x, and thus $f(x) \in Acc$. If $x \notin L$, then T doesn't accept x, and $f(x) \notin Acc$. The function f is computable, since

computing $f(x)$ simply amounts to encoding the TM T and the string x.

11.7. If the language L is accepted by a TM T, and the question Given x, does T accept x? is solvable, then L is recursive, since there is an algorithm to determine whether any given string is in L.

11.8. An instance of **Accepts** is a pair (T, y), where T is a TM and y is a string. An instance of **Accepts**-x is simply a TM. Consider the algorithm that takes a pair (T, y) and constructs a TM T' to work as follows. T' starts by comparing its input to the string x. If the input string is not x, T' simply executes T on the input. If the input is x, then T' replaces it by y and then executes T on y. Then T' accepts x if and only if T accepts y. Therefore, the algorithm reduces **Accepts** to **Accepts**-x.

11.9. In both cases, an instance of the problem is a TM. We need an algorithm that takes an arbitrary TM T and constructs another TM T' having the property that T accepts Λ if and only if $L(T') = \{\Lambda\}$. We may obtain T' by letting it execute T if the input is Λ and immediately rejecting every other input. If T accepts Λ, then T' accepts Λ and rejects every other string, so that $L(T') = \{\Lambda\}$; if T does not accept Λ, then T' still rejects every nonnull input, but now it fails to accept Λ as well, so that $L(T') = \emptyset \neq \{\Lambda\}$.

11.10. (a) We may take $C = A \cup B$ and $D = A \cap B$.

(b) Given two TMs T_1 and T_2, we may construct T_1' and T_2' so that $L(T_1') = L(T_1) \cup L(T_2)$ and $L(T_2') = L(T_1) \cap L(T_2)$ (see the proof of Theorem 10.3). Then $L(T_1) = L(T_2)$ if and only if $L(T_1') \subseteq L(T_2')$. Therefore, the construction is a reduction from **Equivalent** to **Subset**.

11.11. (a) Let $C = A \cap B$ and $D = A$. Then $A \subseteq B$ if and only if $C = D$.

(b) Given T_1 and T_2, let T_1' be a TM accepting $L(T_1) \cap L(T_2)$ and let $T_2' = T_2$. Then the algorithm that constructs T_1' and T_2' from T_1 and T_2 determines a reduction from **Subset** to **Equivalent**.

11.12. (a) This problem is solvable. As T proceeds in its computation, one of the following must eventually occur: i) T changes state; ii) T crashes without having changed state; iii) T moves its tape head to the right, without having changed state; iv) without having changed state, T writes a symbol on square 0 that it has previously written. In the first two cases we have the answer to the question. In either of the last two cases, T must be in an infinite loop, and thus we may answer the question negatively.

(b) To show this problem is unsolvable, we reduce **Accepts**(Λ) to it. Given a TM T, an instance of **Accepts**(Λ), construct a pair (T_1, q), an instance of this problem, as follows. T_1 has all the states T does as well as one additional one, called q. T_1 works by making exactly the same moves as T, except that if T ever halts, T_1 moves instead to state q, then halts on the next move. Clearly T accepts the string Λ if and only if T_1 eventually enters state q.

(c) Exactly the same construction as in (b) shows that **AcceptsSomething** is reducible to this problem; therefore, this problem is unsolvable.

(d) To show that this problem is unsolvable, we reduce **Accepts**(Λ) to it. Given T, construct T_1 so that T_1 copies T but also keeps track of whether it's made an even number of moves or an odd number. (This is easy to do. Instead of using the states p of T, just use pairs $(p, 0)$ and $(p, 1)$ for the states of T_1. A move by T from p to q corresponds to moves by T_1 from $(p, 0)$ to $(q, 1)$ and from $(p, 1)$ to $(q, 0)$.) The other difference between T and T_1 is that if T accepts after an odd number of moves, then T_1 makes an additional move before accepting. The result is that T_1 accepts exactly the same strings as T, but accepts only after an even number of moves. Therefore, T accepts Λ if and only if T_1 accepts Λ after an even number of moves.

(e) Unsolvable. The same construction as in (d) reduces **AcceptsSomething** to this problem.

(f) Given a TM T, construct a new TM T_1 that accepts the same language but never rejects (see Exercise 9.11). Then for any input x, T fails to accept x if and only if T_1 loops forever on x. This means that the problem P: Given a pair (T, x), does T fail to accept x? can be reduced to P_1: Given (T, x), does T loop forever on x? Since P is unsolvable (it is the complementary problem to **Accepts**), P_1 is also.

(g) The same construction as in (f) reduces the problem Given T, is there a string T fails to accept? to this one. Since the first problem is unsolvable, so is this one.

(h) Given a TM T, it is possible to construct another one, T', having the property that for any input string x, T accepts x if and only if T' rejects x. (The construction is essentially to interchange the states h_a and h_r.) Therefore, the problem **Accepts** can be reduced to this problem, which implies that this problem is unsolvable.

(i) The construction in (h) reduces the problem **AcceptsSomething** to this one, which implies that this one is unsolvable.

(j) and (k) are both solvable. The reason is that within ten moves, a TM can't move its tape head any farther right than square 10. Therefore, if necessary we can look at the first 10 moves T makes for every possible input string of length 10 or less. At the end of this process, we will know whether T halts within 10 moves on every string, and we will know whether there are any strings for which T halts within 10 moves.

(l) Consider a specific language that is neither $\emptyset$ nor Σ^*, say $\{\Lambda\}$. Given a TM T, construct two TMs T_1 and T_2 by letting T_1 be T and T_2 be any TM that accepts precisely the language $\{\Lambda\}$. Then $L(T_1) \subseteq L(T_2)$ or $L(T_2) \subseteq L(T_1)$ if and only if $L(T) \subseteq \{\Lambda\}$ or $\{\Lambda\} \subseteq L(T)$. Therefore, the decision problem P: Given T, does $L(T)$ have the property that $L(T) \subseteq \{\Lambda\}$ or $\{\Lambda\} \subseteq L(T)$? can be reduced to this one. However, this property is a nontrivial property of recursively enumerable languages, so that P is unsolvable by Rice's Theorem. Therefore, the given problem is unsolvable.

11.14. There is an algorithm to take an arbitrary TM T and find a grammar generating the language $L(T)$. Therefore, to each of these problems, the corresponding problem involving TMs can be reduced. It follows that all four problems are unsolvable.

11.15. Suppose T has n nonhalting states. Within n moves, T will have halted or entered

some nonhalting state q for the second time. If T has not written a nonblank symbol by that time, it never will—in the second case because it's in an infinite loop.

11.16. The construction reduces a problem to an unsolvable problem, which doesn't prove anything.

11.17.

$\#$	$q_0\Delta$	a	b	$\#$	Δ	q_1a	b	$\#$	Δ	a	q_1b	$\#$	
$\#q_0\Delta ab\#$	Δq_1	a	b	$\#$	Δ	aq_1	b	$\#$	Δ	a	bq_1	$\#$	

Δ	a	$bq_1\#$	Δ	a	q_2b	Δ	$\#$	Δ	$ah_a\Delta$	Δ	$\#$	$\Delta h_a\Delta$	$\#$	$h_a\#\#$
Δ	a	$q_2b\Delta\#$	Δ	a	$h_a\Delta$	Δ	$\#$	Δ	h_a	Δ	$\#$	h_a	$\#$	$\#$

11.18. (a) Any partial solution must begin with domino 1. The next two dominoes are then determined, and both must be domino 2. At that point, the only domino that can follow is domino 1, but it is then impossible to follow it with anything.

(b)

1	001	01
10	0	101

11.19. We can construct a reduction from the general correspondence problem to the restricted problem in which the alphabet has only two symbols, by simply encoding more general symbols by strings of 0's and 1's. This can be done so that any string of 0's and 1's represents at most one string of the original symbols. Then there will be a solution sequence for the original instance if and only if there is a solution sequence for the instance of the restricted problem.

11.20. Given an instance (α_1, β_1), (α_2, β_2), ..., (α_n, β_n) of **PCP** in which $\alpha_i, \beta_i \in \{a\}^*$ for each i, let $d_i = |\alpha_i| - |\beta_i|$. If $d_i = 0$ for some i, the instance is obviously a yes-instance. If either $d_i > 0$ for every i or $d_i < 0$ for every i, the instance is obviously a no-instance. Finally, if $d_i = p > 0$ and $d_j = -q < 0$ for some i and j, then it is a yes-instance, because $\alpha_i^q \alpha_j^p = \beta_i^q \beta_j^p$.

11.21. (a) It is easy to show that the problem **CFGGeneratesAll** is reducible to this one.

(b) **CFGGeneratesAll** is also reducible to this one. Given a CFG G, let G_1 be a CFG generating Σ^* and $G_2 = G$. Then $L(G) = \Sigma^*$ if and only if $L(G_1) \subseteq L(G_2)$.

(c) Since Σ^* is regular, **CFGGeneratesAll** is also reducible to this problem.

11.23. We can construct the function $g : \Sigma_1^* \to \Sigma_2^*$ as follows. If $x \in \Sigma_1^*$ represents an instance of P_1, say I, then $g(x) = e_2(f(I))$, and otherwise $g(x) = z$, where z is a string in Σ_2^* that does not represent an instance of P_2. Then if $x \in Y(P_1)$, $x = e_1(I)$ for some yes-instance I of P_1; then $f(I)$ is a yes-instance of P_2, and $e_2(f(I)) \in Y(P_2)$. If $x \notin Y(P_1)$, then either x is not an instance of P_1 or $x = e_1(I)$ for some no-instance I of P_1. In either case, $g(x) \notin Y(P_2)$.

Note that there is one potential problem with this: There may be strings in Σ_1^* that are not instances of P_1 but no strings in Σ_2^* that are not instances of P_2. The statement of the

exercise should therefore include the assumption that not every string in Σ_2^* represents an instance of P_2.

11.24. Let y_0 be a fixed string encoding a no-instance of P_2, and consider the function t' defined by $t'(x) = t(x)$ if $t(x)$ is an instance of P_2, and $t'(x) = y_0$ otherwise. Since the function t reduces P_1 to P_2, clearly t' takes yes-instances to yes-instances. In addition, if x represents a no-instance of P_1, then $t'(x)$ represents a no-instance of P_2: If $t(x)$ already represents a no-instance of P_2, $t'(x) = t(x)$, and otherwise, $t'(x) = y_0$. t' is computable, because t is and because it is possible to determine algorithmically whether a string represents an instance of P_2.

11.25. According to the assumption, if I is an instance of P_1, $t(e_1(I))$ represents an instance J of P_2. It is easy to check that the function f that assigns to each I the corresponding J is in fact a reduction of P_1 to P_2.

11.26. (a) We can define $f : \{0,1\}^* \to \{0,1\}^*$ as follows. For a string x not of the form $e(T)e(w)$, $f(x) = \Lambda$; if $x = e(T)e(w)$, $f(x) = e(T')$, where the TM T' works by substituting the string w for its input and then executing T on w. If $x \in Acc$ (i.e., $x = e(T)e(w)$ and T accepts w), then $f(x) = e(T')$, where T' accepts every string, so that $f(x) \in AE$. If $x \notin Acc$, then either $f(x) = e(T')$ for some T' not accepting everything (in fact, not accepting anything), or $f(x) = \Lambda$; in either case, $f(x) \notin AE$.

(b) Any function that determines a reduction from Acc to AE also determines a reduction from Acc' to AE'.

(c) The language Acc is recursively enumerable. We can construct a TM T to accept Acc as follows. T tests its input string to see if it is of the form $e(T)e(w)$. If not, it rejects; if so, it then executes a universal TM on the original input. The result is that T accepts the input precisely if it is an element of Acc. Now, since we know that the decision problem **Accepts** is unsolvable, Acc cannot be recursive. Therefore, by Theorem 10.5, Acc' cannot be recursively enumerable. Therefore, by Exercise 11.3, AE' is not recursively enumerable.

(d) Let T_0 be a TM that immediately accepts, no matter what the input. To complete the definition of f we must say only what $f(x)$ is for a string x not of the form $e(T)e(z)$. Since such an x is an element of Acc', we want $f(x)$ to be an element of AE; for this reason, we define $f(x)$ to be $e(T_0)$ in this case.

Now if $x = e(T)e(z)$, the string $f(x)$ is $e(S_{T,z})$. In the case where T accepts z, say in n moves, the computation performed by $S_{T,z}$ on an input string of length $\geq n$ causes it to enter an infinite loop, because by the time it finishes simulating n moves of T on input z it will discover that T accepts z. Therefore, in this case, the TM $S_{T,z}$ does not accept every string, which implies that $f(x) \notin AE$. In the other case, where T does not accept z, $S_{T,z}$ does accept every input, because no matter how long its input is, simulating that number of moves of T on input z will not cause T to accept. The conclusion is that f does determine a reduction from Acc' to AE.

(e) If AE were recursively enumerable, it would follow from the previous part and Exercise 11.3 that Acc' would be recursively enumerable. But we have seen in part (c) that it is not.

11.28. Suppose L is a recursively enumerable language that is not recursive (for example, the language Acc introduced in Exercise 11.26). Then by Theorem 10.5, L' is not recursively enumerable. On the one hand, $L' \not\leq L$, by the result in Exercise 11.3. On the other hand, if $L \leq L'$, then the same function that reduces L to L' also reduces L' to $(L')' = L$; therefore $L \not\leq L'$.

11.29. (a) Given a TM T, we construct a TM T' as follows. T' first compares its input to y; if the input is different from y, it simply executes T on its input, and if the input is y, T' replaces it by x before executing T. Then T accepts x if and only if T' accepts y. Therefore, this construction defines a reduction from $P_{\{x\}}$ to $P_{\{y\}}$.

(b) To get a reduction from $P_{\{x\}}$ to $P_{\{y,z\}}$, we can use the same construction as in (a), except that T' replaces both the input y and the input z by x. It follows that T accepts x if and only if T' accepts both y and z.

(c) For this part the TM T' is constructed from T as follows. If the input is anything other than z, T' executes T. If the input is z, T' first replaces z by x and executes T on x. If (and only if) that computation would cause T to accept x, T' then erases its tape, writes y on the tape, and executes T on the input y, accepting precisely if T would accept y. Then T accepts both x and y if and only if T' accepts z. This construction provides the reduction from $P_{\{x,y\}}$ to $P_{\{z\}}$.

(d) If S is any nonempty finite set, the argument in (c) shows that $P_S \leq P_{\{\Lambda\}}$. If U is any nonempty finite set, the argument in (b) shows that $P_{\{\Lambda\}} \leq P_U$. It follows that for any two finite nonempty sets S and U, $P_S \leq P_U$.

11.30. (a) We can construct T' from T as in (a) of the previous exercise, except that here T' replaces the input y by the input x and vice versa before executing T. This guarantees that T accepts x and nothing else, if and only if T' accepts y and nothing else.

In (b), T' replaces both y and z by x. As in the previous exercise, this guarantees that if T accepts x, then T' accepts y and z, and conversely. Now, since on input y and z, T' does not actually execute T on those strings, we must be careful to guarantee that if T' accepts nothing besides y and z, then T accepts nothing but x. (In particular, we must make sure it doesn't accept y or z.)

Let us assume that $x \notin \{y, z\}$ and that in lexicographic order, x precedes y and y precedes z. The other cases can be handled similarly. We wish to fix it so that the strings on which T' actually executes T, if the input of T' is neither y nor z, include all the strings other than x. T' can replace input x by y before executing T. In addition, for every string that comes after z in lexicographic order, T' replaces that input by the string that immediately precedes it.

Now, if T accepts x and nothing else, then T' accepts y and z and nothing else. (It doesn't accept x, because input x has been replaced by y, which is not one of the strings T accepts.) Conversely, if T' accepts y and z and nothing else, then T accepts x. Furthermore, it doesn't accept any other strings, because for any $w \neq x$, some input other than y or z would cause T' to process w by running T on it. This gives us the reduction we need.

(c) We assume that $z \notin \{x, y\}$. As in (c) of the previous exercise, we have T' replace input z by x, run T on x, and if it accepts, run it on input y. This will guarantee that if T

accepts x and y, then T' accepts z, and conversely. Now we must see to it that the strings on which T can be executed by T', if the input to T' is not z, include z but neither x nor y. We can accomplish this by simply having T' replace both input x and input y by z before running T. It follows that T accepts x and y and nothing else, if and only if T' accepts z and nothing else.

(d) As in the previous exercise, the techniques in (b) and (c) can be easily modified to show that for any nonempty finite sets S and U, $P_S \leq P_{\{\Lambda\}} \leq P_U$.

11.31. (a) To show this problem is unsolvable, we can reduce **Accepts-Λ** to it. Given a TM T, an instance of **Accepts-Λ**, we construct another TM T_1 as follows. T_1 has all of T's states, as well as one additional state q; it has all of T's tape symbols, and one additional one, \$. The transitions of T_1 are the same as those of T, with these additions and modifications. For any accepting move $\delta(p, a) = (h_a, b, D)$, T_1 has instead the move $\delta_{T_1}(p, a) = (q_0, \$, \mathrm{S})$, where q_0 is the initial state of T. If the nonhalting states of T are enumerated $q_0, q_1, \ldots, q_n$, then T_1 has the additional transitions $\delta_T(q_i, \$) = (q_{i+1}, \$, \mathrm{S})$ (for each i with $0 \leq i < n$); finally, $\delta_T(q_n, \$) = (q, \$, \mathrm{S})$ and $\delta_T(q, \$) = (h_a, \$, \mathrm{S})$.

To summarize: for any move that would cause T to accept, T_1 instead moves to q_0 and places \$ on the tape; thereafter, the \$ causes T_1 to cycle through all its nonhalting states, q being the last, before accepting. On the one hand, if T accepts Λ, then some move causes T to accept, and therefore, T_1 enters all its nonhalting states when started with a blank tape. Conversely, if T doesn't accept Λ, then since T never executes an accepting move after starting with a blank tape, T_1 will never enter the state q, and so does not enter all its nonhalting states.

(b) The construction in (a) reduces the problem **AcceptsSomething** to this problem. It follows that this problem is unsolvable.

11.32. We sketch the way an LBA might operate if it is to accept strings representing solutions to the given correspondence system. It starts by creating a second "track" to the tape, which is initially blank. Next, it nondeterministically attempts to decompose the string in the first track (i.e., the input string) into consecutive pieces, each of which is an α_i. The end of each α_i is marked somehow. Assuming the LBA is able to do this successfully, it makes another pass through the string, constructing in the second track of the tape a string of β_i's corresponding to the α_i's in the first track. (This step may require nondeterminism also, if there are strings α_i corresponding to two different β_i's.) Finally, if the resulting string in the second track is exactly the right length, the machine compares the first and second tracks and accepts if the two strings are equal.

11.33. (a) One way to enumerate the CSGs is the following. First, list all CSGs for which the only variable is A_1 and every production $\alpha \to \beta$ has $|\alpha|, |\beta| \leq 1$; there are only finitely many. Next, list those not already listed for which the only variables are A_1 and A_2 and every production $\alpha \to \beta$ has $|\alpha|, |\beta| \leq 2$. If we continue in this way, every CSG satisfying our assumptions will eventually be listed.

(b) Each language $L(G_i)$ is recursive, by Theorem 10.12. Therefore, given $x \in \{a, b\}^*$, we can determine whether $x \in L$ by finding the i for which $x = x_i$ and then testing x for

membership in $L(G_i)$. Therefore, L is recursive. If L is context-sensitive, then L is the same as $L(G_i)$ for some i. In this case, consider whether the corresponding x_i can be an element of L. If it is, then by definition of L, $x \notin L(G_i) = L$. If it is not, however, then $x_i \notin L(G_i)$, and so $x \in L$ by definition of L. This contradiction proves that L cannot be context-sensitive.

11.34. This problem is solvable. A decision algorithm is the following. First test x for membership in $L(G)$. If $x \notin L(G)$, then $L(G) \neq \{x\}$. If $x \in L(G)$, then construct a PDA M accepting $L(G)$, using Section 7.4. Since $L_1 = \{z \in \Sigma^* \mid z \neq x\}$ is regular, the proof of Theorem 8.4 provides an algorithm for constructing another PDA M_1 to accept $L \cap L_1$. Section 7.5 describes an algorithm to produce a CFG G_1 generating $L(M_1)$. In Section 8.3 there is a decision algorithm to decide whether $L(G_1) = \emptyset$. If $L(G_1) = \emptyset$, then $L(G) = \{x\}$; otherwise, $L(G) \neq \{x\}$.

11.35. Using the same technique as in Exercise 11.34, we can construct a CFG generating $L(G) - R$, and we can determine whether this language is nonempty. Since $L(G) \subseteq R$ if and only if $L(G) - R = \emptyset$, this problem is solvable.

11.37. Let I be an instance of **PCP**. As suggested, let G_α and G_β be the CFGs constructed from I. Although the complement of a CFL is not in general a CFL, in this case we can actually construct CFGs G'_α and G'_β generating the complements of $L(G_\alpha)$ and $L(G_\beta)$, respectively. First we can construct a deterministic PDA M_α to accept $L(G_\alpha)$. It works by initially reading and pushing onto its stack all input symbols that are elements of Σ. If and when one of the c_i's is read for the first time, the PDA crashes unless it is able to pop from its stack the symbols of the corresponding α_i in reverse. Thereafter, the only legal inputs are c_i's, and the PDA continues matching them with the reverse of the strings on the stack. If this continues successfully until M_α finally pops from the stack the first α_i put there, so that the stack is empty except for Z_0, M_α accepts. The PDA M_β can be constructed the same way.

Next we construct M'_α and M'_β from M_α and M_β by making the accepting states nonaccepting and vice-versa. In this case (although not in general), the resulting PDAs accept the complements of the original languages. The CFGs G'_α and G'_β are then constructed from the PDAs M'_α and M'_β.

Finally, let G' be a CFG generating the language $L(G'_\alpha) \cup L(G'_\beta) = L(G_\alpha)' \cup L(G_\beta)'$. The instance I is a yes-instance of **PCP** if and only if $L(G_\alpha) \cap L(G_\beta)$ is nonempty. This is true if and only if the complement of this language is different from Σ^*. But the complement of this language is $L(G')$. We have therefore established a reduction of **PCP** to the given problem.

Chapter 12

Computable Functions

12.1. Suppose this function f is computable, and let T_f be a TM with tape alphabet $\{0,1\}$ computing f. Let $T = T_f T_1$, where T_1 is a TM that moves its tape head to the first blank square to the right of its starting position and halts. Suppose T has n states. Then on input n, T makes at least $f(n)+1$ moves moves before halting, since T_1 moves its tape head $f(n)+1$ times. However, by definition of f, no TM with n states and tape alphabet $\{0,1\}$ can make more than $f(n)$ moves, when started with input 1^n. This is a contradiction.

12.2. Suppose f is computable. Then $g = f+1$ is also. Let T_g be a TM computing g, and suppose that T_g has n states and m tape symbols. By adding useless states if necessary, we can assume $n \geq m$. Then on input 1^n, T_g leaves output $1^{f(n)+1}$, which by definition contains more 1's than any TM with n states and m tape symbols can possibly leave on the tape, starting with input 1^n. This is a contradiction.

12.3. For any integer n, there are only a finite number of TMs having n nonhalting states and tape alphabet $\{0,1\}$. If the halting problem were solvable, then we would be able to eliminate all those that did not halt on the input 1^n. We could then compute $bb(n)$ by running each of the remaining ones on this input and seeing which one printed the most 1's.

12.4. For any n, we can construct the following TM T_n. When started with a blank tape, T_n halts in the accepting state with tape $\underline{\Delta}1^n$. Furthermore, it is easy to see that the number of states required for T_n is essentially n: more precisely, $n+k$, for some fixed k independent of n. T_n requires no tape symbols other than 1. Now let f be any computable function, and let T_f be a TM with m states and tape alphabet $\{0,1\}$ that computes f. Let T'_n be the composite TM that first executes T_n, starting with a blank tape, and then executes T_f. The number of states of T'_n is $k'+n$ (where k' is a constant independent of n), and the output produced by T'_n, assuming it begins with a blank tape, is $1^{f(n)}$. It follows from the definition of bb that $bb(k'+n) \geq f(n)$.

Suppose for the sake of contradiction that bb is computable. Then so is the function $bb(2n)$. According to the preceding paragraph, $bb(k'+n) \geq bb(2n)$ for every n. But this is clearly not correct, because for sufficiently large n, there are TMs with $2n$ states and tape alphabet $\{0,1\}$ which, when started with a blank tape, can write more 1's before halting than any such TM having $n+k'$ states.

12.5. It is clear that if f is computable, then the problem: Given n and C, is $f(n) > C$? is solvable. On the other hand, if this problem is solvable, then for any n, $f(n)$ is the smallest value of C for which $f(n) \not> C$, and so there is an obvious algorithm for computing f.

12.6. *PR* would be unchanged. All constant functions of 0 variables can be obtained from the successor function and C_0^0 by repeated applications of composition. Moreover, the

constant function C_k^{n+1} can be obtained from C_k^n and p_{n+2}^{n+2} by primitive recursion. It follows by mathematical induction that all constant functions are primitive recursive according to this definition.

12.7. The only new functions would be the functions f of one variable having the formula $f(x) = x + m$ (for some $m \geq 1$) and the functions g of several variables having the formula $g(x_1, x_2, \ldots, x_k) = x_i + m$ (for some i with $1 \leq i \leq k$ and some $m \geq 1$.

12.9. $f_2(x)$ is 1 if $x = 0$ and 0 otherwise. $f_6(x) = Mod(x, 2)$. $f_{14}(x, y) = Mod(x, 2) + y$. $f_{15}(x) = Div(x, 2)$.

12.10. $g = 0 = C_0^0$; $h(x, y) = y + 2x + 1$.

12.11. (a)
$$\begin{aligned} f_0 &= p_1^1 \\ f_1 &= s \text{ (the successor function)} \\ f_2 &= p_3^3 \\ f_3 &= s(p_3^3) \text{ (obtained from } f_1 \text{ and } f_2 \text{ by composition)} \\ f_4 &= Add \text{ (obtained from } f_0 \text{ and } f_3 \text{ by primitive recursion)} \\ f_5 &= Add(Add, Add) \text{ (obtained from } f_4 \text{ by composition)} \\ f_6 &= p_1^2 \\ f_7 &= f \text{ (obtained from } f_4, f_5, \text{ and } f_6 \text{ by composition)} \end{aligned}$$

(c)
$$\begin{aligned} f_0 &= 1 = C_1^0 \\ f_1 &= s \text{ (the successor function)} \\ f_2 &= s(s) \text{ (obtained from } f_1 \text{ and } f_1 \text{ by composition)} \\ f_3 &= C_0^0 \\ f_4 &= p_2^2 \\ f_5 &= s(s(p_2^2)) \text{ (obtained from } f_2 \text{ and } f_4 \text{ by composition)} \\ f_6 &= 2x \text{ (obtained from } f_3 \text{ and } f_5 \text{ by primitive recursion)} \\ f_7 &= 2p_2^2 \text{ (obtained from } f_6 \text{ and } f_4 \text{ by composition)} \\ f_8 &= f \text{ (obtained from } f_0 \text{ and } f_7 \text{ by primitive recursion)} \end{aligned}$$

(e) Use the fact that $|x - y| = (x \dot{-} y) + (y \dot{-} x)$.

12.12. We prove the result for Add_n by mathematical induction on n, and the result involving *Mult* is similar. The basis step follows by observing that $Add_1(x_1) = p_1^1(x_1)$. Assume that $k \geq 0$ and that Add_k is a primitive recursive function of k variables. We wish to show that Add_{k+1} is a primitive recursive function of $k + 1$ variables. This follows from the formula

$$Add_{k+1}(x_1, \ldots, x_{k+1}) = Add(Add_k(x_1, \ldots, x_k), x_{k+1})$$

and Theorem 13.1.

12.14 For each i satisfying $0 \leq i \leq n_0 + p - 1$, let $m_i = f(i)$. Then $f(n)$ can be defined by

cases as follows. For each $i < n_0$, $f(n) = m_i$ if $n = i$. For each i with $n_0 \leq i \leq n_0 + p - 1$, $f(n) = m_i$ if $n \geq n_0$ and $Mod(n, p) = Mod(i, p)$. Since the cases are all defined by primitive recursive predicates, it follows that f is primitive recursive.

12.15. (a) f can be obtained by primitive recursion from the two functions p_1^1 and h, where h is defined by the formula

$$h(x, y, z) = \begin{cases} x & \text{if } x > y \\ y + 1 & \text{if } x \leq y \end{cases}$$

(c) $f(n) = \min\{x \leq n + 1 \mid x^2 > n\} \dot{-} 1$.

12.17. From Lemma 12.1, we know that the function f_1 defined by

$$f_1(x, k) = \sum_{i=0}^{k} g(x, i)$$

is primitive recursive. Since $f(x) = f_1(x, x)$, f is also.

12.18. This follows from the formula

$$HighestPrime(k) = \max\{y \leq k \mid Exponent(y, k) > 0\}$$

together with the next exercise.

12.19. (a) We have $m^P(X, 0) = 0$ and

$$m^P(X, k + 1) = \begin{cases} k + 1 & \text{if } P(X, k + 1) \\ m^P(X, k) & \text{if } \neg P(X, k + 1) \end{cases}$$

It follows that m^P is primitive recursive.

(b) $m^P(X, k)$ can be expressed as

$$\begin{aligned} & \min\{y \leq k \mid \text{ for every } z \text{ with } y < z \leq k, \neg P(X, z)\} \\ = \ & \min\{y \leq k \mid \text{ for every } z \leq k, z \leq y \vee \neg P(X, z)\} \end{aligned}$$

Thus m^P can be expressed in terms of a bounded minimalization.

12.20. If $H(x, y)$ is the predicate described in the chapter,

$$H(x, y) = \text{ there exists } y \text{ so that } T_u \text{ halts after exactly } y \text{ moves on input } s_x$$

then we may let $N(x, y)$ be the negation of $H(x, y)$. The unbounded universal quantification of N is the 1-place predicate

$$\text{for every } y, T_u \text{ fails to halt after } y \text{ moves on input } s_x$$

and this is not computable for the same reason that $Halts(x)$ is not.

12.21. We can take $f(X, y)$ to be $1 - \chi_P(X, y)$.

12.23. The bounded operations are special cases of the unbounded. For example, suppose P is an $(n+1)$-place predicate, and define the $(n+2)$-place predicate P_1 by

$$P_1(X, y, z) = (P(X, z) \text{ or } z = y + 1)$$

Then the unbounded minimalization of P_1 is the partial function M_{P_1} of $n+1$ variables defined by

$$M_{P_1}(X, y) = \min\{z \mid P_1(X, y, z)\}$$

If there is a z satisfying $0 \leq z \leq y$ and $P(X, z)$, then $M_{P_1}(X, y)$ is the smallest such z, and otherwise $M_{P_1}(X, y)$ is $y + 1$. In either case, $M_{P_1}(X, y) = m_P(X, y)$.

12.24. If there were a solution to this problem, then in particular there would be a solution to the problem Given a TM computing some partial function, does it halt on every input? Now we know that the (apparently) more general problem Given an *arbitrary* TM, does it accept every input? is unsolvable. But it follows from this that the more restricted problem is also unsolvable, since if T is any TM, there is another TM T' that accepts exactly the same strings as T and computes some partial function. (T' can be constructed as follows: first modify T so that instead of writing Δ it writes a different symbol, say Δ'; then let $T' = TT_1$, where T_1 erases the tape and halts with the tape head on square 0. T' halts on input x if and only if T does, and T' computes the constant partial function whose value is the null string.) Therefore, the given problem is unsolvable.

12.25. We can use Theorem 12.9 with $n = 0$, provided we can come up with an appropriate function $h : \mathcal{N}^2 \to \mathcal{N}$. We are looking for a formula of the form

$$f(k+1) = h(k, gn(f(0), f(1), \ldots, f(k)))$$

where h is primitive recursive. For simplicity we ignore the fact that $h(p, q)$ will have to be defined for $p = 0$ separately (since $f(1)$ is not defined using the recursive formula).

The recursive formula for f can be written $f(k+1) = 1 + f(\lfloor\sqrt{k+1}\rfloor)$. The point is that $\lfloor\sqrt{k+1}\rfloor$ is one of the numbers $f(0), f(1), \ldots, f(k)$. So intuitively, the function h is supposed to produce one of the $k+1$ numbers, the Gödel number of which is the second parameter. Remembering that $f(i) = Exponent(i, gn(f(0), \ldots, f(k)))$, let us write

$$h(p, q) = 1 + \sum_{i=0}^{p} c_i(p) Exponent(i, q) = 1 + \sum_{i=0}^{p} d_i(p, q)$$

where we want $c_i(p)$ to be 0 for all values of i except one (the i for which $\lfloor\sqrt{p+1}\rfloor = i$) and to be 1 for that i.

It is not hard to see that each c_i can be defined by considering two cases, each of which is described by a primitive recursive predicate. It follows from Exercise 12.17 (actually, the

generalization of this exercise to functions of two variables) that $h(p, q)$ will be primitive recursive, since each of the functions d_i is.

12.26. Let $g(x, y) = |f(y) - x|$. Then $f^{-1}(x) = M_g(x) = \mu y[g(x, y) = 0]$.

12.27. As in the chapter, we denote the symbols of the alphabet by $a_1, \ldots a_s$. Then an integer x is the Gödel number of a string of length m if and only if

$$x = 2^{i_0}3^{i_1} \ldots PrNo(m-1)^{i_{m-1}}$$

where each of the exponents i_j satisfies $1 \leq i_j \leq s$. Therefore, $Isgn(x)$ if and only if there is an integer m so that for every i satisfying $0 \leq i \leq m \dot{-} 1$, $1 \leq Exponent(i, x) \leq s$, and for every $i \geq m$, $Exponent(i, x) = 0$. This predicate is the existential quantification of another predicate, and the other predicate involves two universal quantifications. Since $Exponent(i, x) = 0$ for every $i \geq x$, we may express all these quantifications as bounded ones. $Isgn(x)$ is therefore equivalent to this statement: there exists $m \leq x$ so that for every $i \leq m \dot{-} 1$, $1 \leq Exponent(i, x) \leq s$ and for every $i \leq x$, if $i \geq m$ then $Exponent(i, x) = 0$. It follows that $Isgn$ is a primitive recursive predicate.

12.28. (a) $f : \Sigma^* \to \mathcal{N}$ is primitive recursive if the corresponding function $\tau_f : \mathcal{N} \to \mathcal{N}$ is, where τ_f is defined by

$$\tau_f(n) = f(gn'(n))$$

(gn' being the "left inverse" of gn discussed in Section 13.6). The definition for μ-recursive is similar.

(b) If $f(x) = |x|$, then

$$\tau_f(n) = |gn'(n)| = \begin{cases} 0 & \text{if NOT } Isgn(n) \\ HighestPrime(n) + 1 & \text{if } Isgn(n) \end{cases}$$

It is clear from this formula that τ_f, and thus f, is primitive recursive.

12.29. Let g be a computable total function. Then $h = g + 1$ is also. Let T_h be a TM with tape alphabet $\{0, 1\}$ computing h, with k states. For any n, $h(n)$ is the number of 1's left on the tape by T_h if it begins with input 1^n; therefore, $h(n)$ is no larger than the maximum number of 1's left on the tape by any TM having k states and tape alphabet $\{0, 1\}$, if it starts with input 1^n and eventually halts. Clearly, if $n \geq k$, this number is no larger than the maximum number of 1's left on the tape by any TM having n states and tape alphabet $\{0, 1\}$, if it starts with input 1^n and eventually halts. This last number is the definition of $f(n)$, from which it follows that $g(n) < h(n) \leq f(n)$ for every $n \geq k$.

12.31. (a) f can be obtained by primitive recursion from the two functions g and h, defined as follows.

$$g(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x > 0 \end{cases}$$

$$h(x,y,z) = \begin{cases} s(z) & \text{if } Mod(x,k+1) = 0 \\ z & \text{if } Mod(x,k+1) \neq 0 \end{cases}$$

Prime can now be defined in terms of f:

$$Prime(n) = (n \geq 2 \wedge f(n, n\dot{-}1) = 1)$$

12.32. (b) We begin with the observation that the last digit in the decimal representation of an integer n is $Mod(n, 10)$. It is therefore sufficient to show that the function g is primitive recursive, where $g(0) = 1$, $g(1) = 14$, $g(2) = 141$, and so on. $g(i)$ is the greatest integer less than or equal to the real number $10^i\sqrt{2}$. Equivalently, $g(i)$ is the largest integer whose square is less than or equal to $(10^i * \sqrt{2})^2 = 2 * 10^{2i}$. This means that $g(i)+1$ is the smallest integer whose square is greater than $2 * 10^{2i}$. Therefore, if we apply the minimalization operator to the primitive recursive predicate P defined by $P(x,y) = (y^2 > 2 * 10^{2*x})$, $g(x)$ is obtained from subtracting 1 from the result. The only remaining problem is to show how a *bounded* minimalization can be used in this last step. We can do this by specifying a value of y for which y^2 is certain to be greater than $2 * 10^{2*x}$. Clearly, $y = 2 * 10^x$ is such a value, since $(2 * 10^x)^2 = 4 * 10^{2x} > 2 * 10^{2x}$. Therefore, $f(x) = m_P(x, 2 * 10^x)\dot{-}1$.

12.33. We give the proof for the product; the proof for the sum is simpler. First we recall that the empty product (corresponding to the case when $l(k) > m(k)$) is taken to be 1. Assuming that $l(k) \leq m(k)$, we could write

$$f_1(X,k) = Div(\prod_{i=0}^{m(k)} g(X,i), \prod_{i=0}^{l(k)\dot{-}1} g(X,i))$$

—except for two potential problems: the first is the possibility of dividing by 0 (although our formula is defined in that case, it doesn't produce the correct value), and the second is that $l(k)$ may be 0. To get around the first problem, we define

$$g'(X,i) = \begin{cases} g(X,i) & \text{if } g(X,i) \neq 0 \\ 1 & \text{if } g(X,i) = 0 \end{cases}$$

It can then be verified easily that

$$f_1(X,k) = \begin{cases} 1 & \text{if } l(k) > m(k) \\ 0 & \text{in case II} \\ \prod_{i=0}^{m(k)} g(X,i) & \text{in case III} \\ Div(\prod_{i=0}^{m(k)} g'(X,i), \prod_{i=0}^{l(k)\dot{-}1} g'(X,i)) & \text{otherwise} \end{cases}$$

where case II is when there exists i with $l(k) \leq i \leq m(k)$ and $g(X,i) = 0$, and case III is when this condition fails and $l(k) = 0$. Moreover, this formula makes it clear that f_1 is

primitive recursive. (Note that the predicate in case II is primitive recursive, since it can be expressed in terms of bounded minimalization.)

12.34. A valid μ-recursive derivation involves the application of unbounded minimalization only to total functions. Since it may not be possible to determine whether a given function is total (see Exercise 12.24), it may not be possible to determine whether a string purporting to be a μ-recursive derivation is in fact valid.

12.35. (b) If $f(x) = ax$, then the function ρ_f takes the number

$$2^{i_0}3^{i_1}\ldots PrNo(m)^{i_m}$$

to the number

$$2^1 3^{i_0} 5^{i_1}\ldots PrNo(m+1)^{i_m}$$

In other words, ρ_f is defined by the formula

$$\rho_f(n) = 2 * \prod_{i=0}^{r(n)} g(n,i)$$

where $r(n) = HighestPrime(n)$ (see the proof of Lemma 12.4) and $g(n,i) = PrNo(i+1)^{Exponent(i,n)}$. This is almost, but not quite, in the form of Lemma 12.1. According to that result, the function

$$h(n,y) = \prod_{i=0}^{y} g(n,i)$$

is primitive recursive. But $\rho_f(n) = 2 * h(n, r(n))$, from which it follows that ρ_f is also.

(c) The argument is similar to that in part (b). In this case we may write

$$\rho_f(n) = \prod_{i=0}^{r(n)} g(n,i)$$

where $r(n) = HighestPrime(n)$ and

$$g(n,i) = (PrNo(i))^{Exponent(HighestPrime(n) \dot{-} i, n)}$$

12.36. (a) $f : (\Sigma^*)^n \to \Sigma^*$ is primitive recursive (resp. recursive) if the corresponding function $\pi_f : \mathcal{N}^n \to \mathcal{N}$ is, where π_f is defined by

$$\pi_f(n_1,\ldots,n_k) = gn(f(gn'(n_1),\ldots,gn'(n_k)))$$

(b) If f is the concatenation function, then $\pi_f : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ is defined by

$$\pi_f(m,n) = gn(gn'(m)gn'(n))$$

$$= m * \prod_{i=l(m,n)}^{u(m,n)} h(m,n,i)$$

where

$$\begin{aligned} l(m,n) &= HighestPrime(m) + 1 \\ u(m,n) &= HighestPrime(m) + HighestPrime(n) + 1 \\ h(m,n,i) &= PrNo(i)^{Exponent(i \dot{-} HighestPrime(m) \dot{-} 1, n)} \end{aligned}$$

It follows (see the solution to Exercise 12.35) that π_f is primitive recursive.

Chapter 13

Measuring and Classifying Complexity

13.1. (a) Suppose $f(n) \leq C_0 h(n)$ for $n \geq n_0$ and $g(n) \leq C_1 k(n)$ for $n \geq n_1$. Then $f(n) + g(n) \leq C(h(n) + k(n))$ for $n \geq N$, where $C = \max(C_0, C_1)$ and $N = \max(n_0, n_1)$; also, for the same N, $f(n) * g(n) \leq D(h(n) * k(n))$ for $n \geq N$, where $D = C_0 * C_1$.

(c) This statement follows from the inequalities $\max(f, g) \leq f + g \leq 2\max(f, g)$.

13.2.

	f_1	f_2	f_3	f_4
f_1		yes	yes	yes
f_2	yes		yes	yes
f_3	no	no		no
f_4	no	no	yes	

For example, $f_3 \neq O(f_1)$: if $n > 1$, $f_3(n)/f_1(n) = n/(2\log n)$, and this ratio is unbounded. $f_4 = O(f_3)$: if $n > 1$,

$$f_4(n)/f_3(n) = \begin{cases} 2\log n/n & \text{if } n \text{ is even} \\ \log^2 n/n & \text{if } n \text{ is odd} \end{cases}$$

and in either case the ratio is bounded (in fact, approaches 0).

13.3. If $f(n) \leq Cp(n)$ for every $n \geq N_0$, then $f(n) \leq Cp(n) + M$ for every n, where $M = \max\{f(n) | n < N_0\}$.

13.4. It is clear that $n! < n^n$ if $n > 1$, and since $n^n = e^{n\log n}$ and $2^{2^n} = e^{2^n \log 2}$, $n^n < 2^{2^n}$. Thus it is sufficient to show that the growth rate of $n!$ is greater than exponential. But this is reasonably clear: if $n > a^2$, for example,

$$\begin{aligned} n!/a^n &= \{(1/a)(2/a)\ldots(a/a)\} \\ &\quad *\{((a+1)/a)((a+2)/a)\ldots((a^2-a)/a)\} \\ &\quad *\{((a^2-a+1)/a)((a^2-a+2)/a)\ldots(a^2/a)\} \\ &\quad *\{((a^2+1)/a)\ldots(n/a)\} \\ &= A * B * C * D \end{aligned}$$

The product $A * C$ is at least 1, $B > 1$, and it is obvious that D gets arbitrarily large as n gets larger.

13.5. (a) $2^{\sqrt{n}} = e^{\sqrt{n}\log 2}$ and $n^{\log n} = e^{\log^2 n}$. Since $\sqrt{n}\log 2 > \log^2 n$ for all large n, $2^{\sqrt{n}} > n^{\log n}$. It is sufficient to show that the growth rate of $n^{\log n}$ is greater than polynomial and that of $2^{\sqrt{n}}$ is less than exponential. The first is clear, since $a^n = e^{a\log n}$ and $\log^2 n$ grows faster than $a\log n$; the second is just as easy, and follows from comparing $\sqrt{n}\log 2$

to $n \log a$ for any fixed a.

(b) The function $\sqrt{n}$ has a larger growth rate than $\log^2(n)$. It follows that $2^{\sqrt{n}}$ has a larger growth rate than $n^{\log n}$.

13.6. $(\log n)^{\log n} = e^{\log n \log\log n}$. Since $\log\log n$ eventually gets large, this function has a larger growth rate than $e^{k \log n}$, for any k, and therefore has a larger growth rate than any polynomial. However, since $\log n \log\log n$ has a smaller grown rate than cn, for any positive constant c, this function has a smaller growth rate than any exponential function.

13.7. We wish to show that for any fixed k, and any $a > 1$, $n^k = O(a^n)$. (From this fact the theorem will follow, just as it did in the proof given in the chapter.)

Choose N_0 so that $((N_0+1)/N_0)^k \leq a$. Then if $n = N_0 + m$,

$$\begin{aligned} n^k &= (N_0+m)^k \\ &= N_0^k((N_0+1)/N_0)^k \ldots ((N_0+m)/(N_0+m-1))^k \\ &\leq N_0^k a^m = C a^{N_0+m} = C a^n \end{aligned}$$

where C is the constant N_0^k / a^{N_0}.

13.8. For simplicity, suppose the input string is $a^n = a^{2k+1}$. (Note that for input strings of odd length, the number of moves depends only on the length, not on the symbols themselves.) It is easy to check that $4k+3$ moves are required to transform the tape contents $\Delta \underline{a} a \ldots a$ to tape contents $\Delta A \underline{a} a \ldots a A$; $4(k-1)+3$ moves to transform this to $\Delta A A \underline{a} a \ldots a A A$; $\ldots$; and $4(1)+3$ to go to $\Delta A \ldots A \underline{a} A \ldots A$. Adding the single move at the beginning and the two additional moves before the machine crashes, we obtain

$$\begin{aligned} 3 + 4\sum_{i=1}^{k} i + 3k &= 3 + 4k(k+1)/2 + 3k \\ &= 3 + 4((n-1)/2)((n+1)/2)/2 + 3(n-1)/2 \\ &= 3 + (n^2-1)/2 + 3n/2 - 3/2 \\ &= (n^2+3n+2)/2 \end{aligned}$$

13.9(a) Suppose the input is $a^n = a^{2k}$. $4k+1$ moves are needed to go from $\Delta \underline{a} a^{2k-1}$ to $\Delta\Delta \underline{a} a^{2k-3}$; $4(k-1)+1$ to go to $\Delta\Delta\Delta \underline{a} a^{2k-5}$; $\ldots$; and $4(1)+1$ to go to $\Delta^{k+1} \underline{\Delta}$. Add one move at the beginning and at the end, and the result is

$$\begin{aligned} 2 + \sum_{i=1}^{k}(4i+1) &= 2 + 4k(k+1)/2 + k \\ &= 2k^2 + 3k + 2 \\ &= (n^2+3n+4)/2 \end{aligned}$$

Moreover, it can be checked that the same formula holds when n is odd.

(b) Suppose that the input is a^n. $2n+3$ moves are required to get from tape contents $\Delta\underline{a}a^{n-1}$ to tape contents $\Delta A\underline{a}a^{n-2}\Delta a$. Each additional such pass requires the same number of moves. The next-to-last ends with $\Delta A^{n-1}\underline{a}\Delta a^{n-1}$, and the last ends with $\Delta A^n\underline{\Delta}a^n$. $n+2$ moves are then required to finish up. Adding the one move at the beginning, we obtain

$$n(2n+3)+n+3 = 2n^2+4n+3$$

13.11. Saying that the problem is solvable is the same as saying that the language L of strings that represent instances of the problem for which the answer is yes is recursive. Let T be any TM computing χ_L, and let τ_T be its time complexity. Let us now consider a different encoding of instances. Take a symbol \$ not used in the original encodings; if x was the original encoding of an instance, then the new encoding will be $x\$^m$, where $m = \tau_T(|x|)$. Let L_1 be the new language corresponding to L. Then we may modify T to form a new TM T_1 computing χ_{L_1} as follows: T_1 simply erases all the \$'s and executes T. The time complexity of T_1 satisfies the inequality

$$\begin{aligned}\tau_{T_1}(|x|+m) &\leq 2(|x|+m+1)+\tau_T(|x|)\\ &\leq 2(|x|+m+1)+m\\ &\leq 3(|x|+m)+2\end{aligned}$$

which implies that the time complexity of T_1 is at most linear.

13.14. For a particular f, Theorem 13.3 says that the property of being in $Time(f)$ is a nontrivial language property. Therefore, by Rice's theorem, the problem is unsolvable.

13.18. Let T_1 and T_2 be TMs recognizing L_1 and L_2 in times f_1 and f_2, respectively. If we consider a 2-tape TM that executes T_1 and T_2 simultaneously, one on each tape, then aside from the time required to copy the input onto tape 2, the time required to recognize either the union or the intersection is no more than the maximum of the two functions. We can take both g and h to be the function $2n+2+\max(f_1(n), f_2(n))$.

13.19. In both cases, we can construct a TM so that at each step, every work tape contains an integer, in binary notation, no larger than the length of the input. (This is where the log term comes from: the number of digits required to represent n in binary.)

For palindromes, we could use three work tapes and proceed as follows. Begin by placing the integer 1 on tape 1 and the integer n on tape 2. (On the kth pass, these tapes have the numbers k and $n-k$, respectively.) On each pass, start at the position on the input tape specified by the first work tape, use the third tape to count from that point to the number on the second tape, and check that those two symbols of the input are the same. Then increment the number on the first tape, decrement the number on the second tape, erase the third tape, and start the next pass.

For balanced strings of parentheses, we can get by with one work tape. Process the input left-to-right, keeping track on the work tape of the difference between the numbers of right and left parentheses. This number is initially 0, is incremented each time a left

parenthesis is read and decremented each time a right parenthesis is read. If the number never goes below 0 and ends up at 0, the string is balanced.

13.20. The statement $f = o(g)$ does not follow. If we allow the functions to be nondecreasing, instead of strictly increasing, there is a straightforward way to construct a counterexample. Let $g(n) = n$ for every n. Let $f(0) = 0$, $f(1) = f(2)) = 1$, $f(3) = \ldots = f(9) = 3$, $f(10) = \ldots f(40) = 10$, $f(41) = \ldots = f(205) = 41$, etc. The pattern is that on the first interval where f is constant and nonzero, the ratio f/g decreases from 1 to 1/2, on the second such interval f/g decreases from 1 to 1/3, on the next such interval f/g decreases from 1 to 1/4, etc. Then $f = O(g)$, since $f(n) \leq g(n)$ for every n; $g \neq O(f)$, because for each $n > 0$ there is an i for which $g(i) = nf(i)$; and $f \neq o(g)$, because there are infinitely many n's for which $f(n) = g(n)$. Constructing a counterexample is a little harder if f and g are required to be increasing, since f is not allowed to be constant on a long interval. We can use the same general idea if we take $g(n) = 2^n$. $f(n)$ is still defined to be $g(n)$ at infinitely many different n's; instead of being constant on an interval, f will increase by 1 at each step; and the ith interval will be long enough so that the ratio f/g will decrease from 1 to some number less than or equal to $1/i$.

13.21. No. Consider the functions f and g defined as follows. $f(0) = g(0) = 1$; $f(1) = f(0) + 1$; $g(1) = 2f(1)$; $g(2) = g(1) + 1$; $f(2) = 3g(2)$; $f(3) = f(2) + 1$; $g(3) = 4f(3)$; ...

Then for any $n > 1$ there are values of j and k so that $f(j)/g(j) > n$ and $g(k)/f(k) > n$. It follows that $f \neq O(g)$ and $g \neq O(f)$.

13.22. One approach is to have the TM copy the input onto the second tape and return the tape heads to the beginning; have tape head 1 move two symbols for every one tape head 2 moves (rejecting if the length is not even), so that tape head 2 is now in the middle of the input; move tape head 1 back to the beginning; then compare the first half of tape 1 to the second half of tape 2, symbol by symbol.

13.23. For simplicity, we consider the case in which there is only one string x for which T makes more than $f(|x|)$ moves on input x. The general case follows by induction. Let $|x| = n$. We would like to modify T to obtain a TM T_1 so that T_1 recognizes L and $\tau_{T_1} \leq f$. One approach would be to have T_1 simply behave like a finite automaton reading the input, until it determines whether the input string is x; if it is, it erases the tape and leaves the appropriate output (1 if $x \in L$, 0 otherwise), and if it is not, it returns to the beginning of the tape and proceeds to execute T. The problem with this is that in the second case, it might require $f(m) + 2n + 2$ moves to process a string of length m, whereas we want the maximum to be $f(m)$—or more precisely, $\max(f(m), 2m + 2)$. We can use the same idea, though, just being more careful.

By using a larger alphabet, we can have T_1 start moving its tape head to the right, replacing the symbol σ on tape square i by a new symbol (σ, i), for $0 \leq i \leq n$. If it reaches the end of the input by this time (that is, if the input string is *any one* of the strings of length n or less), then T_1 erases the tape, leaves the appropriate output, and halts. Otherwise, the input string has length at least $n + 1$. In this case, there are exactly

s^n possibilities for the prefix of length n, where s is the size of the input alphabet. Let us consider a specific prefix α of length n. For an input string starting with prefix α, there are two things T might do: crash before ever moving its tape head to square $n+1$; or eventually move its tape head to square $n+1$, having first replaced the symbol on square i by σ_i for each i with $0 \leq i \leq n$. If T crashes before it reaches square $n+1$, T_1 then proceeds to erase the tape and print the output 0. Otherwise, T_1 proceeds with exactly the same computation T would make, starting just after the move on which T reaches square $n+1$ for the first time, *except* that in the subsequent computation, T_1 treats the symbol (σ, i) exactly the same as it would the symbol σ_i when it reads it. In other words, T_1 has effectively bypassed the preliminary steps of T, during which T has not yet reached square $n+1$, replacing these steps by the first $n+1$ moves of T_1. It has not lost any information by doing this, because of the way it has marked the first $n+1$ squares of the tape. The result is that T_1 is able to complete the computation in no more steps than T would have required. In particular, $\tau_{T_1}(m) \leq \max(f(m), 2m+2)$ for each m.

13.25. See the solution to Exercise 9.44. Although that exercise involves a 2-tape TM, the same argument applies to an arbitrary multitape TM.

13.26. As in the proof of Theorem 14.3, let

$$L = \{w \mid w = e(T'), \text{ where } T' \text{ crashes on } w \text{ within } f(|w|) \text{ moves}\}$$

Let T be any TM accepting L. Then there are infinitely other TMs that also accept L and have the same time complexity as T. (Simply modify T by adding any number of moves that are never actually executed.) Let T_1 be any such TM, and let $w = e(T_1)$. If T halts on w, then $w \in L$, since T accepts L. This means that T_1 crashes on input w. But this is a contradiction, since T_1 halts on the same strings as T. On the other hand, if T crashes on w within $f(|w|)$ moves, then the same is true of T_1, and thus $w = e(T_1) \in L$ by definition of L. This is also a contradiction, since T accepts L. The only possibility left is that T crashes on w and that it requires more than $f(|w|)$ moves for this to happen. Since T_1 can be one of infinitely many different TMs, we may conclude that $\tau_T(n) > f(n)$ for infinitely many n.

13.27. Let T be a TM computing f. Then let T_1 be a TM that, on any input x, changes it to $1^{|x|}$ and executes T on that string. Then the function τ_{T_1} is a step-counting function, since T_1 makes the same number of moves for every input string of a given length. Moreover, $\tau_{T_1}(n) > f(n)$, because T_1 requires at least $f(n)$ moves to write the output string $1^{f(n)}$.

Chapter 14

Tractable and Intractable Problems

14.2. Suppose that the input string is of length n, and we wish to find all occurrences of a substring $\alpha = \$1^k\$$ of the input. Comparing a single character of α to another character of the input might take up to $2n$ moves, but no character in the input needs to be compared to a character of α more than once, so all occurrences of α can be found within $2n^2$ moves.

14.3. (c) The statement $L \in NP$ means that there exist a nondeterministic TM T and a polynomial p so that T accepts L and $\tau_T \leq p$. This implies that for any string of length n in L, there is a sequence of no more than $p(n)$ moves that causes T to accept the string; and for a string of length n not in L, there is a sequence of $p(n)$ or fewer moves causing T to reject the string. If we construct another TM T' that halts whenever T crashes, and vice versa, then $\tau_{T'}$ is still $\leq p$. However, T' accepts the strings for which there is at least one sequence of moves of T causing T to crash, and these are not in general the same as the strings not in L. There is no other obvious way to modify T so that the modified TM accepts L' in polynomial time.

14.4. (a) Any function reducing L_1 to L_2 also reduces L_1' to L_2'.

(b) Suppose L is NP-complete and $L' \in NP$. Let L_1 be any language in NP. Since $L' \in NP$, we can show $L_1' \in NP$ by showing that $L_1' \leq L'$. But since L is NP-complete, $L_1 \leq L$. Therefore, the result follows from part (a).

14.5. By assumption, there are two strings x_1 and x_2 so that $x_1 \in L_2$ and $x_2 \notin L_2$. Define the function $f : \Sigma^* \to \Sigma^*$ by the formula

$$f(x) = \begin{cases} x_1 & \text{if } x \in L_1 \\ x_2 & \text{otherwise} \end{cases}$$

Then for any x, $x \in L_1$ if and only if $f(x) \in L_2$, and the assumption that $L_1 \in P$ implies that f is computable in polynomial time.

14.6. (a) To say that "P_2 is hard" presumably means that there are instances of P_2 that are hard (not necessarily that all instances are hard). It may be that all the instances of P_1 are easy, so that P_1 is easy.

(b) We show that *3-Satisfiable* $\leq_p$ *Satisfiable*. Define $f : \Sigma^* \to \Sigma^*$ by $f(x) = x$ if x is the encoding of a CNF expression involving three or fewer conjuncts, and $f(x) = \Lambda$ otherwise. This function reduces *3-Satisfiable* to *Satisfiable*, and it is not hard to see that it is polynomial-time computable.

(c) *3-Satisfiable* is the intersection of two languages, *Satisfiable* and the language *3-CNF* of all encodings of CNF expressions in which there are three or fewer conjuncts. The second language is in P. One generalization is therefore the following: if $L, L_1 \subseteq \Sigma^*$, $L \neq \Sigma^*$, and $L_1 \in P$, then $L \cap L_1 \leq_p L$. To show this, use the reduction f, where $f(x)$ is x if $x \in L_1$ and x_0 otherwise (where x_0 is a fixed string not in L). Then if $x \in L \cap L_1$, $f(x) \in L$; if

$x \notin L$, then $f(x) \notin L$ (either because $f(x) = x$ or because $f(x) = x_0$); and if $x \notin L_1$, then $f(x) = x_0 \notin L$.

14.9. We give the solution for $k = 4$, and the more general case is similar. Given an instance

$$\bigwedge_{i=1}^{n} (x_{i,1} \vee x_{i,2} \vee x_{i,3})$$

of the 3-satisfiability problem, we may construct the instance

$$\bigwedge_{i=1}^{n} (a \vee x_{i,1} \vee x_{i,2} \vee x_{i,3}) \wedge \bigwedge_{i=1}^{n} (\overline{a} \vee x_{i,1} \vee x_{i,2} \vee x_{i,3})$$

of the 4-satisfiability problem (where a is a variable not appearing in the original expression), which is satisfiable if and only if the original is. This clearly implies that the 3-satisfiability problem is polynomial-time reducible to the 4-satisfiability problem. Since the former is NP-complete, so is the latter.

14.10. Consider any possible conjunct, say $x \wedge y \wedge z$. If this one is missing from the expression, then the expression must be satisfied by the assignment in which all three variables are false, because every other possible conjunct contains the negation of one variable. Therefore, the smallest unsatisfiable expression satisfying these conditions is the one with all 8 possible conjuncts.

14.11. (a) The entire expression is satisfiable if and only if at least one of the disjuncts is satisfiable, and this is true if and only if at least one of the disjuncts doesn't contain both a variable and its negation. Testing this condition can obviously be done in polynomial time.

(b) A CNF expression is a tautology if and only if each conjunct contains some variable and its negation, and this can be checked in polynomial time.

14.14. The language L is in NP; a nondeterministic procedure to accept L is to take the input string and, provided it is of the form $e(T)\$e(x)\$1^n$, choose a sequence of n moves of T on input x, accepting the input if the sequence causes x to be accepted by T.

To show that L is NP-hard, let L_1 be any language in NP, and let T be a nondeterministic TM that accepts L_1 and has nondeterministic time complexity bounded by some polynomial p. Then consider the function $f : \Sigma^* \to \{0,1\}^*$ defined by $f(x) = e(T)\$e(x)\$1^{p(|x|)}$. Any string x is in L_1 if and only if $f(x) \in L$, and it is easy to check that f is polynomial-time computable.

14.15. Consider the following procedure for coloring the vertices in each "connected component" of the graph. Choose one and color it white. Color all the vertices adjacent to it black. For each of those, color all the vertices adjacent to it (and not yet colored) white. Continue this until there are no colored vertices with uncolored adjacent vertices. Repeat for each component. This can be carried out in polynomial time, and it can be determined

in polynomial time whether the resulting assignment of colors is in fact a 2-coloring of the graph. Furthermore, the graph can be 2-colored if and only if this procedure produces a 2-coloring.

14.17. A TM can perform these two steps on the input string x in order to recognize $f^{-1}(A)$: first calculate $f(x)$, and then test $f(x)$ for membership in A. Since $A \in P$, and since the length of the string $f(x)$ is no more than a polynomial function of $|x|$, both steps can be performed in time no more than a polynomial function of $|x|$.

14.18. We already have a reduction from **CSG** to **VC**; exactly the same function defines a reduction in the other direction. If $G = (V, E)$, (G, k) is an instance of **VC**, and there is a vertex cover of G having the k vertices $\nu_1, \ldots, \nu_k$, then the remaining $|V| - k$ vertices determine a complete subgraph in the complement of G, since no two of them can be joined by an edge in V. Conversely, if there is a complete subgraph on $|V| - k$ vertices in the complement of G, then the remaining k vertices form a vertex cover for G.

Now consider the function that sends a pair (G, k) to the pair (G', k), where G' is the complement of G. This function is both a reduction from **CSG** to **IS** and a reduction from **IS** to **CSG**. The reason is that a set of vertices determines a complete subgraph in one graph if and only if the same set is independent in the complement.

Finally, we can get reductions from **IS** to **VC** and back by considering the composition of the two functions above. If (G, k) is an instance of **IS** , consider the instance $(G, |V| - k)$ of **VC**. Vertices $\nu_1, \ldots, \nu_k$ form an independent set in G if and only if the remaining $|V| - k$ vertices form a vertex cover of G. This function is therefore a reduction from either of the two problems to the other.

14.21. (a) The function f defined by $f(x) = x0$ reduces L_1 in polynomial time to $L_1 \oplus L_2$, because if $x \in L_1$, $f(x) \in L_1\{0\}$, and if $x \notin L_1$, $x0 \notin L_1\{0\} \cup L_2\{1\}$. Similarly, $g(x) = x1$ reduces L_1 to $L_1 \oplus L_2$.

(b) The problem also requires the hypothesis that $L \neq \Sigma^*$; let $z \notin L$. Suppose that f_1 and f_2 are polynomial-time reductions from L_1 to L and from L_2 to L, respectively. Define $f : \Sigma^* \to \Sigma^*$ by $f(\Lambda) = z$, $f(y0) = f_1(y)$, and $f(y1) = f_2(y)$ (for any $y \in \Sigma^*$). Then if $x \in L_1 \oplus L_2$, there are two cases: if $x = y0$ for some $y \in L_1$, then $f(x) = f_1(y) \in L$; and if $x = y1$ for some $y \in L_2$, $f(x) = f_2(y) \in L$. If $x \notin L_1 \oplus L_2$, then there are three cases. If $x = \Lambda$, $f(x) \notin L$ by the assumption on z. If $x = y0$ and $y \notin L_1$, then $f(x) = f_1(y) \notin L$; and if $x = y1$ and $y \notin L_2$, then $f(x) = f_2(y) \notin L$. Since f_1 and f_2 are both polynomial-time computable, f is also.

14.22. We shall describe a polynomial-time algorithm for determining whether a CNF expression in which each conjunct has exactly two literals is satisfiable. (It is taken from Salomaa, *Computation and Automata*, Cambridge University Press, 1985, pp. 162-163.)

1. Delete any conjuncts of the form $(x \vee \overline{x})$.

2. Replace any conjuncts of the form $(x \vee x)$ by the single term x (where x is either an a or an $\overline{a}$).

3. If the resulting expression contains a conjunct that is a single x (either an a or an $\overline{a}$), then

   (a) If $\overline{x}$ appears by itself as another conjunct, then stop, and conclude that the expression is unsatisfiable.

   (b) Remove all conjuncts containing x (either by itself or with another variable).

   (c) Replace any conjunct of the form $(\overline{x} \vee y)$ by y.

   Iterate step 3 until there are no remaining conjuncts of the form x.

4. At this point, all conjuncts look like $(x \vee y)$, where $x \neq y$ and $x \neq \overline{y}$. If some conjunct contains x and no other conjunct contains $\overline{x}$, then delete all conjuncts containing x. Iterate this step until, for every literal x in the expression, both x and $\overline{x}$ appear in the expression. If there are no literals remaining, stop and conclude that the expression is satisfiable.

5. Choose an arbitrary literal x in the expression, and let the conjuncts containing x and $\overline{x}$ be

   $$(x \vee y_1), \ldots, (x \vee y_m), (\overline{x} \vee z_1), \ldots, (\overline{x} \vee z_n)$$

   Let the remaining portion of the expression be α. Then replace the expression by

   $$\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n} (y_i \vee z_j) \wedge \alpha$$

   (This is correct because the expression is equivalent to

   $$((\bigwedge_{i=1}^{m} y_i) \vee (\bigwedge_{j=1}^{n} z_j)) \wedge \alpha$$

   and the given expression is simply the CNF version of this.)

6. Go to Step 1.

It is straightforward to check that each step requires time that is only polynomial in the length of the current expression, and that the expression resulting from each step has length that is only polynomial in the length of the previous expression. Since each time the algorithm returns to step 1, the number of distinct variables has been reduced by at least 1, we may conclude that the entire algorithm takes only polynomial time.

14.23. The part that is least obvious is that P is closed under Kleene *. Suppose $L \in P$. We describe an algorithm for recognizing L^* in general terms, and it is not difficult to see that a TM can execute it in polynomial time. (There are considerably more efficient algorithms, whose runtimes would be polynomials of lower degree.)

Given a string $x = a_1a_2 \ldots a_n$ of length n, and integers i and j with $1 \le i \le j \le n+1$, denote by $x[i,j]$ the substring $a_ia_{i+1} \ldots a_{j-1}$ of length $i-j$. There are $(n+1)(n+2)/2$ ways of choosing i and j (although the substrings $x[i,j]$ are not all distinct). In the algorithm, we keep a table in which every $x[i,j]$ is marked with a 1 if it has been found to be in L^* and 0 otherwise. We initialize the table by marking $x[i,j]$ with a 1 if it is in L, and 0 if not. Since $L \in P$, this initialization can be done in polynomial time. We then begin a sequence of iterations. Each iteration consists of examining every pair $x[i,j]$, $x[j,k]$, where $0 \le i \le j \le k \le n+1$; if $x[i,j]$ and $x[j,k]$ are both marked with 1, then their concatenation $x[i,k]$ is marked with 1. We continue until either we have performed n iterations or we have executed an iteration in which the table did not change. x is in L^* if and only if $x = x[1, n+1]$ is marked with a 1 at the termination of the algorithm.

14.24. Suppose L is *NP*-complete, T recognizes L, and $\tau_T \le f$, where f is subexponential. Let L_1 be any language in *NP*. Since L is *NP*-complete, there is a function r reducing L_1 to L so that r can be computed in time bounded by some polynomial p. We construct a TM T_1 to recognize L_1 as follows: T_1 takes an input string, computes $y = r(x)$, and then tests y for membership in L. The time required for this is no more than $p(|x|) + f(|r(x)|)$. In order to show that this is a subexponential function of $|x|$, it is sufficient to show that the second term is.

We may assume that f is nondecreasing, since the function f_1 defined by $f_1(n) = \max\{f(i) \mid 0 \le i \le n\}$ is nondecreasing, at least as big as f, and still subexponential. Then $f(|r(x)|) \le f(p(|x|))$, since $|r(x)| \le p(|x|)$. Since p is a polynomial, there is a k so that $p(n) \le n^k$ for sufficiently large n. Therefore, $f(p(n)) \le f(n^k)$. Since f is subexponential, then for any $\delta > 0$, $f(n^k) \le C2^{(n^k)^\delta}$ for some C and all sufficiently large n. Let us choose $\delta = \epsilon/k$. It follows that for sufficiently large n, if $|x| = n$,

$$f(|r(x)|) \le f(p(|x|)) \le f(p(n)) \le f(n^k) \le C2^{(n^k)^\delta} \le C2^{n^\epsilon}$$

14.25. We reduce the vertex cover problem to this one as follows. Given G and k, an instance of the vertex cover problem, we may construct G_1 and k_1 as follows: add 3 new vertices v_1, v_2, and v_3 to G; add the three edges joining each pair of these; and add edges to connect v_1 to every vertex of G having odd degree; $k_1 = k + 2$. In G_1, every vertex has even degree. Now the claim is that G has a vertex cover with k vertices if and only if G_1 has a vertex cover with k_1 vertices. One direction is easy: if G has a vertex cover with k vertices, those vertices plus v_1 and v_2 clearly form a vertex cover for G_1. Now suppose there is a vertex cover for G_1 having $k+2$ vertices. Then clearly, the vertices in the vertex cover that are in G form a vertex cover for G. The vertex cover for G_1 must contain at least two of the three vertices v_1, v_2, and v_3, since it has edges joining any two of these. Therefore, there is a vertex cover for G having k vertices. Since it is fairly obvious that this is a polynomial-time reduction, and since the vertex cover problem is *NP*-complete, this problem is also *NP*-complete.

14.28. Let $G = (E, V)$ be a graph and k an integer. We wish to construct an instance

$(S_1, S_2, \ldots, S_m)$ of the exact cover problem corresponding to (G, k). The elements belonging to the subsets $S_1, \ldots, S_m$ will be of two types: elements of V, and pairs of the form (e, i), where $e \in E$ and $1 \leq i \leq k$. Specifically, for each $v \in V$ and each i with $1 \leq i \leq k$, let

$$S_{v,i} = \{v\} \cup \{(e, i) \mid v \text{ is an end point of } e\}$$

In addition, for each pair (e, i), let $T_{e,i} = \{(e, i)\}$. Then the union of all the sets $S_{v,i}$ and $T_{e,i}$ contains all the vertices of the graph, as well as all the possible pairs (e, i).

Suppose on the one hand that there is a k-coloring of G. Let us denote the colors by 1, 2, $\ldots$, k, and let us suppose that each vertex v is colored with the color i_v. Then we can construct an exact cover for our collection of sets as follows. We include every set S_{v,i_v}. The union of these contains all the vertices. We also include all the $T_{e,i}$'s for which the pair (e, i) has not already been included in some S_{v,i_v}. Clearly, these sets form a cover—that is, their union contains all the elements in the original union. To see that it is an exact cover, it is sufficient to check that no two sets S_{v,i_v} and S_{w,i_w} can intersect if $v \neq w$. This is obvious if v and w are not adjacent; if they are adjacent, it follows from the fact that the colors i_v and i_w are different.

On the other hand, suppose that there is an exact cover for the collection of $S_{v,i}$'s and $T_{e,i}$'s. Then for each vertex v, v can appear in only one set in the exact cover, and thus there can be only one i, say i_v, for which $S_{v,i}$ is included. Now we can check that if we color each vertex v with the color i_v, we do in fact get a k-coloring of G. If not, then some edge e joins two vertices v and w with $i_v = i_w = i$. Then, however, both S_{v,i_v} and S_{w,i_w} contain the pair (e, i) which is impossible if the cover is exact.

14.29. Suppose S_1, S_2, $\ldots$, S_k are finite sets, where $\cup_{j=1}^{k} S_j = X = \{x_0, x_1, \ldots, x_{n-1}\}$. We view the S_j's as an instance of the exact cover problem: it is a yes-instance if it is possible to choose some of the S_j's which are pairwise disjoint and still have union X. We wish to construct from the S_j's an instance of the sum-of-subsets problem.

For each j with $1 \leq j \leq k$, define

$$a_j = \sum_{i=0}^{n-1} \epsilon_{ji}(k+1)^i, \text{ where } \epsilon_{ji} = \begin{cases} 1 & \text{if } x_i \in S_j \\ 0 & \text{otherwise} \end{cases}$$

Let the integer M be $\sum_{i=0}^{n-1}(k+1)^i$. This is a yes-instance of the sum-of-subsets problem if it is possible to choose some of the a_j's whose sum is M.

Suppose on the one hand that there is an exact cover for the S_j's, say $\{S_j \mid j \in J\}$. Then for each i, there is exactly one $j \in J$ for which $x_i \in S_j$, so that $\sum_{j \in J} \epsilon_{ji} = 1$. It follows that

$$\sum_{j \in J} a_j = \sum_{j \in J} \sum_{i=0}^{n-1} \epsilon_{ji}(k+1)^i = \sum_{i=0}^{n-1} (\sum_{j \in J} \epsilon_{ji})(k+1)^i = \sum_{i=0}^{n-1} (k+1)^i = M$$

On the other hand, if there is a subset J of $\{1, 2, \ldots, k\}$ for which $\sum_{j \in J} a_j = M$, then rearranging the double sum the way we did above, we obtain

$$\sum_{i=0}^{n-1} b_i(k+1)^i = \sum_{i=0}^{n-1} 1(k+1)^i = M$$

where $b_i = \sum_{j \in J} \epsilon_{ji}$. The set J contains at most k elements, so that $0 \leq b_i \leq k$. Since expansions of an integer to base $k+1$ are unique, it follows that $b_i = 1$ for each i. This means that for each i, there is exactly one $j \in J$ with $\epsilon_{ji} = 1$, and this implies that the S_j's with $j \in J$ form an exact cover for the S_j's.

As usual, it is necessary to check that the instance of the sum-of-subsets problem can be constructed in polynomial time from the instance of the exact cover problem, but this is straightforward.

14.30. Let $\{a_1, a_2, \ldots, a_n\}$ and M represent an instance of the sum-of-subsets problem. We construct an instance b_1, b_2, $\ldots$, b_m of the partition problem by letting $m = n+2$ and letting the b_i's be defined by

$$b_i = a_i \text{ for } 1 \leq i \leq n \qquad b_{n+1} = M+1 \qquad b_{n+2} = \sum_{i=1}^{n} a_i + 1 - M$$

If there is a subset I of $\{1, 2, \ldots, n\}$ so that $\sum_{i \in I} a_i = M$, then we let $J = I \cup \{n+1\}$, and it is easy to check that

$$\sum_{j \in J} b_j = \sum_{j \notin J} b_j$$

On the other hand, if there is a subset J of $\{1, 2, \ldots, n+2\}$ with $\sum_{j \in J} b_j = \sum_{j \notin J} b_j$, then since $b_{n+1} + b_{n+2}$ is bigger than the sum of the remaining b_j's, exactly one of the two numbers $n+1$, $n+2$ is in J. If we suppose that it's $n+1$, then it's easy to check that $\sum_{i \in I} a_i = M$, where $I = J - \{n+1\}$.

14.31. Let a_1, a_2, $\ldots$, a_n represent an instance of the partition problem. We construct an instance of the 0-1 knapsack problem by letting $w_i = p_i = a_i$ for each i, and letting $W = P = (1/2)\sum_{i=1}^{n} a_i$. If there is a subset I of $\{1, 2, \ldots, n\}$ with $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$, then $\sum_{i \in I} w_i = W$ and $\sum_{i \in I} p_i = P$. On the other hand, if there is an I so that $\sum_{i \in I} w_i \leq W$ and $\sum_{i \in I} p_i \geq P$, then both inequalities are in fact equalities, and I determines a solution to the partition problem.