For more information, contact:

BROOKS/COLE PUBLISHING COMPANY
511 Forest Lodge Road
Pacific Grove, CA 93950
USA

International Thomson Publishing Europe
Berkshire House 168-173
High Holborn
London WC1V 7AA
England

Thomas Nelson Australia
102 Dodds Street
South Melbourne, 3205
Victoria, Australia

Nelson Canada
1120 Birchmount Road
Scarborough, Ontario
Canada M1K 5G4

International Thomson Editores
Seneca 53
Col. Polanco
11560 México, D. F., México

International Thomson Publishing GmbH
Königswinterer Strasse 418
53227 Bonn
Germany

International Thomson Publishing Asia
60 Albert Street
#15-01 Albert Complex
Singapore 189969

International Thomson Publishing Japan
Palaceside Building, 5F
1-1-1 Hitotsubashi
Chiyoda-ku, Tokyo 100-0003
Japan

Printed in the United States of America.

5 4 3 2 1

ISBN 0-534-37462-X

Table of Contents

Preface

This instructor's manual is designed to accompany the textbook, *Introduction to the Theory of Computation*, by Michael Sipser, PWS Publishers, 1997. It contains solutions to almost all of the exercises and problems appearing in Chapters 1–5 and 7 along with a handful of solutions to problems in other chapters. Most of the omitted solutions in the early chapters require figures, and producing these required more work that we were able to put into this manual at this point. The next edition will be more complete.

This manual is available only to instructors and only in hardcopy form. Requests for copies should be directed to: 606–282–4629, fax 606–647–5020.

Thanks to Jonathan Feldman for sending in corrections to a draft of this manual. An errata site for this instructor's manual maintained at

`http://www-math.mit.edu/~sipser/itocsm-errs1.1.html` .

The email address for correspondence regarding the textbook and the instructor's manual is `sipserbook@math.mit.edu` .

Chapter 0

0.12 Here is a sketch of the solution. Make two piles, A and B, of nodes; initially empty. Then, starting with the entire graph, add each remaining node to A if its degree is greater than 1/2 of all remaining nodes and to B otherwise, then discard all nodes to which it isn't (is) connected if it was added to A (B). Continue until nothing is left.

Chapter 1

1.1

a. The start state of M_1 is q_1.
b. The set of accept states of M_1 is $\{q_2\}$.
c. The start state of M_2 is q_1.
d. The set of accept states of M_2 is $\{q_1, q_4\}$.
e. On input aabb M_1 goes through the state sequence q_1, q_2, q_3, q_1, q_1.
f. M_1 does not accept *aabb*.
g. M_2 does accept ε.

1.2

a. $M_1 = (\{q_1, q_2, q_3\}, \{\mathtt{a}, \mathtt{b}\}, \delta_1, q_1, \{q_2\})$.
The transition function δ_1 is

	a	b
q_1	q_2	q_1
q_2	q_3	q_3
q_3	q_2	q_1

b. $M_2 = (\{q_1, q_2, q_3, q_4\}, \{\mathtt{a}, \mathtt{b}\}, \delta_2, q_1, \{q_1, q_4\})$.
The transition function δ_2 is

	a	b
q_1	q_1	q_2
q_2	q_3	q_4
q_3	q_2	q_1
q_4	q_3	q_4

1.9 Let $N = (Q, \Sigma, \delta, q_0, F)$ be any NFA. We build an NFA N' with only a single accept state that accepts the same language as N. Informally, N' looks exactly like N except it has ε-transitions from states corresponding to accept states of N to a new special accept state, q_{accept}. Finally, the state q_{accept}has no transitions coming out of it. More formally, $N' = (Q \cup \{q_{\text{accept}}\}, \Sigma, \delta', q_0, \{q_{\text{accept}}\})$. Where for any $q \in Q$ and $a \in \Sigma$:

$$\delta'(q,a) = \begin{cases} \delta(q,a) & \text{if } a \neq \varepsilon \text{ or } q \notin F \\ \delta(q,a) \cup \{q_{\text{accept}}\} & \text{if } a = \varepsilon \text{ and } q \in F \end{cases}$$

and $\delta'(q_{\text{accept}}, a) = \emptyset$ for $a \in \Sigma_\varepsilon$.

1.10 **a.** Let M' be the DFA M with the accept and non-accept states swapped. We will show that M' recognizes the complement of B, where B is the language recognized by M. Suppose M' accepts x. If we run M' on x we end in an accept state of M'. Because M and M' have swapped accept/non-accept states, if we run M on x, we would end in a non-accept state. Therefore, $x \notin B$. Similarly, if x is not accepted by M', then it would be accepted by M. So M' accepts exactly those strings not accepted by M. Therefore, M' recognizes the complement of B.

Since B could be any arbitrary regular language and our construction shows how to build an automaton to recognize its complement, it follows that the complement of any regular language is also regular. Therefore, the class of regular languages is closed under complement.

b. Consider the NFA in exercise 1.12(a). The string a is accepted by this automaton. If we swap the accept and reject states, the string a is still accepted. This shows that swapping the accept and non-accept states of an NFA doesn't necessarily yield a new NFA recognizing the complement of the original one. The class of languages recognized by NFAs is, however, closed under complement. This follows from the fact that the class of languages recognized by NFAs is precisely the class of languages recognized by DFAs which we know is closed under complement from part (a).

1.13 Let $\Sigma = \{0, 1\}$.

a. $1\Sigma^*0$

b. $\Sigma^*1\Sigma^*1\Sigma^*1\Sigma^*$

c. $\Sigma^*0101\Sigma^*$

d. $\Sigma\Sigma0\Sigma^*$

e. $(0 \cup 1\Sigma)(\Sigma\Sigma)^*$

f. $(0 \cup (10)^*)^*1^*$

g. $(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)$

h. $\Sigma^*0\Sigma^* \cup 1111\Sigma^* \cup 1 \cup \varepsilon$

i. $(1\Sigma)^*(1 \cup \varepsilon)$

j. $0^*(100 \cup 010 \cup 001)0^*$

k. $\varepsilon \cup 0$

l. $(1^*01^*01^*)^* \cup 0^*10^*10^*$

m. $\emptyset$

n. $\Sigma\Sigma^*$

1.15 **a.** ab,ε;ba,aba

b. ab,abab;ε,aabb

c. ε,aa;ab,aabb

d. ε,aaa;aa,b

e. aba,aabbaa;ε,abbb

f. aba,bab;ε,ababab

g. b,ab;ε,bb
h. ba,bba;b,ε

1.16 In both parts we first add a new start state and a new accept state. Several solutions are possible, depending on the order states are removed.

a. Here we remove state 1 then state 2 and we obtain
$\mathtt{a}^*\mathtt{b}(\mathtt{a} \cup \mathtt{b}\mathtt{a}^*\mathtt{b})^*$

b. Here we remove states 1, 2, then 3 and we obtain
$\varepsilon \cup ((\mathtt{a} \cup \mathtt{b})\mathtt{a}^*\mathtt{b}((\mathtt{b} \cup \mathtt{a}(\mathtt{a} \cup \mathtt{b}))\mathtt{a}^*\mathtt{b})^*(\varepsilon \cup \mathtt{a}))$

1.17 **a.** $A_1 = \{0^n1^n2^n \mid n \geq 0\}$. Assume that A_1 is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string $0^p1^p2^p$. Because s is a member of A_1 and s has length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in A_1.

i) The string y consists only of 0s, only of 1s or only of 2s. In these cases the string $xyyz$ will not have equal number of 0s, 1s and 2s, leading to a contradiction.

ii) The string y consists of more than one kind of symbol. In this case $xyyz$ will have the 0s, 1s or 2s out of order. Hence it is not a member of A_1, which is a contradiction.

Therefore, A_1 is not regular.

b. $A_2 = \{ww \mid w \in \{\mathtt{a},\mathtt{b}\}^*\}$. Assume that A_2 is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string $\mathtt{a}^p\mathtt{b}\mathtt{a}^p\mathtt{b}$. The pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in A_2.

i) The string y consists of a's only. In both cases that y is left or right of the middle b, $xyyz$ would not be in the form of ww.

ii) The string y is $\mathtt{a}^p\mathtt{b}$. The condition $|xy| \leq p$ would be violated.

iii) The string y contains the center b and $y \neq \mathtt{a}^p\mathtt{b}$. Therefore, $xyyz$ would not be in the form of ww.

Since all cases cause contradictions, A_2 is not regular.

c. $A_3 = \{\mathtt{a}^{2^n} \mid n \geq 0\}$. Assume that A_3 is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string $\mathtt{a}^{2^p}$. The pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in A_3.
The shortest string in A_3 which is longer than $\mathtt{a}^{2^p}$ is $s' = \mathtt{a}^{2^{p+1}} = \mathtt{a}^{2 \cdot 2^p}$. Thus the length of s' is double the length of s. The only way that $xyyz$ could be in A_3 is by letting y be $\mathtt{a}^{2^p}$. However, that would violate the pumping lemma condition that $|xy| \leq p$. Therefore, A_3 is not regular.

1.18 The error is that $s = 0^p1^p$ *can* be pumped. Let $s = xyz$, where $x = 0$, $y = 0$ and $z = 0^{p-2}1^p$. The conditions are satisfied because

i) for any $i \geq 0$, $xy^iz = 00^i0^{p-2}1^p$ is in 0^*1^*.
ii) $|y| = 1 > 0$, and
iii) $|xy| = 2 \leq p$.

1.19

a. q_1, q_1, q_1, q_1; 000.
b. q_1, q_2, q_2, q_2; 111.
c. q_1, q_1, q_2, q_1, q_2; 0101.
d. q_1, q_3; 1.
e. q_1, q_3, q_2, q_3, q_2; 1111.
f. $q_1, q_3, q_2, q_1, q_3, q_2, q_1$; 110110.
g. q_1; ε.

1.20

A *finite state transducer* is a 5-tuple $(Q, \Sigma, \Gamma, \delta, q_0)$, where

i) Q is a finite set called the *states*,
ii) Σ is a finite set called the *alphabet*,
iii) Γ is a finite set called the *output alphabet*,
iv) $\delta\colon Q \times \Sigma \longrightarrow Q \times \Gamma$ is the *transition function*,
v) $q_0 \in Q$ is the *start state*.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0)$ be a *finite state transducer*, $w = w_1w_2 \cdots w_n$ be a string over Σ, and $v = v_1v_2 \cdots v_n$ be a string over the Γ. Then M *outputs* v if a sequence of states $r_0, r_1, \ldots, r_n$ exists in Q with the following two conditions:

i) $r_0 = q_o$
ii) $\delta(r_i, w_{i+1}) = (r_{i+1}, v_{i+1})$ for $i = 0, \ldots, n-1$.

1.21

a. $T_1 = (Q, \Sigma, \Gamma, \delta, q_1)$, where
i) $Q = \{q_1, q_2\}$,
ii) $\Sigma = \{0, 1, 2\}$,
iii) $\Gamma = \{0, 1\}$,
iv) δ is described as

	0	1	2
q_1	$(q_1,0)$	$(q_1,0)$	$(q_2,1)$
q_2	$(q_1,0)$	$(q_2,1)$	$(q_2,1)$

v) q_1 is the start state.

b. $T_2 = (Q, \Sigma, \Gamma, \delta, q_1)$, where
i) $Q = \{q_1, q_2, q_3\}$,
ii) $\Sigma = \{\mathtt{a}, \mathtt{b}\}$,
iii) $\Gamma = \{0, 1\}$,
iv) δ is described as

	a	b
q_1	$(q_2,1)$	$(q_3,1)$
q_2	$(q_3,1)$	$(q_1,0)$
q_3	$(q_1,0)$	$(q_2,1)$

v) q_1 is the start state.

1.23 **a.** Assume $L = \{0^n1^m0^n \mid m,n \geq 0\}$ is regular. Let p be the pumping length given by the pumping lemma. The string $s = 0^p10^p \in L$, and $|s| \geq p$. Thus the pumping lemma implies that s can be divided as xyz with $x = 0^a, y = 0^b, z = 0^c10^p$, where $b \geq 1$ and $a+b+c = p$. However, the string $s' = xy^0z = 0^{a+c}10^p \notin L$, since $a+c < p$. That contradicts the pumping lemma.

b. If L, the complement of $C = \{0^n1^n \mid n \geq 0\}$ is regular, then C itself is regular, using the result of Exercise 1.10a. However, we have shown in Example 1.38 that C is non-regular. So L cannot be regular.

c. Assume $L = \{0^m1^n \mid m \neq n\}$ is regular. Let p be the pumping length given by the pumping lemma. Set $m = p$ and $n = p+p!$. The string $s = 0^p1^{p+p!} \in L$, and $|s| \geq p$. Thus the pumping lemma implies that s can be divided as xyz with $x = 0^a, y = 0^b, z = 0^c1^{p+p!}$, where $b \geq 1$ and $a+b+c = p$. However, the string $s' = xy^{\frac{p!}{b}+1}z = 0^{a+p!+b+c}1^{p+p!} = 0^{p+p!}1^{p+p!} \notin L$. That contradicts the pumping lemma.
Alternative solution: The nonregular language $\{0^n1^n \mid n \geq 0\} = C \cap 0^*1^*$. Hence C cannot be regular because the regular languages are closed under intersection.

d. Assume $C = \{w \mid w \in \{0,1\}^* \text{ is a palindrome}\}$ is regular. Let p be the pumping length given by the pumping lemma. The string $s = 0^p10^p \in C$ and $|s| \geq p$. Follow the argument as in part (a). Hence C isn't regular, so neither is its complement.

1.24 For any regular language A, let M_1 be the DFA recognizing it. We need to find a DFA that recognizes A^R. Since any NFA can be converted to an equivalent DFA, it suffices to find an NFA M_2 that recognizes A^R.

We keep all the states in M_1 and reverse the direction of all the arrows in M_1. We set the accept state of M_2 to be the start state in M_1. Also, we introduce a new state q_0 as the start state for M_2 which goes to every accept state in M_1 by an ϵ-transition.

For every string s accepted by M_1, the path that s follows in M_1 starts at the start state and stops at one of the accept states, say, q_{end}. If we input s^R to M_2, it starts at q_0 and forks into different paths starting at each of the accept states in M_1. The computation can reach the accept state in M_2 by following the same path as its reversed counterpart in M_1 but in the opposite direction.

Similarly, for every string w accepted by M_2, there exists a path P starting at q_0 going through q_{end} as its next step and stopping at the unique accept

state. Converting the NFA M_2 back to its counterpart M_1 and reversing the input string, we can get from the start state to q_{accept} in M_1 by following the same path in the opposite direction.

Therefore, M_2 recognizes A^R if M_1 recognizes A.

1.27 The idea is that we start by comparing the most significant bit of the two rows. If the bit in the top row is bigger, we know that the string is in the language. The string does not belong to the language if the bit in the top row is smaller. If the bits on both rows are the same, we move on to the next most significant bit until a difference is found. We implement this idea with a DFA having states q_0, q_1, and q_2. State q_0 indicates the result is not yet determined. States q_1 and q_2 indicate the top row is known to be larger, or smaller, respectively. We start with q_0. If the top bit in the input string is bigger, it goes to q_1, the only accept state, and stays there till the end of the input string. If the top bit in the input string is smaller, it goes to q_2 and stays there till the end of the input string. Otherwise, it stays in state q_0.

1.28 Assume language E is regular. Use the pumping lemma to a get a pumping length p satisfying the conditions of the pumping lemma. Set $s = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^p \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]^p$. Obviously, $s \in E$ and $|s| \geq p$. Thus, the pumping lemma implies that the string s can be written as xyz with $x = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^a$, $y = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^b$, $z = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^c \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]^p$, where $b \geq 1$ and $a + b + c = p$. However, the string $s' = xy^0z = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^{a+c} \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]^p \notin E$, since $a + c < p$. That contradicts the pumping lemma. Thus E is not regular.

1.29 For each $n \geq 1$, we build a DFA with the n states $q_0, q_1, \ldots, q_{n-1}$ to count the number of consecutive a's modulo n read so far. For each character a that is input, the counter increments by 1 and jumps to the next state in M. It accepts the string if and only if the machine stops at q_0. That means the length of the string consists of all a's and its length is a multiple of n.

More formally, the set of states of M is $Q = \{q_0, q_1, \ldots, q_{n-1}\}$. The state q_0 is the start state and the only accept state. Define the transition function as: $\delta(q_i, a) = q_j$ where $j = i + 1 \bmod n$.

1.30 By simulating binary division, we create a DFA M with n states that recognizes C_n. M has n states which keep track of the n possible remainders of the division process. The start state is the only accept state and corresponds to remainder 0.

The input string is fed into M starting from the most significant bit. For each input bit, M doubles the remainder that its current state records, and then adds the input bit. Its new state is the sum modulo n. We double the remainder because that corresponds to the left shift of the computed remainder in the long division algorithm.

If an input string ends at the accept state (corresponding to remainder 0), the binary number has no remainder on division by n and is therefore a member of C_n.

The formal definition of M is $(\{q_0, \ldots, q_{n-1}\}, \{0,1\}, \delta, q_0, \{q_0\})$. For each $q_i \in Q$ and $b \in \{0,1\}$, define $\delta(q_i, b) = q_j$ where $j = (2i + b) \bmod n$.

1.32 **a.** Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA recognizing A, where A is some regular language. We construct $M' = (Q', \Sigma, \delta, q_0', F')$ recognizing $NOPREFIX(A)$ as follows:

i) $Q' = Q$

ii) For $r \in Q'$ and $a \in \Sigma$ define $\delta'(r,a) = \begin{cases} \delta(r,a) & R \notin F \\ \emptyset & R \in F. \end{cases}$

iii) $q_0' = q_0$

iv) $F' = F$

b. Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA recognizing A, where A is some regular language. We construct $M' = (Q', \Sigma, \delta, q_0', F')$ recognizing $NOEXTEND(A)$ as follows:

i) $Q' = Q$

ii) $\delta' = \delta$

iii) $q_0' = q_0$

iv) $F' = \{q|$ there is no path from q to an accept state$\}$.

1.33 Assume to the contrary that some FST T outputs $w^{\mathcal{R}}$ on input w. Consider the input strings 00 and 01. On input 00, T must output 00, and on input 01, T must output 10. That is impossible, because in both cases the first input bit is a 0, but in one case the first output bit is a 0 and in the other case the first output is a 1, and the first output bit depends only on the first input bit. Hence no such FST can exist.

1.34 To show that $\equiv_L$ is an equivalence relation we show it is reflexive, symmetric, and transitive. It is reflexive because no string can distinguish x from itself and hence $x \equiv_L x$ for every x. It is symmetric because x is distinguishable from y whenever y is distinguishable from x. It is transitive because if $w \equiv_L x$ and $x \equiv_L y$, then for each z, $wz \in L$ iff $xz \in L$ and $xz \in L$ iff $yz \in L$, hence $wz \in L$ iff $yz \in L$, and so $w \equiv_L y$.

1.35 **a.** We prove this assertion by contradiction. Let M be a k-state DFA that recognizes L. Suppose for a contradiction that L has index greater than k. That means some set X with more than k elements is pairwise distinguishable by L. Because M has k states, the pigeonhole principle implies that X contains two distinct strings x and y where $\delta(q_0, x) = \delta(q_0, y)$. Here $\delta(q_0, x)$ is the state that M is in after starting in the start state q_0 and reading

input string x. Then, for any string $z \in \Sigma^*$, $\delta(q_0, xz) = \delta(q_0, yz)$. Therefore either both xz and yz are in L or neither are in L. But then x and y aren't distinguishable by L, contradicting our assumption that X is pairwise distinguishable by L.

b. Let $X = \{s_1, \ldots, s_k\}$ be pairwise distinguishable by L. We construct DFA $M = (Q, \Sigma, \delta, q_0, F)$ with k states recognizing L. Let $Q = \{q_1, \ldots, q_k\}$, and define $\delta(q_i, a)$ to be q_j where $s_j \equiv_L s_i a$ (the relation $\equiv_L$ is defined in problem 1.34). Note that $s_j \equiv_L s_i a$ for some $s_j \in X$ otherwise $X \cup s_i a$ would have $k+1$ elements and would be pairwise distinguishable by L and that contradicts our assumption that L has index k. Let $F = \{q_i |\ s_i \in L\}$. Let the start state, q_0, be the q_i such that $s_i \equiv_L \varepsilon$. M is constructed so that, for any state q_i, $\{s |\ \delta(q_0, s) = q_i\} = \{s |\ s \equiv_L s_i\}$. Hence M recognizes L.

c. Suppose L is regular and let k be the number of states in a DFA recognizing L. Then from part (a) L has index at most k. Conversely, if L has index k, then by part (b) it is recognized by a DFA with k states, and thus is regular. To show that the index of L is the size of the smallest DFA accepting it, suppose that L's index is *exactly* k. Then, by part (b), there is a k-state DFA accepting L. That is the smallest such DFA since if it were any smaller, then we could show by part (a) that the index of L is less than k.

1.36 Assume to the contrary that *ADD* is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string $1^p 0^p 1^p$, which is a member of *ADD*. Because s has length greater than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, satisfying the conditions of the lemma. By the third condition in the pumping lemma have that $|xy| \leq p$, it follows that y is 1^k for some $k \geq 1$. Then xy^2z is the string $1^{p+k} 0^p 1^p$, which is not a member of *ADD*, violating the pumping lemma. Hence *ADD* isn't regular.

1.37 **Note:** The version of the problem that appears in the first printing is incorrect. The correct version (which appears in the second and subsequent printings) is:

Show that the language $F = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k |\ i, j, k \geq 0 \text{ and if } i = 1 \text{ then } j = k\}$ satisfies the three conditions of the pumping lemma even though it is not regular. Explain why this fact does not contradict the pumping lemma.

Language F satisfies the conditions of the pumping lemma using pumping length 2. If $s \in F$ is of length 2 or more we show that it can be pumped by considering four cases, depending on the number of a's that s contains.

i) If s is of the form $\mathtt{b}^*\mathtt{c}^*$, let $x = \varepsilon$, y be the first symbol of s, and let z be the rest of s.

ii) If s is of the form $\mathtt{a}\mathtt{b}^*\mathtt{c}^*$, let $x = \varepsilon$, y be the first symbol of s, and let z be the rest of s.

iii) If s is of the form $\mathtt{a}\mathtt{a}\mathtt{b}^*\mathtt{c}^*$, let $x = \varepsilon$, y be the first two symbols of s, and let z be the rest of s.

iv) If s is of the form $\mathtt{aaa^*b^*c^*}$, let $x = \varepsilon$, y be the first symbol of s, and let z be the rest of s.

In each case, the strings xy^iz are members of F for every $i \geq 0$. Hence F satisfies the conditions of the pumping lemma. However, F is clearly not regular, because the nonregular language $\{\mathtt{ab}^n\mathtt{c}^n \mid n \geq 0\}$ is the same as $F \cap \mathtt{ab^*c^*}$, and the regular languages are closed under intersection. The pumping lemma is not violated because it states only that regular languages satisfy the three conditions, and it doesn't state that nonregular languages fail to satisfy the three conditions.

1.38 The objective of this problem is for the student to pay close attention to the exact formulation of the pumping lemma.

a. The minimum pumping length is 4. The string $s = 000 \in 0001^*$ cannot be pumped, and any string in 0001^* of length 4 or more contains a 1 and hence can be pumped by dividing it so that $x = 0$, $y = 1$, and z is the rest.

b. The minimum pumping length is 1. The string ε cannot be pumped because of condition 2 in the pumping lemma and hence 0 is not a pumping length. Any string in 0^*1^* of length 1 or more contains a 0 or a 1 and hence can be pumped by dividing it so that $x = \varepsilon$, $y = 0$ or 1, and z is the rest, so 1 is a pumping length.

c. The minimum pumping length is 1. The pumping length cannot be 0, as in part (b). Any string in $(01)^*$ of length 1 or more contains 01 and hence can be pumped by dividing it so that $x = \varepsilon$, $y = 01$, and z is the rest.

d. The minimum pumping length is 3. The string 01 cannot be pumped, so 2 is not a pumping length. All strings in the language of length 3 or more can be pumped, (note that there aren't any such string, so the statement is true vacuously) so 3 is a pumping length.

e. The minimum pumping length is 1. String ε cannot be pumped because of condition 2 in the pumping lemma and hence 0 is not a pumping length. The language has no strings of length 1 or more so 1 is a pumping length. (the conditions hold vacuously).

1.39 Let $A_k = \Sigma^*0^{k-1}0^*$. A DFA with k states can recognize A_k. In addition to the start state, it has one state for each number of 0s it has read since the last 1. After $k-1$ 0s the machine enters a accept state whose transitions are self-loops.

In any DFA with fewer than k states, two of the k strings 1, 10, ... , 10^{k-1} must cause the machine to enter the same state, by the pigeon hole principle. But then, if we add to both of these strings enough 0s to cause the longer of these two strings to have exactly $k-1$ 0s, the two new strings will still cause the machine to enter the same state, but one of these strings is in A_k and the other is not. Hence, the machine must fail to produce the correct accept/reject response on one of these strings.

1.40 Let $A = \{2^n \mid n \geq 0\}$. The set $B_3(A)$ consists of all strings of the form 10^n which is obviously regular. We show by contradiction that $B = B_3(A)$ isn't regular. Suppose that it is. Then there exists a pumping length p such that for any string $s \in B$ with $|s| \geq p$, we can break s up into strings x, y, and z satisfying the conditions of the lemma. Consider the string $s \in B$ that is the base 3 representation of 2^l where we choose l large enough to force s to have length at least p. Then $s = xyz$ satisfying the conditions of the lemma. In particular, $xyz \in B$, $xy^2z \in B$, and $xz \in B$. Now, define the length of y to be m and define the length of x to be n. Recall that m is strictly greater than 0. Treat strings as base 3 representations of numbers in the following equations:

i) $xz = x3^m + z$. (Call this quantity q_0)

ii) $xyz = x3^{m+n} + y3^n + z$. (Call this quantity q_1)

iii) $xyyz = x3^{2m+n} + y3^{m+n} + y3^n + z$. (Call this quantity q_2)

We now derive a contradiction algebraically. Using equations (i), (ii), and (iii), we have

$$\frac{q_2 - q_1}{q_1 - q_0} = \frac{x3^{2m+n} + (y-x)3^{m+n}}{x3^{m+n} + (y-x)3^n} = 3^m.$$

We know from the pumping lemma that $xz, xyz, xyyz \in B$ so it follows that $q_0 = 2^{l_0}$ and $q_2 = 2^{l_2}$, for some $l_0, l_2 \geq 0$. Because $s = xyz$ and s is the base 3 representation of 2^l, we have $q_1 = 2^l$. Moreover, $l_0 < l < l_2$. Thus,

$$\frac{q_2 - q_1}{q_1 - q_0} = \frac{2^{l_2} - 2^l}{2^l - 2^{l_0}} = \frac{2^{l_2-l_0} - 2^{l-l_0}}{2^{l-l_0} - 1} = \frac{2^{l-l_0}(2^{l_2-l} - 1)}{2^{l-l_0} - 1}.$$

Setting these two expression equal we obtain

$$\frac{2^{l-l_0}(2^{l_2-l} - 1)}{2^{l-l_0} - 1} = 3^m \quad \text{and thus} \quad 2^{l-l_0}(2^{l_2-l} - 1) = 3^m(2^{l-l_0} - 1).$$

The left-hand side of this equation is an even number and the right-hand side of this equation is an odd number. Hence we arrive at a contradiction.

1.41 The language D can be written in another way, namely, $0\Sigma^*0 \cup 1\Sigma^*1$, which is obviously regular.

1.42 Given an NFA M recognizing A we construct an NFA N accepting $A_{\frac{1}{2}}$ using the following idea. M keeps track two states in N using two "fingers". As it reads each input symbol N uses one finger to simulate M on that symbol. At the same time it from an accept state on a guessed symbol. Simultaneously, M uses the other finger to run M backwards from an accept state on a guessed symbol. N accepts whenever the forward simulation and the backward simulation are in the same state, that is, whenever the two fingers re together. At those points we are sure that N has found a string

where another string of the same length can be appended to yield a member of A, precisely the definition of $A_{\frac{1}{2}-}$.

Formally, we assume for simplicity that the NFA M recognizing A has a single accept state as guaranteed by Exercise 1.9. Let $M = (Q, \Sigma, \delta, q_0, q_{\text{accept}})$. Construct NFA $N = (Q', \Sigma, \delta', q_0', F')$ recognizing the first halves of the strings in A as follows:

i) $Q' = Q \times Q$.

ii) For $q, r \in Q$ define $\delta'((q,r),a) = \{(u,v) |\ u \in \delta(q,a)$ and $r \in \delta(v,b)$ for some $b \in \Sigma\}$.

iii) $q_0' = (q_0, q_{\text{accept}})$.

iv) $F' = \{(q,q) |\ q \in Q\}$.

1.43 Let $A = \{\mathtt{0}^*\mathtt{\#}\mathtt{1}^*\}$. Thus, $A_{\frac{1}{3}-\frac{1}{3}} \cap \{\mathtt{0}^*\mathtt{1}^*\} = \{\mathtt{0}^n\mathtt{1}^n |\ n \geq 0\}$. Regular sets are closed under intersection, and $\{\mathtt{0}^n\mathtt{1}^n |\ n \geq 0\}$ is not regular, so $A_{\frac{1}{3}-\frac{1}{3}}$ is not regular.

1.44 Let $E_n = \{x \in \{0,1\}^* |$ the $(n-1)$st symbol from the right in x is $1\}$ for $n \geq 2$. Constructing an n-state NFA for E_n is easy.

Let M be a DFA recognizing E_n. We claim that M has at least 2^{n-1} states. Let S be the set of all strings of length $n-1$. For each y in S let q_y be the state that M is in after starting at the start state and the reading y. If $y \neq z$, then $q_y \neq q_z$, otherwise M fails to recognize the E_n for the following reason. Say that y and z differ on the ith bit. Then one of the strings $y0^{i-1}$ and $z0^{i-1}$ is in E_n and the other one isn't. But M has the same behavior on both, so M fails to recognize E_n. Hence, M has at least as many states a S has strings, and so M has at least 2^{n-1} states.

Chapter 2

2.1 Here are the derivations (but not the parse trees).

a. $E \Rightarrow T \Rightarrow F \Rightarrow \mathtt{a}$

b. $E \Rightarrow E\mathtt{+}T \Rightarrow T\mathtt{+}T \Rightarrow F\mathtt{+}T \Rightarrow F\mathtt{+}F \Rightarrow \mathtt{a+}F \Rightarrow \mathtt{a+a}$

c. $E \Rightarrow E\mathtt{+}T \Rightarrow E\mathtt{+}T\mathtt{+}T \Rightarrow T\mathtt{+}T\mathtt{+}T \Rightarrow F\mathtt{+}T\mathtt{+}T \Rightarrow F\mathtt{+}F\mathtt{+}T \Rightarrow F\mathtt{+}F\mathtt{+}F \Rightarrow \mathtt{a+}F\mathtt{+}F \Rightarrow \mathtt{a+a+}F \Rightarrow \mathtt{a+a+a}$

d. $E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (T) \Rightarrow (F) \Rightarrow ((E)) \Rightarrow ((T)) \Rightarrow ((F)) \Rightarrow ((\mathtt{a}))$

2.2 **a.** First, we show that both A and B are context-free. It's not hard to see that the following grammar recognizes A:

$$\begin{aligned} S &\to RT \\ R &\to \mathtt{a}R \mid \varepsilon \\ T &\to \mathtt{b}T\mathtt{c} \mid \varepsilon \end{aligned}$$

The following grammar recognizes B:

$$S \to TR$$
$$T \to \mathtt{a}T\mathtt{b} \mid \varepsilon$$
$$R \to \mathtt{c}R \mid \varepsilon$$

So, both A and B are context-free languages. Now, observe that $A \cap B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. We know from Example 2.20 that this language is not context-free. We have found two context free languages whose intersection is not context-free. Therefore context-free languages are not closed under intersection.

b. First, the context-free languages are closed under the union operation. Let $G_1 = (V_1, \Sigma, R_1, S_1)$ and $G_2 = (V_2, \Sigma, R_2, S_2)$ be two arbitrary context free grammars. We construct a grammar G that recognizes their union. Formally, $G = (V, \Sigma, R, S)$ where:

i) $V = V_1 \cup V_2$

ii) $R = R_1 \cup R_2 \cup \{S \to S_1,\ S \to S_2\}$
(Here we assume that R_1 and R_2 are disjoint, otherwise we change the variable names to ensure disjointness)

Next, we show that the CFGs are not closed under complementation. Assume, for a contradiction, that the CFGs are closed under complementation. Then, if G_1 and G_2 are context free grammars, it would follow that $\overline{L(G_1)}$ and $\overline{L(G_2)}$ are context free. We previously showed that context-free languages are closed under union and so $\overline{L(G_1)} \cup \overline{L(G_1)}$ is context free. That, by our assumption, implies that $\overline{\overline{L(G_1)} \cup \overline{L(G_1)}}$ is context free. But by DeMorgan's laws, $\overline{\overline{L(G_1)} \cup \overline{L(G_1)}} = L(G_1) \cap L(G_2)$. However, if G_1 and G_2 are chosen as in part (a), $\overline{\overline{L(G_1)} \cup \overline{L(G_1)}}$ isn't context free. This contradiction shows that the context-free languages are not closed under complementation.

2.3

a. The variables of G are: R, X, S, T. The terminals of G are: $\mathtt{a}, \mathtt{b}$. The start variable is R.

b. Three strings in G are: $\mathtt{ab}, \mathtt{ba}$, and $\mathtt{aab}$.

c. Three strings not in G are: $\mathtt{a}, \mathtt{b}$, and ε.

d. False.

e. True.

f. False.

g. True.

h. True.

i. False.

j. True.

k. True.

l. False.

m. $L(G)$ consists of all strings over $\mathtt{a}$ and $\mathtt{b}$ that are not palindromes.

2.4

a. $S \to R1R1R1R$
$R \to 0R \mid 1R \mid \varepsilon$

b. $S \to 0R0 \mid 1R1 \mid \varepsilon$
$R \to 0R \mid 1R \mid \varepsilon$

c. $S \to 0 \mid 1 \mid 00S \mid 01S \mid 10S \mid 11S$

d. $S \to 0 \mid 0S0 \mid 0S1 \mid 1S0 \mid 1S1$

e. $S \to B1B$
$B \to BB \mid 0B1 \mid 1B0 \mid 1 \mid \varepsilon$
This CFG works because B generates all strings that have at least as many 1s as 0s, and so S forces an extra 1.

f. $S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$

g. $S \to S$

2.5

a. This set is regular, so the PDA doesn't even need to use its stack. The PDA scans the string and uses its finite control to maintain a counter which counts up to 3. It keeps track of the number of 1's it has seen using this counter. The PDA accepts the moment it sees three ones.

b. This set is also regular. The PDA scans the string and keep track of the first and last symbol in its finite control. If they are the same, it accepts, otherwise it rejects.

c. Again, this set is regular. The PDA scans the string and keep track of the length (modulo 2) using its finite control. If the length is 1 (modulo 2) it accepts, otherwise it rejects.

d. The PDA scans across the string and push the symbols onto the stack. At some point it nondeterministically guesses where the middle is. It looks at the middle symbol. If that symbol is a 1, it rejects. Otherwise, it scans the rest of the string, and for each character scanned, it pops one element off of its stack. If the stack is empty before it reaches the end of the input, it rejects. If the stack is empty when it finishes reading the input (i.e. it correctly guessed the middle) then it accepts.

e. The PDA scans across the input. If it sees a 1 and its top stack symbol is a 0, it pops the stack. Similarly, if it scans a 0 and its top stack symbol is a 1, it pops the stack. In all other cases, it pushes the input symbol onto the stack. After it scans the input, if there is a 1 on top of the stack, it accepts. Otherwise it rejects.

f. The PDA begins by scanning across the string and pushing each symbol onto its stack. At some point it nondeterministically guesses when it has reached the middle. It also nondeterministically guesses if string has odd length or even length. If it guessed even, then it pushes the current symbol it's reading onto the stack (recall that the PDA has guessed that this symbol is the middle symbol). If it guesses that string has odd length, it goes to the next input symbol without changing the stack. Now, it scans the rest of the string, and it compares each symbol it scans to the symbol on the top of the stack. If they are the same, it pops the stack, and continues scanning. If they are different, it rejects. If the stack is empty before it finishes reading all the

input, it rejects. If the stack becomes empty just after it reaches the end of the input then it accepts. In all other cases, it rejects.

g. The PDA simply rejects immediately.

2.6 a. $S \to S\mathtt{a}S\mathtt{a}S\mathtt{b}S \mid S\mathtt{a}S\mathtt{b}S\mathtt{a}S \mid S\mathtt{b}S\mathtt{a}S\mathtt{a}S \mid \varepsilon$

b. $S \to X\mathtt{b}X\mathtt{a}B \mid T \mid U$
$T \to \mathtt{a}T\mathtt{b} \mid T\mathtt{b} \mid \mathtt{b}$
$U \to \mathtt{a}U\mathtt{b} \mid \mathtt{a}U \mid \mathtt{a}$
$X \to \mathtt{a} \mid \mathtt{b}$

c. $S \to TX$
$T \to 0T0 \mid 1T1 \mid \#X$
$X \to 0X \mid 1X \mid \varepsilon$

d. $S \to UPV$
$P \to \mathtt{a}P\mathtt{a} \mid \mathtt{b}P\mathtt{b} \mid T \mid \varepsilon$
$T \to \#MT \mid \#$
$U \to M\#U \mid \varepsilon$
$V \to \#MV \mid \varepsilon$
$M \to \mathtt{a}M \mid \mathtt{b}M \mid \varepsilon$
Note that we need to allow for the case when $i = j$, that is, some x_i is a palindrome. Also, ε is in the language since it's a palindrome.

2.7 a. The PDA uses its stack to keep a count of the number of a's minus twice the number of b's. It enters an accepting state whenever this count is 0. In more detail, it operates as follows. It reads the next input symbol. If it is an a and the top of the stack is a b, it pops the b. If it is an a and the top of is either an a or empty, it pushes the a. If the next input symbol is a b, then repeat the following sentence twice. If the top if the stack is an a, pop the stack, otherwise push the b.

b. The PDA scans across the string. It pushes the a's it reads onto the stack until it encounters a b. Then, for each b it reads, it pops one a off the stack. If the machine encounters an a after a b, it accepts (by entering an accept state and moving its head to the end of the input). If the stack becomes empty before the end of the input is reached, of if the end of the input is reached without the stack becoming empty, the machine accepts. Otherwise it rejects.

c. The PDA scans across the input string and pushes every symbol it reads until it reads a #. If # is never encountered, reject. Then, it skips over part of the input, nondeterministically deciding when to stop skipping. Then, it compares the input symbols it next read with the symbols it pops off the stack. At any disagreement, or if the input finishes while the stack is nonempty, this branch of the computation rejects. If the stack becomes empty, the machine accepts.

d. The PDA uses its nondeterminism to guess the values of i and j and uses its stack to compare x_i with $x_j^{\mathcal{R}}$. In more detail, first it nondeterministically branches to the procedures in the next two paragraphs.

The machine begins by nondeterministically skipping over a part (possibly empty) of the input until it arrives at a symbol which is either at the beginning of the input string or is immediately to the right of a #. Then, the machine continues to read input symbols, while pushing them onto the stack. During this reading and popping phase, two cases arise. In one case, if an input # is read, the machine nondeterministically skips to another # in the input and continues reading inputs symbols, now comparing them with symbols popped off the stack. If all agree and the stack becomes empty at the same time as either the end of the input of # symbol is reached, accept. However, if a disagreement occurs, or the stack becomes empty before either the end of the input of # symbol is reached, or either the end of the input of # symbol is reached without the stack becoming empty, reject on this branch of the nondeterminism. In the other case, during the reading and popping phase, the machine may nondeterministically chose to enter the reading and popping phase, as just described. Then it continues to operate as in that case.

2.8 Here is one derivation:

⟨SENTENCE⟩ ⇒
⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩ ⇒
⟨CMPLX-NOUN⟩⟨VERB-PHRASE⟩ ⇒
⟨CMPLX-NOUN⟩⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩ ⇒
⟨ARTICLE⟩⟨NOUN⟩⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩ ⇒
The boy ⟨VERB⟩⟨NOUN-PHRASE⟩⟨PREP-PHRASE⟩ ⇒
The boy ⟨VERB⟩⟨NOUN-PHRASE⟩⟨PREP⟩⟨CMPLX-NOUN⟩ ⇒
The boy touches ⟨NOUN-PHRASE⟩⟨PREP⟩⟨CMPLX-NOUN⟩ ⇒
The boy touches ⟨CMPLX-NOUN⟩⟨PREP⟩⟨CMPLX-NOUN⟩ ⇒
The boy touches ⟨ARTICLE⟩⟨NOUN⟩⟨PREP⟩⟨CMPLX-NOUN⟩ ⇒
The boy touches the girl with ⟨CMPLX-NOUN⟩ ⇒
The boy touches the girl with ⟨ARTICLE⟩⟨NOUN⟩ ⇒
The boy touches the girl with the flower

Here is another derivation:

⟨SENTENCE⟩ ⇒
⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩ ⇒
⟨CMPLX-NOUN⟩⟨VERB-PHRASE⟩ ⇒
⟨ARTICLE⟩⟨NOUN⟩⟨VERB-PHRASE⟩ ⇒
The boy ⟨VERB-PHRASE⟩ ⇒
The boy ⟨CMPLX-VERB⟩ ⇒
The boy ⟨VERB⟩⟨NOUN-PHRASE⟩ ⇒
The boy touches ⟨NOUN-PHRASE⟩ ⇒
The boy touches ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩ ⇒
The boy touches ⟨ART⟩⟨NOUN⟩⟨PREP-PHRASE⟩ ⇒
The boy touches the girl ⟨PREP-PHRASE⟩ ⇒
The boy touches the girl ⟨PREP⟩⟨CMPLX-NOUN⟩ ⇒
The boy touches the girl with ⟨CMPLX-NOUN⟩ ⇒

`The boy touches the girl with` ⟨ARTICLE⟩⟨NOUN⟩ ⇒
`The boy touches the girl with the flower`

Each of these derivations corresponds to a different English meaning. In the first derivation, the sentence means that the boy used the flower in order to touch the girl. In the second derivation, the girl happens to be holding the flower when the boy touches her.

2.9 A CFG G that generates A is given as follows:
$G = (V, \Sigma, R, S)$.
V is $\{S, E_{ab}, E_{bc}, C, A\}$ and Σ is $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. The rules are

$$\begin{aligned} S &\rightarrow E_{ab}C \mid AE_{bc} \\ E_{ab} &\rightarrow \mathtt{a}E_{ab}\mathtt{b} \mid \varepsilon \\ E_{bc} &\rightarrow \mathtt{b}E_{bc}\mathtt{c} \mid \varepsilon \\ C &\rightarrow C\mathtt{c} \mid \varepsilon \\ A &\rightarrow A\mathtt{a} \mid \varepsilon \end{aligned}$$

Initially substituting $E_{ab}C$ for S generates any string with an equal number of as and bs followed by any number of cs. Initially substituting E_{bc} for S generates any string with an equal number of bs and cs prepended by any number of as.

The grammar is ambiguous. Consider the string ε. On the one hand, it can be derived by choosing E_{ab}C with each of E_{ab} and C yielding ε. On the other hand, ε can be derived by choosing AE_{bc} with each of A and E_{bc} yielding ε. In general, any string $\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k$ with $i = j = k$ can be derived ambiguously in this grammar.

2.10 Informal description of a PDA that recognizes A in Exercise 2.9:

1. Nondeterministically branch to either step 2 or step 6.
2. Read and push a's.
3. Read b's, while popping a's.
4. If b's finish when stack is empty, skip c's on input and *accept*.
5. Skip a's on input.
6. Read and push b's.
7. Read c's, while popping b's.
8. If c's finish when stack is empty, *accept*.

2.11 Informal description of a PDA that recognizes the CFG in Exercise 2.1:

1. Place the marker symbol \$ and the start variable E on the stack.
2. Repeat the following steps forever.

3. If the top of stack is the variable E, pop it and nondeterministically push either E+T or T into the stack.
4. If the top of stack is the variable T, pop it and nondeterministically push either T×F or F into the stack.
5. If the top of stack is the variable F, pop it and nondeterministically push either (E) or a into the stack.
6. If the top of stack is a terminal symbol, read the next symbol from the input and compare it to the terminal in the stack. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
7. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

The formal definition of the equivalent PDA is $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where Here $Q = \{q_1, q_2\}$; $\Sigma = \{+, \times, (,), \mathtt{a}\}$; and $\Gamma = \{E, T, F\} \cup \Sigma$; $F = \{q_2\}$. The transition function $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is given as follows.

$$\delta(q,x,y) = \begin{cases} \{(q_2,\varepsilon)\} & \text{if } q = q_1, x = \varepsilon, y = \$ \\ \{(q_1,E{+}T),(q_1,T)\} & \text{if } q = q_1, x = \varepsilon, y = E \\ \{(q_1,T{\times}F),(q_1,F)\} & \text{if } q = q_1, x = \varepsilon, y = T \\ \{(q_1,(E)),(q_1,\mathtt{a})\} & \text{if } q = q_1, x = \varepsilon, y = F \\ \{(q_1,\varepsilon)\} & \text{if } q = q_1, x = y \end{cases}$$

2.12 Informal description of a PDA that recognizes the CFG in Exercise 2.3:

1. Place the marker symbol \$ and the start variable R on the stack.
2. Repeat the following steps forever.
3. If the top of stack is the variable R, pop it and nondeterministically push either XRX or S into the stack.
4. If the top of stack is the variable S, pop it and nondeterministically push either aTb or bTa into the stack.
5. If the top of stack is the variable T, pop it and nondeterministically push either XTX, X or ε into the stack.
6. If the top of stack is the variable X, pop it and nondeterministically push either a or b into the stack.
7. If the top of stack is a terminal symbol, read the next symbol from the input and compare it to the terminal symbol in the stack. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
8. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

The formal definition of the equivalent PDA is $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where $Q = \{q_1, q_2\}$; $\Sigma = \{\mathtt{a}, \mathtt{b}\}$; $\Gamma = \{R, S, T, X\} \cup \Sigma$; and $F = \{q_2\}$. The transition

function $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is given as follows.

$$\delta(q,x,y) = \begin{cases} \{(q_2,\varepsilon)\} & \text{if } q = q_1, x = \varepsilon, y = \$ \\ \{(q_1,XRX),(q_1,S)\} & \text{if } q = q_1, x = \varepsilon, y = R \\ \{(q_1,\mathtt{a}T\mathtt{b}),(q_1,\mathtt{b}T\mathtt{a})\} & \text{if } q = q_1, x = \varepsilon, y = S \\ \{(q_1,XTX),(q_1,X),(q_1,\varepsilon)\} & \text{if } q = q_1, x = \varepsilon, y = T \\ \{(q_1,\mathtt{a}),(q_1,\mathtt{b})\} & \text{if } q = q_1, x = \varepsilon, y = X \\ \{(q_1,\varepsilon)\} & \text{if } q = q_1, x = y \end{cases}$$

2.13 **a.** $L(G)$ is the set of strings of 0s and #s that either contain exactly 2 #s and any number of 0s, or contain exactly 1 # and the number of 0s to the right of the # is twice the number of 0s to the left.

b. Assume $L(G)$ is regular. Let $A = L(G) \cap 0^*\#0^*$. If $L(G)$ is regular, so is A. Let p be the pumping length for A given by the pumping lemma for regular languages. Consider the string $w = 0^p\#0^{2p}$. Because $|w| > p$ and $w \in A$, the pumping lemma that $w = xyz$ such that $|xy| \leq p$, $y \neq \varepsilon$ and $xy^iz \in L(G)\ \forall i \geq 0$. We investigate all possible ways of cutting w and prove that such a cut cannot exist.

i) x contains the character #. In this case, y is to the right of #. Pumping it down makes the number of 0s on the right less than twice of the number of 0s on the left.

ii) y contains the character #. Pumping y down makes the resulting string contain no #s.

iii) z contains the character #. In this case, y is to the left of #. Pumping y down makes the number of 0s on the right more than twice of the number of 0s on the left.

In each case the resulting string doesn't belong to A. Hence A fails the pumping lemma, so it cannot be regular, and neither can $L(G)$.

2.14 The equivalent CFG in Chomsky normal form is given as follows.

$$\begin{aligned} S_0 &\to AB \mid CC \mid BA \mid BD \mid BB \mid \varepsilon \\ A &\to AB \mid CC \mid BA \mid BD \mid BB \\ B &\to CC \\ C &\to 0 \\ D &\to AB \end{aligned}$$

2.15 Let $L(G_1)$ and $L(G_2)$ be CFGs where G_1 and G_2 are defined as follows. (Here we assume that that the sets of variables are disjoint.) let $G_1 = (V_1, \Sigma, R_1, S_1)$ and $G_2 = (V_2, \Sigma, R_2, S_2)$.

We construct a new CFG $G_\cup$ where $L(G_\cup) = L(G_1) \cup L(G_2)$. Let S' be a new variable that is neither in V_1 nor in V_2.

Let $G_\cup = (V_1 \cup V_2 \cup \{S'\},\ \Sigma,\ R_1 \cup R_2 \cup \{r_0\},\ S')$, where r_0 is $S' \to S_1 \mid S_2$.

We construct CFG $G_\circ$ that generates $L(G_1) \circ L(G_2)$ as in $G_\cup$ by changing r_0 in $G_\cup$ into $S' \to S_1 S_2$.

To construct CFG G_* that generates the language $L(G_1)^*$, let S' be a new variable not in V_1 and make it the starting variable. Let r_0 be $S' \to S'S_1 \mid \varepsilon$ be a new rule in G_*.

2.16 Let A be a regular language generated by regular expression R. If R is one of the atomic regular expressions b, for $b \in \Sigma$, construct the equivalent CFG $(\{S\}, \{b\}, \{S \to b\}, S)$. If R is the atomic regular expression ε, construct the equivalent CFG $(\{S\}, \{b\}, \{S \to \varepsilon\}, S)$. If R is the atomic regular expressions $\emptyset$, construct the equivalent CFG $(\{S\}, \{b\}, \{S \to S\}, S)$.

If R is a regular expression composed of smaller regular expressions combined with a regular operation, use the result of Problem 2.15 to construct an equivalent CFG out of the CFLs that are equivalent to the smaller expressions.

2.17 **a.** Let C be a context-free language and R be a regular language. Let P be the PDA that recognizes C, and D be the DFA that recognizes R. If Q is the set of states of P and Q' is the set of states of D, we construct a PDA P' that recognizes $C \cap R$ with the set of states $P \times Q$. P' will do what P does and also keep track of the states of D. It accepts a string w if and only if it stops at a state $q \in F_P \times F_D$, where F_P is the set of accept states of P and F_D is the set of accept states of D. Since $C \cap R$ is recognized by P', it is context free.

b. Let R be the regular language $\mathtt{a^*b^*c^*}$. If A were A a CFL then $A \cap R$ would be a CFL by part (a). However, $A \cap R = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ and Example 2.20 proves $A \cap R$ is not context-free. Thus A is not a CFL.

2.18 **a.** Let $A = \{0^n1^n0^n1^n \mid n \geq 0\}$. Let p be the pumping length given by the pumping lemma. We show that $s = 0^p1^p0^p1^p$ cannot be pumped. Let $s = uvxyz$.

If either v or y contain more than one type of alphabet symbol, uv^2xy^2z does not contain the symbols in the correct order. Hence it cannot be a member of A. If both v and y contain (at most) one type of alphabet symbol, uv^2xy^2z contains runs of 0's and 1's of unequal length. Hence it cannot be a member of A. Because s cannot be pumped without violating the pumping lemma conditions, A is not context free.

b. Let $B = \{0^n\#0^{2n}\#0^{3n} \mid n \geq 0\}$. Let p be the pumping length given by the pumping lemma. Let $s = 0^p\#0^{2p}\#0^{3p}$. We show that $s = uvxyz$ cannot be pumped.

Neither v nor y can contain #, otherwise xv^2wy^2z contains more than two #s. Therefore, if we divide s into three segments by #'s: 0^p, 0^{2p}, and 0^{3p}, at least one of the segments is not contained within either v or y. Hence, xv^2wy^2z is not in B because the $1:2:3$ length ratio of the segments is not maintained.

c. Let $C = \{w\#x \mid w \text{ is a substring of } x, \text{ where } w, x \in \{\mathtt{a},\mathtt{b}\}^*\}$. Let p be the pumping length given by the pumping lemma. Let $s = \mathtt{a}^p\mathtt{b}^p\#\mathtt{a}^p\mathtt{b}^p$. We show that the string $s = uvxyz$ cannot be pumped.

Neither v nor y can contain #, otherwise uv^0xy^0z does not contain # and therefore is not in C. If both v and y are nonempty and occur on the left-hand side of the #, the string uv^2xy^2z cannot be in C because it is longer on the left-hand side of the #. Similarly, if both strings occur on the right-hand side of the #, the string uv^0xy^0z cannot be in C, because it is again longer on the left-hand side of the #. If only one of v and y is nonempty (both cannot be nonempty), treat them as if both occurred on the same side of the # as above.

The only remaining case is where both v and y are nonempty and straddle the #. But then v consists of b's and y consists of a's because of the third pumping lemma condition $|vxy| \leq p$. Hence, uv^2xy^2z contains more b's on the left-hand side of the #, so it cannot be a member of C.

d. Let $D = \{x_1\#x_2\#\cdots\#x_k \mid k \geq 2, \text{ each } x_i \in \{\mathtt{a},\mathtt{b}\}^*, \text{and for some } i \neq j, x_i = x_j\}$. Let p be the pumping length given by the pumping lemma. Let $s = \mathtt{a}^p\mathtt{b}^p\#\mathtt{a}^p\mathtt{b}^p$. We show that $s = uvxyz$ cannot be pumped. Use the same reasoning as in part (c).

2.19 **Note:** The version of the problem that appears in the first printing is incorrect. The correct version (which appears in the second and subsequent printings) is: Show that the language $F = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j, k \geq 0 \text{ and if } i = 1 \text{ then } j = k\}$ satisfies the three conditions of the pumping lemma even though it is not regular. Explain why this fact does not contradict the pumping lemma.

Every rule of a Chomsky normal form grammar is of the form

$$A \to BC$$
$$A \to \mathtt{a}$$

(except possibly for the rule $S \to \varepsilon$, where S is the start variable. This rule isn't used in the derivation of w because $n \geq 1$.)

Consider a derivation of w. Each application of a rule of the form $A \to BC$ increases the length of the string by 1. So we have $n - 1$ steps here. Besides that, we need exactly n applications of terminal rules ($A \to a$) to convert the variables into terminals. Therefore, exactly $2n - 1$ steps are required.

2.20 **Note:** The version of the problem that appears in the first printing is incorrect. The correct version (which appears in the second and subsequent printings) replaces the phrase "more than b steps" with the phrase "at least 2^b steps".

Let G be a CFG in Chomsky normal form that contains b variables. Assume G generates a string w using a derivation with at least 2^b steps. Let n be the length of w. By the results of Problem 2.20, $n \geq \frac{2^b+1}{2} > 2^{b-1}$.

Consider a parse tree of w. The right-hand side of each rule contains at most two variables, so each node of the parse tree has at most two children. Additionally, the length of w is at least 2^b, so the parse tree of w must have height at least $b+1$ to generate a string of length at least 2^b. Hence, the tree contains a path with at least $b+1$ variables, and therefore some variable is repeated on that path. Using a surgery on trees argument identical to the one used in the proof of the CFL pumping lemma, we can now divide w into pieces $uvxyz$ where $uv^ixy^iz \in G$ for all $i \geq 0$. Therefore, $L(G)$ is infinite.

2.21 **a.** To see that G is ambiguous, note that the string

`if condition then if condition then a:=1 else a:=1`

has two different leftmost derivations (and hence parse trees):

⟨STMT⟩
→ ⟨IF-THEN⟩
→ `if condition then` ⟨STMT⟩
→ `if condition then` ⟨IF-THEN-ELSE⟩
→ `if condition then if condition then` ⟨STMT⟩ `else` ⟨STMT⟩
→ `if condition then if condition then a:=1 else` ⟨STMT⟩
→ `if condition then if condition then a:=1 else a:=1`

and

⟨STMT⟩
→ ⟨IF-THEN-ELSE⟩
→ `if conditon then` ⟨STMT⟩ `else` ⟨STMT⟩
→ `if conditon then` ⟨IF-THEN⟩ `else` ⟨STMT⟩
→ `if conditon then if condition then` ⟨STMT⟩ `else` ⟨STMT⟩
→ `if condition then if condition then a:=1 else` ⟨STMT⟩
→ `if condition then if condition then a:=1 else a:=1`

b. The ambiguity in part a) arises because the grammar allows matching an `else` both to the nearest and to the farthest `then`. To avoid this ambiguity we construct a new grammar that only permits the nearest match, by disabling derivations which introduce an ⟨IF-THEN⟩ before an `else`. This grammar has two new variables: ⟨E-STMT⟩ and ⟨E-IF-THEN-ELSE⟩, which work just like their non-⟨E⟩ counterparts except that they cannot generate the dangling ⟨IF-THEN⟩. The rules of the new grammar are the same as for the old grammar except we remove the ⟨IF-THEN-ELSE⟩ rules and add the following new rules:

⟨E-STMT⟩ → ⟨ASSIGNMENT⟩ | ⟨BEGIN-END⟩ | ⟨E-IF-THEN-ELSE⟩
⟨E-IF-THEN-ELSE⟩ → `if conditon then` ⟨E-STMT⟩ `else` ⟨E-STMT⟩
⟨IF-THEN-ELSE⟩ → `if conditon then` ⟨E-STMT⟩ `else` ⟨STMT⟩

2.22 $L(G)$ is the set of strings of 0s and #s that either contain exactly two #s and any number of 0s, or contain exactly one # and the number of 0s to the right of the # is twice the number of 0s to the left. First we show that three is not a pumping length for this language. The string 0#00 has length at least three, and it is in $L(G)$. It cannot be pumped using $p = 3$, because the only way to divide it into $uvxyz$ satisfying the first two conditions of the pumping lemma is $u = z = \varepsilon$, $v = 0$, $x = \#$, and $y = 00$, but that division fails to satisfy the third condition.

Next, we show that 4 is a pumping length for $L(G)$. If $w \in L(G)$ has length at least 4 and if it contains two #s, then it contains at least one 0. Therefore, by cutting w into $uvxyz$ where either v or y is the string 0, we obtain a way to pump w. If $w \in L(G)$ has length at least 4 and if it contains a single #, then it must be of the form $0^k\#0^{2k}$ for some $k \geq 1$. Hence, by assigning $u = 0^{k-1}$, $v = 0$, $x = \#$, $y = 00$, and $z = 0^{2k-2}$, we satisfy all three conditions of the lemma.

2.23 Let $F = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\mathtt{d}^l \mid i, j, k, l \geq 0 \text{ and if } i = 1 \text{ then } j = k = l\}$. F is not context free because $F \cap \mathtt{ab}^*\mathtt{c}^*\mathtt{d}^* = \{\mathtt{ab}^n\mathtt{c}^n\mathtt{d}^n \mid n \geq 0\}$, which is not a CFL by the pumping lemma, and the intersection of a CFL and a regular language is a CFL (see Problem 2.17). However, F does satisfy the pumping lemma with $p = 2$. ($p = 1$ works, too, with a bit more effort.)

If $s \in F$ has length 2 or more then:

i) If $s \in \mathtt{b}^*\mathtt{c}^*\mathtt{d}^*$ write s as $s = rgt$ for $g \in \{b, c, d\}$ and divide $s = uvxyz$ where $u = r$, $v = g$, $x = y = \varepsilon$, $z = t$.

ii) If $s \in \mathtt{ab}^*\mathtt{c}^*\mathtt{d}^*$ write s as $s = \mathtt{a}t$ and divide $s = uvxyz$ where $u = \varepsilon$, $v = \mathtt{a}$, $x = y = \varepsilon$, $z = t$.

iii) If $s \in \mathtt{aaa}^*\mathtt{b}^*\mathtt{c}^*\mathtt{d}^*$ write s as $s = \mathtt{aa}t$ and divide $s = uvxyz$ where $u = \varepsilon$, $v = \mathtt{aa}$, $x = y = \varepsilon$, $z = t$.

In each of the three cases, we may easily check that the division of s satisfies the conditions of the pumping lemma.

2.24 Let G be a CFG generating A. Let p be the pumping length for G given by the pumping lemma for CFGs. Let $k = p! = p(p-1)(p-2)\cdots 1$. Let $s = \mathtt{a}^k\mathtt{b}^k\mathtt{c}^k$. We show that s has two distinct parse trees in G. To do so we'll also consider the following two strings in A, $s_1 = \mathtt{a}^k\mathtt{b}^p\mathtt{c}^p$ and $s_2 = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^k$ with parse trees τ_1 and τ_2.

In τ_1, remove all nodes that have only a's below them. The remaining subtree has $2p$ leaves that are only b's and c's. Because $2p > p$ we know that the subtree contains a path with a repeated has variable R, using the same argument that appears in the proof of the pumping lemma. Again by the same argument, use R to divide s into $s = uvxyz$, and R can be chosen so that $|vxy| \leq p$. Strings v and y can each only contains one type of symbol; otherwise uv^2xy^2z is not in A. Furthermore, y contains no a's, because R

was on a path to a b or c. Hence v must be b's and y must be c's and they must have an equal length l. Letting $d = k/l$ (which must be an integer because $l \leq p$ and $k = p!$) the string $vu^d xy^d = s$ has a parse tree where most b's have a parent node in common with c's but not with a's. Repeating this argument with τ_2 we obtain a parse tree for s where most b's have a parent node in common with a's but not with c's. Hence s has two distinct parse trees, so G is ambiguous.

2.25 The grammar generates all strings not of the form $\mathtt{a}^k\mathtt{b}^k$ for $k \geq 0$. Thus the complement of the language generated is $\overline{L(G)} = \{\mathtt{a}^k\mathtt{b}^k |\ k \geq 0\}$. The following grammar generates $\overline{L(G)}$: $\{\{S\}, \{\mathtt{a}, \mathtt{b}\}, \{S \to \mathtt{a}S\mathtt{b} \mid \varepsilon\}, S\}$.

2.26 Let $C = \{x\#y |\ x, y \in \{0,1\}^* \text{ and } x \neq y\}$. We construct a PDA P that recognizes C. It operates by guessing corresponding positions on which the strings x and y differ, as follows. It reads the input at the same time as it pushes some symbols, say 1s, onto the stack. At some point it nondeterministically guesses a position in x and it records the symbol it is currently reading there in its finite memory and skips to the #. Then it pops the stack while reading symbols from the input until the stack is empty and checks that the symbol it is now currently reading is different from the symbol it had recorded. If so, it accepts.

Here is a more detailed description of P's algorithm. If something goes wrong, for example, popping when the stack is empty, or getting to the end of the input prematurely, P rejects on that branch of the computation.

1. Read next input symbol, and push 1 onto stack.
2. Nondeterministically jump to either 1 or 3.
3. Record the current input symbol in the finite control. For convenience call this symbol a.
4. Read input symbols until # is read.
5. Read the next symbol, and pop the stack.
6. If stack is empty, go to 7, otherwise go to 5.
7. *Accept* if the current input symbol differs from a, otherwise *reject*.

2.27 Let $D = \{xy |\ x, y \in \{0,1\}^* \text{ and } |x| = |y| \text{ but } x \neq y\}$. We construct a PDA P recognizing D. This PDA guesses corresponding places on which x and y differ. Checking that the places correspond is tricky. Doing so relies on the observation that the two corresponding places are $n/2$ symbols apart, where n is the length of the entire input. Hence, by ensuring that the number of symbols between the guessed places is equal to the number other symbols, the PDA can check that the guessed places do indeed correspond,

Here we give a more detailed description of the PDA algorithm. If something goes wrong, for example, popping when the stack is empty, or getting to the end of the input prematurely, P rejects on that branch of the computation.

1. Read next input symbol and push 1 onto the stack.
2. Nondeterministically jump to either 1 or 3.
3. Record the current input symbol in the finite control. For convenience call this symbol a.
4. Read next input symbol and pop the stack. Repeat until stack is empty.
5. Read next input symbol and push 1 onto the stack.
6. Nondeterministically jump to either 5 or 7.
7. *Reject* if current input symbol differs from a.
8. Read next input symbol and pop the stack. Repeat until stack is empty.
9. *Accept* if input is empty.

Alternatively we can give a CFG for this language as follows.

$$\begin{aligned} S &\to AB \mid BA \\ A &\to XAX \mid 0 \\ B &\to XBX \mid 1 \\ X &\to 0 \mid 1 \end{aligned}$$

2.28 Let $G = (\Sigma, V, T, P, S)$ be a context free grammar for A. Define r to be the length of the longest string of symbols occurring on the right hand side of any production in G. We set the pumping length k to be $r^{2|V|+1}$. Let s be any string in A of length at least k and let T be a derivation tree for A with the fewest number of nodes. Observe that since $s \geq k$, the depth of T must be at least $2|V|+1$. Thus, some path p in T has length at least $2|V|+1$. By the pigeonhole principle, some variable V' appears at least three times in p. Label the last three occurrences of V' in p as V_1, V_2, and V_3. Moreover, define the strings generated by V_1, V_2, and V_3 in T as s_1, s_2, and s_3 respectively. Now, each of these strings is nested in the previous because they are all being generated on the same path. Suppose then that $s_1 = l_1 s_2 r_1$ and $s_2 = l_2 s_3 r_2$. We now have three cases to consider:

i) $|l_1 l_2| \geq 1$ and $|r_1 r_2| \geq 1$: In this case, there is nothing else to prove since we can simply pump in the usual way.

ii) $|l_1 l_2| = 0$: Since we defined T to be a minimal tree, neither r_1 nor r_2 can be the empty string. So, we can now pump r_1 and r_2.

iii) $|r_1 r_2| = 0$: This case is handled like the previous one.

Chapter 3

3.1 **a.** $q_1 0, \sqcup q_2 \sqcup, \sqcup\sqcup q_{\text{accept}}$

b. $q_1 00, \sqcup q_2 0, \sqcup \mathtt{x} q_3 \sqcup, \sqcup q_5 \mathtt{x} \sqcup, q_5 \sqcup \mathtt{x} \sqcup, \sqcup q_2 \mathtt{x} \sqcup, \sqcup \mathtt{x} q_2 \sqcup, \sqcup \mathtt{x} \sqcup q_{\text{accept}}$

c. $q_1 000, \sqcup q_2 00, \sqcup \mathtt{x} q_3 0, \sqcup \mathtt{x} 0 q_4 \sqcup, \sqcup \mathtt{x} 0 \sqcup q_{\text{reject}}$

d. $q_1000000, \sqcup q_200000, \sqcup \mathtt{x} q_30000 \sqcup, \sqcup \mathtt{x0} q_4000, \sqcup \mathtt{x0x} q_300, \sqcup \mathtt{x0x0} q_40,$
$\sqcup \mathtt{x0x0x} q_3 \sqcup, \sqcup \mathtt{x0x0} q_5 \mathtt{x} \sqcup, \sqcup \mathtt{x0x} q_5 \mathtt{0x} \sqcup, \sqcup \mathtt{x0} q_5 \mathtt{x0x} \sqcup, \sqcup \mathtt{x} q_5 \mathtt{0x0x} \sqcup, \sqcup q_5 \mathtt{x0x0x} \sqcup,$
$q_5 \sqcup \mathtt{x0x0x} \sqcup, \sqcup q_2 \mathtt{x0x0x} \sqcup, \sqcup \mathtt{x} q_2 \mathtt{0x0x} \sqcup, \sqcup \mathtt{xx} q_3 \mathtt{x0x} \sqcup, \sqcup \mathtt{xxx} q_3 \mathtt{0x} \sqcup, \sqcup \mathtt{xxx0} q_4 \mathtt{x} \sqcup,$
$\sqcup \mathtt{xxx0x} q_4 \sqcup, \sqcup \mathtt{xxx0x} \sqcup q_{\text{reject}}$

3.2

a. $q_111, \sqcup q_31; \sqcup 1 q_3 \sqcup, \sqcup 1 \sqcup q_{\text{reject}}$

b. $q_11\#1, \sqcup q_3\#1, \sqcup\#q_51, \sqcup\#1q_5\sqcup, \sqcup\#q_71\sqcup, \sqcup q_7\#1\sqcup, q_7\sqcup\#1\sqcup, \sqcup q_9\#1\sqcup, \sqcup\#q_{11}1\sqcup,$
$\sqcup q_{12}\#\mathtt{x}\sqcup, q_{12}\sqcup\#\mathtt{x}\sqcup, \sqcup q_{13}\#\mathtt{x}\sqcup, \sqcup\#q_{14}\mathtt{x}\sqcup, \sqcup\#\mathtt{x}q_{14}\sqcup, \sqcup\#\mathtt{x}\sqcup q_{\text{accept}}$

c. $q_11\#\#1, \sqcup q_3\#\#1, \sqcup\#q_5\#1, \sqcup\#\#q_{\text{reject}}1$

d. $q_110\#11, \sqcup q_30\#11, \sqcup 0q_3\#11, \sqcup 0\#q_511, \sqcup 0\#1q_51, \sqcup 0\#11q_5\sqcup, \sqcup 0\#1q_71\sqcup,$
$\sqcup 0\#q_711\sqcup, \sqcup 0q_7\#11\sqcup, \sqcup q_70\#11\sqcup, q_7\sqcup 0\#11\sqcup, \sqcup q_90\#11\sqcup, \sqcup 0q_9\#11\sqcup, \sqcup 0\#q_{11}11\sqcup,$
$\sqcup 0q_{12}\#\mathtt{x}1\sqcup, \sqcup q_{12}0\#\mathtt{x}1\sqcup, q_{12}\sqcup 0\#\mathtt{x}1\sqcup, \sqcup q_{13}0\#\mathtt{x}1\sqcup, \sqcup \mathtt{x}q_8\#\mathtt{x}1\sqcup, \sqcup \mathtt{x}\#q_{10}\mathtt{x}1\sqcup,$
$\sqcup \mathtt{x}\#\mathtt{x}q_{10}1\sqcup, \sqcup \mathtt{x}\#\mathtt{x}1q_{\text{reject}}\sqcup$

e. $q_110\#10, \sqcup q_30\#10, \sqcup 0q_3\#10, \sqcup 0\#q_510, \sqcup 0\#1q_50, \sqcup 0\#10q_5\sqcup, \sqcup 0\#1q_70\sqcup,$
$\sqcup 0\#q_710\sqcup, \sqcup 0q_7\#10\sqcup, \sqcup q_70\#10\sqcup, q_7\sqcup 0\#10\sqcup, \sqcup q_90\#10\sqcup, \sqcup 0q_9\#10\sqcup, \sqcup 0\#q_{11}10\sqcup,$
$\sqcup 0q_{12}\#\mathtt{x}0\sqcup, \sqcup q_{12}0\#\mathtt{x}0\sqcup, q_{12}\sqcup 0\#\mathtt{x}0\sqcup, \sqcup q_{13}0\#\mathtt{x}0\sqcup, \sqcup \mathtt{x}q_8\#\mathtt{x}0\sqcup, \sqcup \mathtt{x}\#q_{10}\mathtt{x}0\sqcup,$
$\sqcup \mathtt{x}\#\mathtt{x}q_{10}0\sqcup, \sqcup \mathtt{x}\#q_{12}\mathtt{xx}\sqcup, \sqcup \mathtt{x}q_{12}\#\mathtt{xx}\sqcup, \sqcup q_{12}\mathtt{x}\#\mathtt{xx}\sqcup, q_{12}\sqcup \mathtt{x}\#\mathtt{xx}\sqcup, \sqcup q_{13}\mathtt{x}\#\mathtt{xx}\sqcup,$
$\sqcup \mathtt{x}q_{13}\#\mathtt{xx}\sqcup, \sqcup \mathtt{x}\#q_{14}\mathtt{xx}\sqcup, \sqcup \mathtt{x}\#\mathtt{x}q_{14}\mathtt{x}\sqcup, \sqcup \mathtt{x}\#\mathtt{xx}q_{14}\sqcup, \sqcup \mathtt{x}\#\mathtt{xx}\sqcup q_{\text{accept}}$

3.3 If a language L is decidable, it can be decided by a deterministic TM and that is automatically a nondeterministic TM.

If a language L is decided by a nondeterministic TM N, we construct a deterministic TM D_2 that decides L. TM D_2 uses the same algorithm as in the TM D described in the proof of Theorem 3.10, with an additional step 5: *Reject* if all branches of nondeterminism of N are exhausted.

We argue that D_2 is a decider for L. If N accepts, D_2 will eventually find an accepting branch and accept, too. If N rejects, all of its branches halt and reject. Therefore, by the theorem on trees given in the exercise, its entire computation tree is finite, and D will halt and reject when this entire tree has been explored.

3.4 An enumerator is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{print}}, q_{\text{accept}})$, where Q, Σ, Γ are all finite sets and

i) Q is the set of states,

ii) Γ is the work tape alphabet,

iii) Σ is the output tape alphabet,

iv) $\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\} \times \Sigma_\varepsilon$ is the transition function,

v) $q_0 \in Q$ is the start state,

vi) $q_{\text{print}} \in Q$ is the print state, and

vii) $q_{\text{accept}} \in Q$ is the reject state, where $q_{\text{print}} \neq q_{\text{reject}}$.

The computation of an enumerator E is defined as in an ordinary TM, except for the following points. It has two tapes, a work tape and a print tape, both initially blank. At each step, the machine may write a symbol from Σ on the output tape, or nothing, as determined by δ. If $\delta(q, a) = (r, b, \mathrm{L}, c)$, it means that in state q, reading a, enumerator E enters state r, writes b on the work tape, moves the work tape head left (or right, if L had been R), writes c on the output tape, and moves the output tape head to the right if $c \neq \varepsilon$.

Whenever state q_{print} is entered, the output tape is reset to blank and the head returns to the left-hand end. The machine halts when q_{accept} is entered. $L(E) = \{w \in \Sigma^* | \ w$ appears on the work tape if q_{print} is entered$\}$.

3.5

a. Yes. Γ is the tape alphabet. A Turing Machine can write any characters in Γ on its tape, and $\sqcup \in \Gamma$ according to the definition.

b. No. Σ never contains $\sqcup$ but Γ always contains $\sqcup$. So, they cannot be equal.

c. Yes. If the Turing Machine tries to move its head off the left-hand end of the tape, it remains on the same tape cell.

d. No. Any Turing Machine must contain two distinct states q_{accept} and q_{reject}. So, a Turing Machine contains at least two states.

3.6 In Stage 2 of this algorithm: "Run M on s_i", if M loops on a certain input s_i, E would not check any inputs after s_i. If that were to occur, E might fail to enumerate $L(M)$ as required.

3.7 The variables $x_1, \ldots, x_k$ have infinitely may possible settings. A Turing machine would required infinite time to try them all. But, we require that every stage in the Turing machine description be completed in a finite number of steps.

3.8

a. "On input string w:

1. Scan the tape and mark the first 0 which has not been marked. If there is no unmarked 0, go to stage 4. Otherwise, move the head back to the front of the tape.
2. Scan the tape and mark the first 1 which has not been marked. If there is no unmarked 1, *reject*.
3. Move the head back to the front of the tape and repeat stage 1.
4. Move the head back to the front of the tape. Scan the tape to see if any unmarked 1s remain. If there are none, *accept*. Otherwise, *reject*."

b. "On input string w:

1. Scan the tape and mark the first 0 which has not been marked. If there is no unmarked 0, go to stage 4.

2. Move on and mark the next unmarked 0. If there is not any on the tape, *reject*. Otherwise, move the head back to the front of the tape.
3. Scan the tape and mark the first 1 which has not been marked. If there is no unmarked 1, *reject*.
4. Move the head back to the front of the tape and repeat stage 1.
5. Move the head back to the front of the tape. Scan the tape to see if there are any unmarked 1s. If there is not, *accept*. Otherwise, *reject*."

c. "On input string w:
1. Scan the tape and mark the first 0 which has not been marked. If there is no unmarked 0, go to stage 4.
2. Move on and mark the next unmarked 0. If there is not any on the tape, *accept*. Otherwise, move the head back to the front of the tape.
3. Scan the tape and mark the first 1 which has not been marked. If there is no unmarked 1, *accept*.
4. Move the head back to the front of the tape and repeat stage 1.
5. Move the head back to the front of the tape. Scan the tape to see if there are any unmarked 1s. If there is not, *reject*. Otherwise, *accept*."

3.9 a. By Example 2.20, no PDA recognizes $B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. The following 2-PDA recognizes B. Push all the a's that appear in the front of the input tape to stack 1. Then push all the b's that follow the a's in the input stream into stack 2. If it sees any a's at this stage, *reject*. When it sees the first c, pop stack 1 and stack 2 at the same time. After this, reject the input if it sees any a's or b's in the input stream. Pop stack 1 and stack 2 for each input character c it reads. If the input ends when both stacks are empty, *accept*. Otherwise, *reject*.

b. We show that a 2-PDA can simulate a TM. Furthermore, a 3-tape NTM can simulate a 3-PDA, and an ordinary deterministic, 1-tape TM can simulate a 3-tape NTM. Therefore a 2-PDA can simulate a 3-PDA, and so they are equivalent in power.

The TM by 2-PDA simulation is done as follows. We split the tape of a TM into two stacks. Stack 1 stores the characters on the left of the head, with the bottom of stack storing the leftmost character of the tape in the TM. Stack 2 stores the characters on the right of the head, with the bottom of the stack storing the rightmost character on the tape in the TM. In other words, if a TM configuration is aq_ib, the corresponding 2-PDA is in state q_i, with stack 1 storing the string a and stack 2 storing the string b^R, both from bottom to top.

For each transition $\delta(q_i, c_i) = (q_j, c_j, \mathrm{L})$ in the TM, the corresponding PDA transition pops c_i off stack 2, pushes c_j into stack 2, pops stack 1 and pushes the character into stack 2, and goes from state q_i to q_j. For any transition $\delta(q_i, c_i) = (q_j, c_j, \mathrm{R})$ in the TM, the corresponding PDA transition pops c_i off stack 2 and takes away c_i, push c_j into stack 1, and goes from state q_i to q_j.

3.10 We first simulate an ordinary TM by a write-twice TM. The write-twice TM simulates a single step of the original TM by copying the entire tape over to a fresh section of the tape to the right of the currently used section. The copying procedure operates character by character, marking a character as it is copied. This procedure alters each tape square twice, once to write the character for the first time and another time to mark that it is copied. The position of the original TMs tape head is marked on the tape. When copying the cells around the the marked position, the tape contents is updated according to the rules of the TM.

To carry out the simulation with a write-once TM, operate as before, except that each cell of the tape is represented by two cells. The first of these contains the original TM's tape contents and the second is for the mark used in the copying procedure. The input is not presented to the machine in the format with two cells per symbol, so the very first time the tape is copied, the copying marks are put directly over the input symbols.

3.11 A Turing machine with doubly infinite tape can easily simulate an ordinary Turing Machine. It needs to mark the left-hand end of the input so that it can prevent the head from moving off of that end.

To simulate the doubly infinite tape TM by an ordinary TM, we show how to simulate it with a 2-tape TM, which was already shown to be equivalent in power to an ordinary TM. The first tape of the 2-tape TM is written with the input string and the second tape is blank. We cut the tape of the doubly infinite tape TM into two parts, at the starting cell of the input string. The portion with the input string and all the blank spaces to its right appears on the first tape of the 2-tape TM. The portion to the left of the input string appears on the second tape, in reverse order.

3.12 We simulate an ordinary TM with a reset TM has only the RESET and R operations. When the ordinary TM moves its head right, the reset TM does the same. When the ordinary TM moves its head left, the reset TM cannot, so it gets the same effect by marking the current head location on the tape, then resetting and copying the entire tape one cell to the right, except for the mark, which is kept on the same tape cell. Then it resets again, and scans right until it find the mark.

3.13 We simulate this TM variant by an NFA. The NFA only needs to keep track of the TM's current state and what the TMs written on the current tape cell. Remembering what it has written on tape cells to the left of the current head position is unnecessary, because the TM is unable to return to these cells an read them. Using an NFA in the actual construction is convenient because it allows ε moves which are useful for simulating the "stay put" TM transitions.

More formally, the NFA has $|Q \times \Gamma| + 3$ states. The first $|Q \times \Gamma|$ states correspond to the state of the TM variant and the character on the tape that the head of the TM points to. The extra states are the start state q_{start}, a special accept state q_{accept} and a special reject state q_{reject}. The transition function δ' for the NFA is constructed according δ. First we set $\delta'(q_{\text{start}}, p) = \{q_{0p}\}$, where q_0 is the start state of the TM variant. Next, we set $\delta'(q_{\text{accept}}, i) = \{q_{\text{accept}}\}$ for any i.

If $\delta(p, a) = (q_{\text{accept}}, b, w)$, where $w =$ R or S, we set $\delta'(q_{pa}, \varepsilon) = \{q_{\text{accept}}\}$. If $\delta(p, a) = (q_{\text{reject}}, b, w)$, where $w =$ R or S, we set $\delta'(q_{pa}, \varepsilon) = \{q_{\text{reject}}\}$.

3.14 **a.** For any two decidable languages L_1 and L_2, let M_1 and M_2 be the TMs that decide them. We construct a TM M' that decides the union of L_1 and L_2:

"On input w:

1. Run M_1 on w, if it accepts, *accept*.
2. Run M_2 on w, if it accepts, *accept*. Otherwise, *reject*."

M' accepts w if either M_1 or M_2 accepts it. If both reject, M' rejects.

b. For any two decidable languages L_1 and L_2, let M_1 and M_2 be the TMs that decide them. We construct a NTM M' that decides the concatenation of L_1 and L_2:

"On input w:

1. For each way to cut w into two parts $w = w_1 w_2$:
2. Run M_1 on w_1.
3. Run M_2 on w_2.
4. If both accept, *accept*. Otherwise, continue with the next w_1, w_2.
5. All cuts have been tried without success, so *reject*."

We try every possible cut of w. If we ever come across a cut such that the first part is accepted by M_1 and the second part is accepted by M_2, w is in the concatenation of L_1 and L_2. So M' accept w. Otherwise, w does not belong to the concatenation of the languages and is rejected.

c. For any decidable language L, let M be the TM that decides it. We construct a NTM M' that decides the star of L:

"On input w:

1. For each way to cut w into parts so that $w = w_1 w_2 \ldots w_n$:
2. Run M on w_i for $i = 1, 2, \ldots, n$. If M accepts each of these string w_i, *accept*.
3. All cuts have been tried without success, so *reject*."

If there is a way to cut w into different substrings such that every substring is accepted by M, w belongs to the star of L and thus M' accepts w. Otherwise, w is rejected. Since there are finitely many possible cuts of w, M' will halt after finitely many steps.

d. For any decidable language L, let M be the TM that decides it. We construct a TM M' that decides the complement of L:

"On input w:

1. Run M on w. If M accepts, *reject*; if M rejects, *accept*."

Since M' does the opposite of whatever M does, it decides the complement of L.

e. For any two decidable languages L_1 and L_2, let M_1 and M_2 be the TMs that decide them. We construct a TM M' that decides the intersection of L_1 and L_2:

"On input w:

1. Run M_1 on w, if it rejects, *reject*.
2. Run M_2 on w, if it accepts, *accept*. Otherwise, *reject*."

M' accepts w if both M_1 and M_2 accept it. If either of them rejects, M' rejects w, too.

3.15 **a.** For any two Turing-recognizable languages L_1 and L_2, let M_1 and M_2 be the TMs that recognize them. We construct a TM M' that recognizes the union of L_1 and L_2:

"On input w:

1. Run M_1 and M_2 alternatively on w step by step. If either accept, *accept*. If both halt and reject, then *reject*."

If any of M_1 and M_2 accept w, M' will accept w since the accepting TM will come to its accepting state after a finite number of steps. Note that if both M_1 and M_2 reject and either of them does so by looping, then M will loop.

b. For any two Turing-recognizable languages L_1 and L_2, let M_1 and M_2 be the TMs that recognize them. We construct a NTM M' that recognizes the concatenation of L_1 and L_2:

"On input w:

1. Nondeterministically cut w into two parts $w = w_1w_2$.
2. Run M_1 on w_1. If it halts and rejects, *reject*. If it accepts, go to stage 3.
3. Run M_2 on w_2. If it accepts, *accept*. If it halts and rejects, *reject*."

If there is a way to cut w into two substrings such M_1 accepts the first part and M_2 accepts the second part, w belongs to the concatenation of L_1 and L_2 and M' will accept w after a finite number of steps.

c. For any Turing-recognizable language L, let M be the TM that recognizes it. We construct a NTM M' that recognizes the star of L:

"On input w:

1. Nondeterministically cut w into parts so that $w = w_1 w_2 \cdots w_n$.
2. Run M on w_i for all i. If M accepts all of them, *accept*. If it halts and rejects any of them, *reject*."

If there is a way to cut w into substrings such M accepts the all the substrings, w belongs to the star of L and M' will accept w after a finite number of steps.

d. For any two Turing-recognizable languages L_1 and L_2, let M_1 and M_2 be the TMs that recognize them. We construct a TM M' that recognizes the intersection of L_1 and L_2:

"On input w:

1. Run M_1 on w. If it halts and rejects, *reject*. If it accepts, go to stage 3.
2. Run M_2 on w. If it halts and rejects, *reject*. If it accepts, *accept*."

If both of M_1 and M_2 accept w, w belongs to the intersection of L_1 and L_2 and M' will accept w after a finite number of steps.

3.16 If A is decidable, the enumerator operates by generating the strings in lexicographic order and testing each in turn for membership in A using the decider. Those strings which are found to be in A are printed.

If A is enumerable in lexicographic order, we consider two cases. If A is finite, it is decidable because all finite languages are decidable. If A is infinite, a decider for A operates as follows. On receiving input w, the decider enumerates all strings in A in order until some string lexicographically after w appears. That must occur eventually because A is infinite. If w has appeared in the enumeration already then *accept*, but if it hasn't appeared yet, it never will, so *reject*.

Note: Breaking into two cases is necessary to handle the possibility that the enumerator may loop without producing additional output when it is enumerating a finite language. As a result, we end up showing that the language is decidable but we do not (and cannot) algorithmically construct the decider for the language from the enumerator for the language. This subtle point is the reason for the star on the problem.

3.17 We give a DFA A that is equivalent to TM M. Design A in two stages. First, we arrange the states and transitions of A to simulate M on the read-only input, portion of the tape. Second, we assign the accept states of A to correspond to the action of M on the read/write portion of the tape to the right of the input. In the first stage, we construct A to store a function

$$F_s : (Q \cup \{\text{first}\}) \longrightarrow (Q \cup \{\texttt{acc}, \texttt{rej}\})$$

in its finite control where Q is the set of M's states and s is the string that it has read so far. Note that only finitely many such functions exist, so we assign one state for each possibility.

The function F_s contains information about the way M would compute on a string s. In particular, $F_s(\text{first})$ is the state that M enters when it is about to move off of the right end of s for the first time, when M is started in its starting state at the left end of s. If M accepts or rejects (either explicitly or by looping) before ever moving off of the right end of s, then $F_s(\text{first}) = \texttt{acc}$ or $\texttt{rej}$ accordingly. Furthermore, for $F_s(q)$ is the state M enters when it is about to move off of the right end of s for the first time, when M is started in state q at the right end of s. As before, if M accepts or rejects before ever moving off of the right end of s, then $F_s(q) = \texttt{acc}$ or $\texttt{rej}$.

Observe that, for any string s and symbol a in Σ, we can determine F_{sa} from a, F_s, and M's transition function. Furthermore, F_ε is predetermined because $F_\varepsilon(\text{first}) = q_0$, the start state of M, and $F_\varepsilon(q) = q$ (because TMs that attempt to move off the leftmost end of the tape just stay in the left-hand cell). Hence we can obtain A's start state and transition function. That completes the first stage of A's design.

To complete the second stage, we assign the accept states of M. For any two strings s and t, if $F_s = F_t$, then M must have the same output on s and t—either both s and t are in $L(M)$ or both are out. Hence A's state at the end of s is enough to determine whether M accepts s. More precisely, we declare a state of A to be accepting if it was assigned to a function F that equals F_s for any string s that M accepts.

3.19 The language A is one of the two languages, $\{\texttt{0}\}$ or $\{\texttt{1}\}$. In either case the language is finite, and hence decidable. If we are not able to determine which of these two languages is A, we won't be able to describe the decider for A, but we can give two TMs, one of which is A's decider.

Chapter 4

4.1
a. Yes
b. No
c. No
d. No
e. No
f. Yes

4.2 Let $EQ_{\text{DFA,REX}} = \{\langle A, R\rangle \mid A \text{ is a DFA}, R \text{ is a regular expression and } L(A) = L(R)\}$. The following TM E decides $EQ_{\text{DFA,REX}}$.

E = "On input $\langle A, R\rangle$:

1. Convert regular expression R to an equivalent DFA B using the procedure given in Theorem 1.28.
2. Use the TM C for deciding EQ_{DFA} given in Theorem 4.5, on input $\langle A, B\rangle$.

3. If R accepts, *accept*. If R rejects, *reject*."

4.3 Let $ALL_{\text{DFA}} = \{\langle A \rangle | \ A \text{ is a DFA that recognizes } \Sigma^*\}$. The following TM L decides ALL_{DFA}.

L = "On input $\langle A \rangle$ where A is a DFA:

1. Construct DFA B that recognizes $\overline{L(A)}$ as described in Exercise 1.10.
2. Run TM T from Theorem 4.4 on input $\langle B \rangle$, where T decides E_{DFA}.
3. If T accepts, *accept*. If T rejects, *reject*."

4.4 Let $A\varepsilon_{\text{CFG}} = \{\langle G \rangle | \ G \text{ is a CFG that generates } \varepsilon\}$. The following TM V decides $A\varepsilon_{\text{CFG}}$.

V = "On input $\langle G \rangle$ where G is a CFG:

1. Run TM S from Theorem 4.6 on input $\langle G, \varepsilon \rangle$, where S is a decider for A_{CFG}.
2. If S accepts, *accept*. If S rejects, *reject*."

4.5 Let $INFINITE_{\text{DFA}} = \{\langle A \rangle | \ L(A) \text{ is an infinite language}\}$. The following TM I decides $INFINITE_{\text{DFA}}$.

I = "On input $\langle A \rangle$ where A is a DFA:

1. Let k be the number of states of A.
2. Construct a DFA D that accepts strings of length $\geq k$.
3. Construct a DFA M such that $L(M) = L(A) \cap L(D)$.
4. Run TM T from Theorem 4.4 on input $\langle M \rangle$, where T decides E_{DFA}.
5. If T accepts, *reject*. If T rejects, *accept*."

If A accepts a string with length at least k, this string can be must cause A to enter the same state at least twice, so it can be pumped up to obtain infinitely many strings that are accepted by A. Conversely, if A accepts infinitely many strings, it must accept strings of arbitrarily long lengths, in particular, a string of length at least k.

4.6

a. g is one-to-one. f is not one-to-one because $f(1) = f(3)$.

b. g is onto. f is not onto because there does not exist $x \in X$ such that $f(x) = 10$.

c. g is a correspondence because g is one-to-one and onto. f is not a correspondence because f is not one-to-one and onto.

4.7 Suppose $\mathcal{B}$ is countable and a correspondence $f: \mathcal{N} \longrightarrow \mathcal{B}$ exists. We construct x in $\mathcal{B}$ that is not paired with anything in $\mathcal{N}$. Let $x = x_1 x_2 \ldots$. Let $x_i = 0$ if $f(i)_i = 1$, and $x_i = 1$ if $f(i)_i = 0$ where $f(i)_i$ is the ith bit of $f(i)$. Therefore, we ensure that x is not $f(i)$ for any i because it differs from $f(i)$ in the ith symbol, and a contradiction occurs.

4.8 We demonstrate a one-to-one $f: T \longrightarrow \mathcal{N}$. Let $f(i,j,k) = 2^i 3^j 5^k$. Function f is one-to-one because if $a \neq b$, $f(a) \neq f(b)$. Therefore, T is countable.

4.9 Let $\sim$ be the binary relation "is the same size". In other words If $A \sim B$, A and B are the same size. We show that $\sim$ is an equivalence relation. First, $\sim$ is reflexive because the identity function $f(x) = x, \forall x \in A$ is a correspondence $f: A \longrightarrow A$. Second, $\sim$ is symmetric because any correspondence has an inverse, which itself is a correspondence. Third, $\sim$ is transitive because if $A \sim B$ via correspondence f, and $B \sim C$ via correspondence g, then $A \sim C$ via correspondence $f \circ g$ (the composition of f and g). Because $\sim$ is reflexive, symmetric, and transitive, $\sim$ is an equivalence relation.

4.10 The following TM X decides A.

X = "On input $\langle M \rangle$ where M is a DFA:

1. Construct a DFA O that accepts any string containing an odd number of 1s.
2. Construct DFA B such that $L(B) = L(M) \cap L(O)$.
3. Run TM T from Theorem 4.4 on input $\langle B \rangle$, where T decides E_{DFA}.
4. If T accepts, *accept*. If T rejects, *reject*."

4.11 We observe that $L(R) \subseteq L(S)$ if and only if $\overline{L(S)} \cap L(R) = \emptyset$. The following TM X decides A.

X = "On input $\langle R, S \rangle$ where R and S are regular expressions:

1. Construct DFA E such that $L(E) = \overline{L(S)} \cap L(R)$.
2. Run TM T from Theorem 4.4 on input $\langle E \rangle$, where T decides E_{DFA}.
3. If T accepts, *accept*. If T rejects, *reject*."

4.12 We have shown in Problem 2.17 that if C is a context-free language and R is a regular language, then $C \cap R$ is context free. Therefore $1^* \cap L(G)$ is context free. The following TM X decides A.

X = "On input $\langle G \rangle$ where G is a CFG:

1. Construct CFG H such that $L(H) = 1^* \cap L(G)$.
2. Run TM R from Theorem 4.7 on input $\langle H \rangle$, where R decides E_{CFG}.

3. If R accepts, *reject*. If T rejects, *accept*."

4.14 The following TM X decides A.

X = "On input $\langle R \rangle$ where R is a regular expression:

1. Construct DFA E that accepts $\Sigma^*111\Sigma^*$.
2. Construct DFA B such that $L(B) = L(R) \cap L(E)$.
3. Run TM T from Theorem 4.4 on input $\langle B \rangle$, where T decides E_{DFA}.
4. If T accepts, *reject*. If T rejects, *accept*."

4.15 Let $A = \{v |\ v \text{ is of the form } ww^{\mathcal{R}} \text{ for } w \in \{0,1\}^*\}$. A is a CFL. Problem 2.17 showed that $B \cap A$ is context free if B is regular. The following TM X decides E.

X = "On input $\langle M \rangle$ where M is a DFA:

1. Let $C = L(M) \cap A$. Let G be the CFG of C.
2. Run TM R from Theorem 4.7 on input $\langle G \rangle$, where R decides E_{CFG}.
3. If R accepts, *reject*. If R rejects, *accept*."

X decides E because a DFA M accepts some string of the form $ww^{\mathcal{R}}$ if and only if $L(M) \cap A \neq \emptyset$.

4.16 For any DFAs A and B, $L(A) = L(B)$ if and only if A and B accept the same strings up to length mn, where m and n are the numbers of states of A and B, respectively. Obviously, if $L(A) = L(B)$, A and B will accept the same strings. If $L(a) \neq L(B)$, we need to show that the languages differ on some string s of length at most mn. Let t be a shortest string on which the languages differ. Let l be the length of t. If $l \leq mn$ we are done. If not, consider the sequence of states $q_0, q_1, \ldots, q_l$ that A enters on input t, and the sequence of states $r_0, r_1, \ldots, r_l$ that B enters on input t. Because A has m states and B has n states, only mn distinct pairs (q, r) exist where q is a state of A and r is a state of B. By the pigeon hole principle, two pairs of states (q_i, r_i) and (q_j, r_j) must be identical. If we remove the portion of t from i to $j - 1$ we obtain a shorter string on which A and B behave as they would on t. Hence we have found a shorter string on which the two languages differ, even though t was supposed to be shortest. Hence t must be no longer than mn.

4.17 We need to prove both directions. To handle the easier one first, assume that D exists. A TM recognizing C operates on input x by going through each possible string y and testing whether $\langle x, y \rangle \in D$. If such a y is ever found, *accept*; if not, just continue searching.

For the other direction, assume that C is recognized by TM M. Define a language B to be $\{\langle x, y\rangle |\ M$ accepts x within $|y|$ steps$\}$. Language B is decidable, and if $x \in C$ then M accepts x within some number of steps, so $\langle x, y\rangle \in B$ for any sufficiently long y, but if $x \notin C$ then $\langle x, y\rangle \notin C$ for any y.

4.18 Let A and B be two languages such that $A \cap B = \emptyset$, and $\overline{A}$ and $\overline{B}$ are recognizable. Let J be the TM recognizing $\overline{A}$ and K be the TM recognizing $\overline{B}$. We will show that the language decided by TM T separates A and B.

T = "On input w:

1. Simulate J and K on w by alternating the steps of the two machines.
2. If J accepts first, *reject*. If K accepts first, *accept*."

The algorithm T terminates because $\overline{A} \cup \overline{B} = \Sigma^*$. So either J or K will accept w eventually. $A \subseteq C$ because if $w \in A$, w will not be recognized by J and will be accepted by K first. $B \subseteq \overline{C}$ because if $w \in B$, w will not be recognized by K and will be accepted by J first. Therefore, C separates A and B.

4.19 For any language A, let $A^{\mathcal{R}} = \{w^{\mathcal{R}} |\ w \in A\}$. If $\langle M\rangle \in S$, then $L(M) = L(M)^{\mathcal{R}}$. The following TM T decides S.

T = "On input $\langle M\rangle$, where M is a DFA:

1. Construct DFA N that recognizes $L(M)^{\mathcal{R}}$.
2. Run TM F from Theorem 4.5 on $\langle M, N\rangle$, where F is the Turing machine deciding EQ_{DFA}.
3. If F accepts, *accept*. If F rejects, *reject*."

4.20 Let $U = \{\langle P\rangle |\ P$ is a PDA that has useless states$\}$. The following TM T decides U.

T = "On input $\langle P\rangle$, where U is a PDA:

1. For each state q of P:
2. Modify P so that q is the only accept state.
3. Use the decider for E_{PDA} to test whether the modified PDA's language is empty. If it is, *accept*. If it is not, continue.
4. At this point, all states have been shown to be useful, so *reject*."

4.21 Use the diagonalization method to obtain D. Let $\langle M_1\rangle, \langle M_2\rangle, \ldots$ be the output of an enumerator for A. Let $\Sigma = \{s_1, s_2, \ldots\}$ be a list of all strings over the alphabet. The algorithm for D is:

D = "On input w:

1. Let i be the index of w on the list of all strings, that is, $s_i = w$.
2. Run $\langle M_i\rangle$ on input w.

3. If it accepts, *reject*. If it rejects, *accept*."

D is a decider because each M_i is a decider. But D doesn't appear on the enumeration for A, because it differs from M_i on input s_i.

4.22 Let K be the enumerator for B. The following TM L is the lexicographic-order enumerator for C.

L = "Ignore the input.

1. Enumerate machines M_i by K.
2. For each M_i, insert useless states, symbols or transitions to obtain N_i such that $L(M_i) = L(N_i)$ and $\langle N_i \rangle$ is lexicographically larger than all $\langle N_j \rangle$ for $1 \leq j < i$. Output N_i."

Let $\langle N_1 \rangle, \langle N_2 \rangle \ldots$ be the strings enumerated by L. Since $L(N_i) = L(M_i)$ for all i, every machine described in B has an equivalent one in C and vice versa. C is decidable because it enumerated by L in lexicographic order, by virtue of Problem 3.16.

Chapter 5

5.1 Suppose for a contradiction that EQ_{CFG} were decidable. We construct a decider M for $ALL_{\mathsf{CFG}} = \{\langle G \rangle | \ G \text{ is a CFG and } L(G) = \Sigma^*\}$ as follows:

M = "On input $\langle G \rangle$:

1. Construct a CFG H such that $L(H) = \Sigma^*$.
2. Run the decider for EQ_{CFG} on $\langle G, H \rangle$.
3. If it accepts, *accept*. If it rejects, *reject*."

M decides ALL_{CFG} assuming a decider for EQ_{CFG} exists. Since we know ALL_{CFG} is undecidable, we have a contradiction.

5.2 Here is a Turing Machine M which recognizes the complement of EQ_{CFG}:

M = "On input $\langle G, H \rangle$:

1. Lexicographically generate the strings $x \in \Sigma^*$.
2. For each such string x:
3. Test whether $x \in L(G)$ and whether $x \in L(H)$, using the algorithm for A_{CFG}.
4. If one of the tests accepts and the other rejects, *accept*; otherwise, continue."

5.3 Here is a match: $\left[\frac{\mathtt{ab}}{\mathtt{abab}}\right]\left[\frac{\mathtt{ab}}{\mathtt{abab}}\right]\left[\frac{\mathtt{aba}}{\mathtt{b}}\right]\left[\frac{\mathtt{b}}{\mathtt{a}}\right]\left[\frac{\mathtt{b}}{\mathtt{a}}\right]\left[\frac{\mathtt{aa}}{\mathtt{a}}\right]\left[\frac{\mathtt{aa}}{\mathtt{a}}\right]$.

5.4 No, it does not imply that A is regular. For example, $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\} \leq_{\mathrm{m}} \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$. The reduction first tests whether its input is a member of $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. If so, it outputs the string $\mathtt{ab}$, and if not, it outputs the string $\mathtt{a}$.

5.5 Suppose for a contradiction that $A_{\mathsf{TM}} \leq_{\mathrm{m}} E_{\mathsf{TM}}$ via reduction f. It follows from the definition of mapping reducibility that $\overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} \overline{E_{\mathsf{TM}}}$ via the same reduction function f. Observe that $\overline{E_{\mathsf{TM}}}$ is Turing-recognizable, but, $\overline{A_{\mathsf{TM}}}$ is not Turing-recognizable, contradicting Theorem 5.22.

5.6 Suppose $A \leq_{\mathrm{m}} B$ and $B \leq_{\mathrm{m}} C$. Then there are computable functions f and g such that $x \in A \iff f(x) \in B$ and $y \in B \iff g(y) \in C$. Consider the function composition $h(x) = g(f(x))$. We can build a TM which computes h as follows: first simulate a TM for f (such a TM exists since we assumed that f is computable) on input x and call the output y. Then simulate a TM for g on y. The output is $h(x) = g(f(x))$. Therefore, h is a computable function. Moreover, $x \in A \iff h(x) \in C$. Therefore, $A \leq_{\mathrm{m}} C$ via the reduction function h.

5.7 Suppose $A \leq_{\mathrm{m}} \overline{A}$. Then, $\overline{A} \leq_{\mathrm{m}} A$, by the definition of mapping reducibility. Because A is Turing-recognizable, Theorem 5.22 implies that $\overline{A}$ is Turing-recognizable, and then Theorem 4.16 implies that A is decidable.

5.8 We need to handle the case where the head is at the leftmost tape call and attempts to move left. To do so we add dominos

$$\left[\frac{\#qa}{\#rb}\right]$$

For every $q, r \in Q$ and $a, b \in \Gamma$, where $\delta(q, a) = (r, b, \mathrm{L})$.

5.9 Suppose L is a Turing-recognizable language and let M be a TM that recognizes it. To reduce L to A_{TM} we map any string x to $\langle M, x\rangle$. Then $x \in L \iff \langle M, x\rangle \in A_{\mathsf{TM}}$. In addition, this mapping is computable, so it gives a mapping reduction from L to A_{TM}.

5.10 To prove that J is not Turing-recognizable, we show that $\overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} J$. The reduction function maps any string y to the string $1y$. Then $y \in \overline{A_{\mathsf{TM}}} \iff 1y \in J$. This mapping is computable, so we have shown that $\overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} J$. To show that $\overline{J}$ is not Turing-recognizable we show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} J$ (since $A_{\mathsf{TM}} \leq_{\mathrm{m}} J \iff \overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} \overline{J}$). The reduction function maps any string x to $0x$. Then $x \in A_{\mathsf{TM}} \iff 0x \in J$. This mapping is computable, so we have shown that $A_{\mathsf{TM}} \leq_{\mathrm{m}} J$.

5.11 Use the language J from Problem 5.10. That problem showed that J is undecidable. Here we show that $J \leq_m \overline{J}$. The reduction f maps ϵ to itself and for other strings x,

$$f(x) = \begin{cases} 0s & \text{if } x = 1s \text{ for some string } s \in \Sigma^* \\ 1s & \text{if } x = 0s \text{ for some string } s \in \Sigma^* \end{cases}$$

The function f is computable, and $x \in J \iff f(x) \in \overline{J}$. Thus, $J \leq_m \overline{J}$.

5.12 We show that $A_{\mathsf{TM}} \leq_m S$ by mapping $\langle M, w\rangle$ to $\langle M'\rangle$ where M' is the following TM:

M' = "On input x:

1. If $x = 01$ then *accept*.
2. If $x \neq 10$ then *reject*.
3. If $x = 10$ simulate M on w. If M accepts w then *accept*; if M halts and rejects w then *reject*."

If $\langle M, w\rangle \in A_{\mathsf{TM}}$ then M accepts w and $L(M') = \{01, 10\}$, so $\langle M'\rangle \in S$. Conversely, if $\langle M, w\rangle \notin A_{\mathsf{TM}}$ then $L(M') = \{01\}$, so $\langle M'\rangle \notin S$. Therefore, $\langle M, w\rangle \in A_{\mathsf{TM}} \iff \langle M'\rangle \in S$.

5.13 Let $USELESS_{\mathsf{TM}} = \{\langle M\rangle |\ M$ is a TM with 1 or more useless states$\}$. We show $USELESS_{\mathsf{TM}}$ is undecidable by reducing E_{TM} to it. Let R be a TM that decides $USELESS_{\mathsf{TM}}$ and construct a TM S that decides E_{TM} as follows.

S = "On input $\langle M, w\rangle$:

1. First we construct a TM N that simulates M on w but ensuring that all of M's original states are not useless. The TM N has input alphabet $\{0, 1, 2\}$. On input 0, N goes through each of the original states of M, except for q_{accept} and q_{reject}, until all nonhalting states of M have been used, and then enters state q_{accept}. On input 1, N enters state q_{reject}. On input 2, N simulates M on w. Finally N restores the original symbol to where the new symbol had been written, and branches to M's start state to simulate M. If that simulation is about to enter M's accept state, it instead enters a new state r.
2. Run $USELESS_{\mathsf{TM}}$ on input $\langle N\rangle$. If it accepts, *reject*. If it rejects, *accept*."

The only possible useless state of N is the state r, because on inputs 0 and 1 it uses all other states. The only way for N to use state r, is if M accepts w. So, testing whether N has any useless states, in effect tests whether M accepts w. Note that the hardest part of this problem is making sure that the machine N has no "unexpected" useless states.

5.14 $L_{\mathsf{TM}} = \{\langle M, w\rangle |\ M$ on w tries moving its head left when on the leftmost tape cell$\}$. This language is undecidable. To prove it, we show that if it was decidable by some machine say R then we could decide A_{TM} with the following machine S:

S = "On input $\langle M, w\rangle$:

1. Convert M to M', where M' first moves its input over one square to the right and puts a special delimiter on the leftmost tape cell. Then M' simulates M on the input. If M' ever sees a delimiter (M must have tried to move left when on the leftmost tape cell), M' automatically moves right and stays in the same state. If M accepts, it moves its head all the way to the left and then moves left off the leftmost tape cell.
2. Run R, the decider for L_{TM}, on $\langle M', w\rangle$.
3. If R accepts then *accept*. If it rejects, *reject*."

TM S decides A_{TM} since the only way M' moves left from the leftmost tape cell is when M accepts w. S never attempts the move during the course of the simulation because we put the special delimiter to "intercept" such moves if made by M.

5.15 Let $LM_{\mathsf{TM}} = \{\langle M, w\rangle |\ M$ ever moves left while computing on $w\}$. LM_{TM} is decidable. Let M_{LEFT} be the TM which on input $\langle M, w\rangle$ determines the number of states $n_M = |Q_M|$ of M and then simulates M on w for $|w|+n_M+1$ steps. If M_{LEFT} discovers that M moves left during that simulation, M_{LEFT} accepts $\langle M, w\rangle$. Otherwise M_{LEFT} rejects $\langle M, w\rangle$.

The reason that M_{LEFT} can reject without simulating M further is as follows. Suppose M does make a left move on input w. Let $p = q_0, q_1, \ldots, q_s$ be the shortest computation path of M on w ending in a left move. Because M has been scanning only blanks (it's been moving right) since state $q_{|w|}$, we may remove any cycles that appear after this state and be left with a legal computation path of the machine ending in a left move. Hence p has no cycles and must have length at most $|w|+n_M+1$. Hence M_{LEFT} would have accepted $\langle M, w\rangle$, as desired.

5.16 Let $B = \{\langle M\rangle |\ M$ is a two tape TM which writes a non-blank symbol on its second tape when it is simulated on some particular input $w\}$. We show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} B$ by mapping $\langle M, w\rangle$ to $\langle M'\rangle$ where M' has the following description:

M' = "On input x:

1. Simulate M on w only using the first tape.
2. If M accepts w then write `blah blah` ! on the second tape."

Observe that $\langle M, w\rangle \in A_{\mathsf{TM}} \iff M' \in B$.

5.17 The *PCP* over a unary alphabet is decidable. We describe a TM M that decides unary *PCP*. Given a unary *PCP* instance

$$\left\{\left[\frac{1^{a_1}}{1^{b_1}}\right], \ldots, \left[\frac{1^{a_n}}{1^{b_n}}\right]\right\}$$

M = "On input $\langle a_1, b_1, \ldots, a_n, b_n\rangle$:

1. Check if $a_i = b_i$ for some i. If so, *accept*.
2. Check if there exist i, j such that $a_i > b_i$ and $a_j < b_j$. If so, *accept*. Otherwise, *reject*."

In the first stage, M first checks for a single domino which forms a solution. In the second stage, M looks for two dominos which form a solution. If it finds such a pair, it can construct a solution by picking $(b_j - a_j)$ pieces of the i^{th} domino, putting them together with $(a_i - b_i)$ pieces of the j^{th} domino. This construction has $a_i(b_j - a_j) + a_j(a_i - b_i) = a_i b_j - a_j b_i$ 1s on the top, and $b_i(b_j - a_j) + b_j(a_i - b_i) = a_i b_j - a_j b_i$ 1s on the bottom. If neither stages of M accept, the problem instance contains dominos with all the upper parts having more (or less) 1s than the lower parts. In such a case no solution can exist. Therefore M rejects.

5.18 We construct a computable function that takes a *PCP* instance of any finite alphabet and produces a binary alphabet *PCP* (*BPCP*) instance. As a result we show that *BPCP* is undecidable.

For any *PCP* instance A over a finite alphabet $\{a_1, \ldots, a_n\}$, the function encodes each character a_i into binary as 10^i. The function is a mapping reduction from the general *PCP* problem to the *BPCP* problem. For any P in *PCP*, the converted instance P' is in *BPCP* because a match for P becomes a match for P' using the same set of dominos after conversion. Conversely, if P' is in *BPCP*, P is in *PCP*, because we can uniquely decode a match in P' to a match in P. Therefore, the function is a mapping reduction from *PCP* to *BPCP*.

5.19 We reduce *PCP* to $AMBIG_{\mathsf{CFG}}$, thereby proving that $AMBIG_{\mathsf{CFG}}$ is undecidable. We use the construction given in the hint in the text.

Note: The hint that appears in the first printing is incorrect. The correct version (which appears in the second and subsequent printings) is: Given an instance

$$P = \left\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \cdots, \left[\frac{t_k}{b_k}\right]\right\},$$

$$\begin{aligned} S &\to T \mid B \\ T &\to t_1 T \mathtt{a}_1 \mid \cdots \mid t_k T \mathtt{a}_k \mid t_1 \mathtt{a}_1 \mid \cdots \mid t_k \mathtt{a}_k \\ B &\to b_1 B \mathtt{a}_1 \mid \cdots \mid b_k B \mathtt{a}_k \mid b_1 \mathtt{a}_1 \mid \cdots \mid b_k \mathtt{a}_k \ , \end{aligned}$$

where $\mathtt{a}_1, \ldots, \mathtt{a}_k$ are new terminal symbols. We prove that this reduction works.

We show that P has a solution iff the CFG is ambiguous. If P has a match $t_{i_1}t_{i_2}\ldots t_{i_l} = b_{i_1}b_{i_2}\ldots b_{i_l}$, the string

$$t_{i_1}t_{i_2}\ldots t_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1} = b_{i_1}b_{i_2}\ldots b_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$$

has two different leftmost derivations, one from T and one from B. Hence the CFG is ambiguous.

Conversely, if the CFG is ambiguous, some string w has multiple leftmost derivations. All generated strings have the form $w = w_{start}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$ where w_{start} contains only symbols from P's alphabet. Notice that once we choose the first step in the derivation of w (either $S \to T$ or $S \to B$), the following steps in the derivation are uniquely determined by the sequence $\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$. Therefore w has at most two leftmost derivations:

i) $S \to T \to t_{i_1}T\mathtt{a}_{i_1} \to t_{i_1}t_{i_2}T\mathtt{a}_{i_2}\mathtt{a}_{i_1} \to \cdots \to t_{i_1}t_{i_2}\ldots t_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$, and

ii) $S \to B \to b_{i_1}T\mathtt{a}_{i_1} \to b_{i_1}b_{i_2}T\mathtt{a}_{i_2}\mathtt{a}_{i_1} \to \cdots \to b_{i_1}b_{i_2}\ldots b_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$.

If w is ambiguous, $t_{i_1}t_{i_2}\ldots t_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1} = b_{i_1}b_{i_2}\ldots b_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$, and therefore $t_{i_1}t_{i_2}\ldots t_{i_l} = b_{i_1}b_{i_2}\ldots b_{i_l}$. Thus, the initial *PCP* instance, P, has a match.

5.20 **a.** A 2DFA M with s states computing on an input x of size n has at most $s(n+2)^2$ possible configurations. Thus, on input x, M either halts in $s(n+2)^2$ steps or loops forever. To decide A_{2DFA}, it is enough to simulate M on x for $s(n+2)^2$ steps and accept if M has halted and accepted x during this finite duration.

b. Suppose a TM D decides E_{2DFA}. We construct a TM E that decides E_{TM}. E runs as follows:

E = "On input $\langle M\rangle$,

1. Construct a 2DFA M' that recognizes the language of accepting computation histories on M, namely, $\{\langle c_1\#c_2\#\ldots\#c_k\rangle |\ c_i$ is a configuration of M, c_1 is a start configuration, c_k is an accepting configuration, c_{i+1} legally follows c_i for each $i\}$. A 2DFA can check whether c_1 is a start configuration of M by verifying that it begins with a start state q_0 and contains legal symbols from M's input alphabet. Similarly, it can check whether c_k is an accepting configuration by verifying that it contains q_{accept} and symbols from M's tape alphabet. These two steps use only one head. To check whether c_{i+1} legally follows c_i, the 2DFA can keep its two heads on c_i and c_{i+1}, and compare them symbol by symbol. Notice that $L(M') = \emptyset$ iff $L(M) = \emptyset$.
2. Run D on $\langle M'\rangle$. If D accepts, *accept*. If D rejects, *reject*."

Since E_{TM} is undecidable, E_{2DFA} is decidable.

5.21 We formulate the problem of testing 2DIM-DFA equivalence as a language: $EQ_{\text{2DIM-DFA}} = \{\langle M_1, M_2\rangle |\ M_1 \text{ is equivalent to } M_2\}$.
We show that if $EQ_{\text{2DIM-DFA}}$ is decidable, we can decide A_{TM} by reducing A_{TM} to $EQ_{\text{2DIM-DFA}}$. Therefore $EQ_{\text{2DIM-DFA}}$ is undecidable. Assume M is the TM deciding $EQ_{\text{2DIM-DFA}}$. We construct S that decides A_{TM} as follows:

S = "On input $\langle A, w\rangle$,

1. Construct a 2DIM-DFA T that accepts only the accepting computation history of A on w. The input that T accepts should represent the computation history of A in the form of an $m \times n$ rectangle, where $(m-2)$ is the length of the longest configuration that A ever has during the computation and $(n-2)$ is the number of steps it takes to get to the accepting state. Each row of the inner cells contains a proper configuration of A during its computation. In order to check if the input is a computation history of A on w, T follows a procedure similar to that given the proof of Theorem 5.9. It first checks that the first inner row contains the configuration with input w and the start state at the left end. Then it checks whether the TM reaches the accept state at the last inner row. After that, it compares two consecutive rows of input to see if the two configurations follow the transition function of A. If the input passes all the three tests, the 2DIM-DFA accepts. Otherwise, it rejects.
2. Construct another 2DIM-DFA E which always rejects. Run M on $\langle T, E\rangle$ to see if T and E are equivalent. If M accepts, *reject* because there is no computation history for A on w. Otherwise, *accept*."

The TM S decides A_{TM}. However, we know that A_{TM} is undecidable. Therefore $EQ_{\text{2DIM-DFA}}$ is also undecidable.

5.22 Suppose P is a decidable language satisfying the properties and let T_P be a TM that decides P. Without loss of generality assume $\langle T_\emptyset\rangle \notin P$ with $L(T_\emptyset) = \emptyset$ since we can consider $\overline{P}$ if $\langle T_\emptyset\rangle$ is in P. Since P is not trivial, there exists a TM M_1 with $\langle M_1\rangle \in P$. We can decide A_{TM} by constructing the following TM:

"On input $\langle M, w\rangle$:

1. Construct a TM M_w that accepts on input x iff M accepts w and M_1 accepts x.
2. Run T_P on $\langle M_w\rangle$. If it accepts, *reject*. Otherwise, *accept*."

If $w \in L(M)$, $L(M_w) = L(M_1)$ and M_w should be accepted by T_P since $\langle M_1\rangle \in P$. If $w \notin L(M)$, $L(M_w) = \emptyset = L(T_\emptyset)$ and should be rejected by the assumption that $\langle T_\emptyset\rangle \notin P$. Therefore we have $w \in L(M)$ iff $\langle M_1\rangle \in P$. However, since A_{TM} is undecidable, P is not decidable either.

5.23 First, let P be the language $\{\langle M\rangle |\ M$ is a TM with 5 states$\}$. P is non-trivial, and so it satisfies the second condition of Rice's Theorem but P can be easily decided by checking the number of states of the input Turing Machine. Second, let P be the empty set. Then it does not contain any TM and so it satisfies the first condition of Rice's Theorem, but and P can be decided by a TM that always rejects.

Therefore both properties are necessary for proving P undecidable.

Chapter 6

6.1 The following program is in LISP:

```
((LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X)))
(QUOTE (LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X)))))
```

6.8 We build up two incomparable sets A and B to kill off all possible Turing-reductions, $R_1, R_2, \ldots$. The sets start off undefined.

First, extend A to be longer and longer initial portions of the empty set and B to be longer and longer initial portions of ATM. Eventually, R_1 must get the wrong answer when trying to reduce B to A. Then extend A and B interchanging the empty set and ATM until R_1 errs in reducing A to B. Continue with R_2, etc.

Chapter 7

7.1
a. True
b. False
c. False
d. True
e. True
f. True

7.2
a. False
b. True
c. True
d. True
e. False
f. False

7.3 We can use the Euclidean algorithm to find the greatest common divisor.

a.

$$\begin{aligned} 10505 &= 1274 \times 8 + 313 \\ 1274 &= 313 \times 4 + 22 \\ 313 &= 22 \times 14 + 5 \\ 22 &= 5 \times 4 + 2 \\ 5 &= 2 \times 2 + 1 \\ 2 &= 1 \times 2 + 0 \end{aligned}$$

The greatest common divisor of 10505 and 1274 is 1. Therefore they are relatively prime.

b.

$$\begin{aligned} 8029 &= 7289 \times 1 + 740 \\ 7289 &= 740 \times 9 + 629 \\ 740 &= 629 \times 1 + 111 \\ 629 &= 111 \times 5 + 74 \\ 111 &= 74 \times 1 + 37 \\ 74 &= 37 \times 2 + 0 \end{aligned}$$

The greatest common divisor of 8029 and 7289 is 37. Therefore they are not relatively prime.

7.4 The table constructed using the algorithm from Theorem 7.14 is as follows:

	1	2	3	4
1	T	T, R	S	S, R, T
2		R	S	S
3			T	T, R
4				R

Because $table(1, 4)$ contains S, the TM accepts w.

7.5 The formula is not satisfiable. For any assignment of the boolean values for x and y, it always makes one of the four clauses false. Therefore, the formula, which is a conjunction of the four clauses, is always false for any assignment of x and y.

7.6 P is closed under union. For any two P-languages L_1 and L_2, let M_1 and M_2 be the TMs that decide them in polynomial time. We construct a TM M' that decides the union of L_1 and L_2 in polynomial time:

M' = "On input $\langle w \rangle$:

1. Run M_1 on w. If it accepts, *accept*.
2. Run M_2 on w. If it accepts, *accept*. Otherwise, *reject*."

M' accepts w if and only if either M_1 and M_2 accept w. Therefore, M' decides the union of L_1 and L_2. Since both stages take polynomial time, the algorithm runs in polynomial time.

P is closed under concatenation. For any two P-languages L_1 and L_2, let M_1 and M_2 be the TMsthat decide them in polynomial time. We construct a TM M' that decides the concatenation of L_1 and L_2 in polynomial time:

M' = "On input $\langle w \rangle$:

1. For each way to cut w into two substrings $w = w_1 w_2$:
2. Run M_1 on w_1 and M_2 on w_2. If both accept, *accept*.
3. If w is not accepted after trying all the possible cuts, *reject*."

M' accepts w if and only if w can be written as $w_1 w_2$ such that M_1 accepts w_1 and M_2 accepts w_2. Therefore, M' decides the concatenation of L_1 and L_2. Since stage 2 runs in polynomial time and is repeated at most $O(n)$ times, the algorithm runs in polynomial time.

P is closed under complement. For any P-language L, let M be the TM that decides it in polynomial time. We construct a TM M' that decides the complement of L in polynomial time:

M' = "On input $\langle w \rangle$:

1. Run M on w.
2. If M accepts, *reject*. If it rejects, *accept*."

M' decides the complement of L. Since M runs in polynomial time, M' also runs in polynomial time.

7.7 NP is closed under union. For any two NP-languages L_1 and L_2, let M_1 and M_2 be the NTMsthat decide them in polynomial time. We construct a NTM M' that decides the union of L_1 and L_2 in polynomial time:

M' = "On input $\langle w \rangle$:

1. Run M_1 on w. If it accepts, *accept*.
2. Run M_2 on w. If it accepts, *accept*. Otherwise, *reject*."

In both stages 1 and 2, M' uses its nondeterminism when the machines being run make nondeterministic steps. M' accepts w if and only if either M_1 and M_2 accept w. Therefore, M' decides the union of L_1 and L_2. Since both stages take polynomial time, the algorithm runs in polynomial time.

NP is closed under concatenation. For any two NP-languages L_1 and L_2, let M_1 and M_2 be the NTMsthat decide them in polynomial time. We construct a NTM M' that decides the concatenation of L_1 and L_2 in polynomial time:

M' = "On input $\langle w \rangle$:

1. For each way to cut w into two substrings $w = w_1 w_2$:
2. Run M_1 on w_1.
3. Run M_2 on w_2. If both accept, *accept*; otherwise continue with the next choice of w_1 and w_2.
4. If w is not accepted after trying all the possible cuts, *reject*."

In both stages 2 and 3, M' uses its nondeterminism when the machines being run make nondeterministic steps. M' accepts w if and only if w can be expressed as w_1w_2 such that M_1 accepts w_1 and M_2 accepts w_2. Therefore, M' decides the concatenation of L_1 and L_2. Since stage 2 runs in polynomial time and is repeated for at most $O(n)$ time, the algorithm runs in polynomial time.

7.8 If we use unary encoding, the input number n has length n. An algorithm can determine the primality of n by checking sequentially whether n is divisible by any integer from 2 to n. If any of the numbers divide n, *reject*. Otherwise *accept*. The time to do the division is polynomial in n, and the total number of divisions done is at most n. Hence the algorithm runs in polynomial (in n) time. Therefore, *UNARY-PRIMES* is in P.

7.9 The algorithm given in page 145 runs in $O(n^3)$ time. Stage 1 takes at most $O(n)$ steps to locate and mark the start node. Stage 2 causes at most $n+1$ repetitions, because each repetition except the last marks at least one additional node. Each execution of stage 3 uses at most $O(n^3)$ steps because G contains at most n to be checked and for each checked node, examining all adjacent nodes to see whether any have been marked uses at most $O(n^2)$ steps. Therefore in total, stages 2 and 3 take $O(n^4)$ time. Stage 4 uses $O(n)$ steps to scan all nodes. Therefore, the algorithm runs in $O(n^4)$ time and *CONNECTED* is in P.

7.10 We construct a TM M that decides *TRIANGLE* in polynomial time.

M = "On input $\langle G \rangle$ where G is a graph:

1. For each triple of vertices v_1, v_2, v_3 in G:
2. If edges (v_1, v_2), (v_1, v_3), and (v_2, v_3), are all edges of G, *accept*.
3. No triangle has been found in G, so *reject*."

A graph with m vertices has $\binom{m}{3} = \frac{m!}{3!(m-3)!} = O(m^3)$ triples of vertices. Therefore, stage 2 will be repeated at most $O(m^3)$ times. In addition, each stage can be implemented to run in polynomial time. Therefore, *TRIANGLE* $\in$ P.

7.11 A nondeterministic polynomial time algorithm for *ISO* operates as follows:

"On input $\langle G, H \rangle$ where G and H are undirected graphs:

1. Let m be the number of nodes of G and H. If they don't have the same number of nodes, *reject*.
2. Nondeterministically select a permutation π of m elements.
3. For each pair of nodes x and y of G check that (x, y) is an edge of G iff $(\pi(x), \pi(y))$ is an edge of H. If all agree, *accept*. If any differ, *reject*."

Stage 2 can be implemented in polynomial time nondeterministically. Stage 3 takes polynomial time. Therefore $ISO \in$ NP.

7.12 The most obvious algorithm multiplies a by itself b times then compares the result to c, modulo p. That uses $b-1$ multiplications which is exponential in the length of b. Hence this algorithm doesn't run in polynomial time. Here is one that does:

"On input $\langle a, b, c, p \rangle$, four binary integers:

1. Let $r \leftarrow 1$.
2. Let $b_1, \ldots, b_k$ be the bits in the binary representation of b.
3. For $i = 1, \ldots, k$:
4. If $b_i = 0$, let $r \leftarrow r^2 \mod p$.
5. If $b_i = 1$, let $r \leftarrow ar^2 \mod p$.
6. If $(c \mod p) = (r \mod p)$, *accept*; otherwise *reject*."

The algorithm is called *repeated squaring*. Each of its multiplications can be done in polynomial time in the standard way, because the numbers are no larger than p. The total number of multiplications is proportional to the number of bits in b. Hence the total running time is polynomial.

7.13 We show that P is closed under the star operation by dynamic programming. Let $A \in$ P, and M be the TM deciding A in polynomial time. On input $w = w_1 \cdots w_n$, use the following procedure ST to construct a table T such that $T[i,j] = 1$ if $w_i \cdots w_j \in A^*$ and $T[i,j] = 0$ otherwise, for all $1 \le i \le j \le n$.

ST = "On input $w = w_1 \cdots w_n$:

1. If $w = \varepsilon$, *accept*. [[handle $w = \varepsilon$ case]]
2. Initialize $T[i,j] = 0$ for $1 \le i \le j \le n$.
3. For $i = 1$ to n, [[test each substring of length 1]]
4. Set $T[i,i] = 1$ if $w_i \in A$.
5. For $l = 2$ to n, [[l is the length of the substring]]
6. For $i = 1$ to $n - l + 1$, [[i is the substring start position]]
7. Let $j = i + l - 1$, [[j is the substring end position]]
8. If $w_i \cdots w_j \in A$, set $T[i,j] = 1$.
9. For $k = i$ to $j - 1$, [[k is the split position]]
10. If $T[i,k] = 1$ and $T[k,j] = 1$, set $T[i,j] = 1$.
11. *Accept* if $T[1,n] = 1$; otherwise *reject*."

Each stage of the algorithm takes polynomial time, and ST runs for $O(n^3)$ stages, so the algorithm runs in polynomial time.

7.14 Let NTM M decide A in nondeterministic polynomial time. The following NTM N decides A^*.

N = "On input w:

1. If $w = \varepsilon$, *accept*.
2. Nondeterministically choose k, where $1 \leq k \leq n$.
3. Nondeterministically select strings $w_1, w_2, \ldots, w_k$, each of length at most n.
4. If $w \neq w_1 \cdots w_k$, then *reject*(on this branch).
5. Simulate (using nondeterminism) M on each w_i.
6. If M accepts each w_i, then *accept*; otherwise *reject*."

Each stage takes polynomial time, so the algorithm runs in polynomial time.

7.15 The reduction fails because it would result in an exponential unary instance, and therefore would not be a polynomial time reduction.

We give a polynomial time algorithm for this problem using dynamic programming. The input to the algorithm is $\langle S, t \rangle$ where S is a set of numbers $\{x_1, \ldots, x_k\}$. The algorithm operates by constructing a list L_i of numbers for $0 \leq i \leq k$ giving the set of possible values that a subset of the numbers $\{x_1, \ldots, x_i\}$ could sum to. Initially $L_0 = \{0\}$. For each i the algorithm constructs L_i from L_{i-1} by assigning $L_i = L_{i-1} \cup \{a + x_i |\ a \in L_{i-1}\}$. Finally, the algorithm accepts if $t \in L_k$, and rejects otherwise.

7.16 **a.** Follow the marking algorithm for recognizing *PATH* while additionally keeping track of the length of the shortest paths discovered. Here is a more detailed description:

"On input $\langle G, a, b, k \rangle$ where G is a directed graph on m nodes where G has nodes a and b:

1. Place a mark with label 0 on node a.
2. For each i from 0 to m:
3. Scan all the edges of G. If an edge (s, t) is found going from a node s marked with i to an unmarked node b, mark node t with $i + 1$.
4. If t is marked with a value of at most k, *accept*. Otherwise, *reject*."

b. First, $LPATH \in$ NP because a nondeterministic machine can guess and verify a simple path of length at least k from a to b. Next, $UHAMPATH \leq_P LPATH$, because the following TM F computes the reduction f.

F = "On input $\langle G, a, b \rangle$, where G is an undirected graph, and a and b are nodes of G.

1. Let k be the number of nodes of G.
2. Output $\langle G, a, b, k \rangle$."

If $\langle G, a, b \rangle \in UHAMPATH$, G contains a Hamiltonian path of length k from a to b, thus $\langle G, a, b, k \rangle \in LPATH$. If $\langle G, a, b, k \rangle \in LPATH$, G contains a simple path of length k from a to b. Since G has k nodes, the path is Hamiltonian. Thus $\langle G, a, b \rangle \in UHAMPATH$.

7.17 Let A be any language in P except $A = \emptyset$ and $A = \Sigma^*$. To show that A is NP-complete, we show that $A \in$ NP and that every $B \in$ NP is polynomial time reducible to A. The first condition holds because $A \in$ P so $A \in$ NP. To demonstrate that the second condition is true, let $x_{\text{in}} \in A$ and $x_{\text{out}} \notin A$ be two strings that are guaranteed to exist by virtue of our assumptions about A. The assumption that P = NP implies that B is recognized by a polynomial time TM M. A polynomial time reduction from B to A simulates M to determine whether its input w is a member of B, then outputs x_{in} or x_{out} accordingly.

7.19 On input ϕ, a nondeterministic polynomial time machine can guess two assignments and accept if both assignments satisfy ϕ. Thus *DOUBLE-SAT* is in NP. We show $SAT \leq_{\text{P}}$ *DOUBLE-SAT*. The following TM F computes the polynomial time reduction f.

F = "On input $\langle \phi \rangle$, a Boolean formula with variables $x_1, x_2, \ldots, x_m$.

1. Let ϕ' be $\phi \wedge (x \vee \overline{x})$, where x is a new variable.
2. Output $\langle \phi' \rangle$"

If $\phi \in SAT$, ϕ' has at least two satisfying assignments because we can obtain two assignments from the original assignment of ϕ by changing the value of x. If $\phi' \in$ *DOUBLE-SAT*, ϕ is also satisfiable, because x does not appear in ϕ. Therefore $\phi \in SAT$ if and only if $f(\phi) \in$ *DOUBLE-SAT*.

7.20 First, note that we can compose two permutations in polynomial time. Composing q with itself t times requires t compositions if done directly, thereby giving a running time that is exponential in the length of t when t is represented in binary. We can fix this problem using a common technique for fast exponentiation. Calculate $q^1, q^2, q^4, q^8, \ldots, q^k$, where k is the largest power of 2 that is smaller than t. Each term is the square of the previous term. To find q^t we compose the previously computed permutations corresponding to the 1s in ts binary representation. The total number of compositions is now at most the length of t, so the algorithm runs in polynomial time.

7.21 *HALF-CLIQUE* $\in$ NP because a nondeterministic machine can guess and verify a clique with at least $m/2$ nodes in polynomial time. We show that *CLIQUE* $\leq_{\text{P}}$ *HALF-CLIQUE*. The following TM F computes the reduction f.

F = "On input $\langle G, k \rangle$ where G is an undirected graph with m nodes and k is an integer.

1. If $k = m/2$ output $\langle G \rangle$.
2. If $k < m/2$, construct G' by adding a complete graph with $m - 2k$ nodes and connecting them to all nodes in G. Output $\langle G' \rangle$.

3. If $k > m/2$, construct G'' by adding $2k - m$ isolated nodes. Output $\langle G'' \rangle$."

When $k < m/2$, if G has a k-clique, G' has a clique of size $k + (m - 2k) = (2m - 2k)/2$, thus $G' \in$ *HALF-CLIQUE*. If G' has a $k + (m - 2k)$-clique, then at most $m - 2k$ of them come from the new nodes, thus G must contain a k-clique. When $k > m/2$, if G has a k-clique, G'' has a clique of size $(m + 2k - m)/2 = k$, thus $G'' \in$ *HALF-CLIQUE*. If $G'' \in$ *HALF-CLIQUE*, G must contain a k-clique because the half-clique cannot include any of the new isolated nodes.

7.22 **a.** In an $\neq$-assignment each clause has at least one literal assigned 1 and at least one literal assigned 0. The negation of an $\neq$-assignment preserves this property, so it too is an $\neq$-assignment.

b. To prove that the given reduction works, we need to show that if the formula ϕ is mapped to ϕ', then ϕ is satisfiable (in the ordinary sense) iff ϕ' has an $\neq$-assignment. First, if ϕ is satisfiable, we can obtain an $\neq$-assignment for ϕ' by extending the assignment to ϕ so that we assign z_i to 1 if both literals y_1 and y_2 in clause c_i of ϕ are assigned 0. Otherwise we assign z_i to 0. Finally, we assign b to 0. Second, if ϕ' has an $\neq$-assignment we can find a satisfying assignment to ϕ as follows. By part (a) we may assume the $\neq$-assignment assigns b to 0 (otherwise, negate the assignment). This assignment cannot assign all of y_1, y_2, and y_3 to 0, because doing so would force one of the clauses in ϕ' to have all 0s. Hence, restricting this assignment to the variables of ϕ gives a satisfying assignment.

c. Clearly, $\neq$*SAT* is in NP. Hence, it is NP-complete because *3SAT* reduces to it.

7.23 First, *MAX-CUT* $\in$ NP because a nondeterministic algorithm can guess the cut and check that it has size at least k in polynomial time.

Second, we show $\neq$ *3SAT* $\leq_P$ *MAX-CUT*. Use the reduction suggested in the hint. For clarity, though, use c instead of k for the number of clauses. Hence, given an instance ϕ of $\neq$ *3SAT* with v variables and c clauses. We build a graph G with $6cv$ nodes. Each variable x_i corresponds to $6c$ nodes; $3c$ labeled x_i and $3c$ labeled $\overline{x_i}$. Connect each x_i node with each $\overline{x_i}$ node for a total of $9c^2$ edges. For each clause in ϕ select three nodes labeled by the literal in that clause and connect them by edges. Avoid selecting the same node more than once (each label has $3c$ copies, so we cannot run out of nodes). Let k be $9c^2v + 2c$. Output $\langle G, k \rangle$. Next we show this reduction works.

We show that ϕ has a $\neq$-assignment iff G has a cut of size at least k. Take an $\neq$-assignment for ϕ. It yields a cut of size k by placing all nodes corresponding to true variables on one side and all other nodes on the other side. Then the cut contains all $9c^2v$ edges between literals and their negations and two edges

for each clause, because it contains at least one true and one false literal, yielding a total of $9c^2v + 2c = k$ edges.

For the converse, take a cut of size k. Observe that a cut can contain at most two edges for each clause, because at least two of the corresponding literals must be on the same side. Thus, the clause edges can contribute at most $2c$ to the cut. The graph G has only $9c^2v$ other edges, so if G has a cut of size k, all of those other edges appear in it and each clause contributes its maximum. Hence, all nodes labeled by the same literal occur on the same side of the cut. By selecting either side of the cut and assigning its literals true, we obtain an assignment. Because each clause contributes two edges to the cut, it must have nodes appearing on both sides and so it has at least one true and one false literal. Hence ϕ has a $\neq$-assignment.

(A minor glitch arises in the above reduction if ϕ has a clause that contains both x and $\overline{x}$ for some variable x. In that case G would need to be a multigraph, because it would contain two edges joining certain nodes. We can avoid this situation by observing that such clauses can be removed from ϕ without changing its $\neq$-satisfiability.)

7.24 To show that all problems in NP are polynomial time reducible to D we show that *3SAT* $\leq_P D$. Given a 3cnf formula ϕ we construct a polynomial p with the same variables. We represent the variable x in ϕ by x in p and its negation $\overline{x}$ by $1 - x$. Next we represent the Boolean operations $\wedge$ and $\vee$ in ϕ by arithmetic operations that simulate them, as follows. The Boolean expression $y_1 \wedge y_2$ becomes $y_1 y_2$ and $y_1 \vee y_2$ becomes $(1 - (1 - y_1)(1 - y_2))$. Hence, the clause $(y_1 \vee y_2 \vee y_3)$ becomes $(1 - (1 - y_1)(1 - y_2)(1 - y_3))$. The conjunction of the clauses of ϕ becomes a product of their individual representations. The result of this transformation is a polynomial, q, which simulates ϕ in the sense they both have the same value when the variables have Boolean assignments. Thus the polynomial $1 - q$ has an integral (in fact Boolean) root if ϕ is satisfiable. However, $1 - q$ does not meet the requirements of p, because $1 - q$ may have an integral (but non-Boolean) root even when ϕ is unsatisfiable. To prevent that situation, we observe that the polynomial $r = (x_1(1 - x_1))^2(x_2(1 - x_2))^2 \cdots (x_m(1 - x_m))^2$ is 0 only when the variables $x1, \ldots, x_l$ take on Boolean values, and is positive otherwise. Hence polynomial $p = (1 - q)^2 + r$ has an integral root iff ϕ is satisfiable.

7.25 U is in NP because on input $\langle d, x, 1^t \rangle$ a nondeterministic algorithm can simulate M_d on x (making nondeterministic branches when M_d does) for t steps and accept if M_d accepts. Doing so takes time polynomial in the length of the input because t appears in unary.

To show *3SAT* $\leq_P U$, let M be a NTM that decides *3SAT* in time cn^k for some constants c and k. Given a formula ϕ, transform it into $w = \langle \langle M \rangle, \phi, 1^{c|\phi|^k} \rangle$. By the definition of M, $w \in U$ if and only if $\phi \in$ *3SAT*.

7.26 *PUZZLE* is in NP because we can guess a solution and verify it in polynomial time. We reduce *3SAT* to *PUZZLE* in polynomial time. Given a Boolean formula ϕ, convert it to a set of cards as follows.

Let $x_1, x_2, \ldots, x_m$ be the variables of ϕ and let $c_1, c_2, \ldots, c_l$ be the clauses. Say that $z \in c_j$ if z is one of the literals in c_j. Let each card have column 1 and column 2 with l possible holes in each column numbered 1 through l. When plàcing a card in the box, say that a card is *face up* if column 1 is on the left and column 2 is on the right; otherwise *face down*. The following algorithm F computes the reduction.

F = "On input ϕ:

1. Create one card for each x_i as follows: in column 1 punch out all holes except any hole j such that $x_i \in c_j$; in column 2 punch out all holes except any hole j' such that $\overline{x_i} \in c_{j'}$.
2. Create an *extra* card with all holes in column 1 punched out and no hole in column 2 punched out.
3. Output the description of the cards created."

Given a satisfying assignment for ϕ, a solution to the *PUZZLE* can be constructed as follows. For each x_i assigned True (False), put its card face up (down), and put the extra card face up. Then, every hole in column 1 is covered because the associated clause has a literal assigned True, and every hole in column 2 is covered by the extra card.

Given a solution to the *PUZZLE*, a satisfying assignment for ϕ can be constructed as follows. Flip the deck so that the extra card covers column 2. Assign the variables according to the correponding cards, True for up and False for down. Because every hole in column 1 is covered, every clause must contain at least one literal assigned True in this assignment. Hence it satisfies ϕ.

7.27 *SET-SPLITTING* in in NP because a coloring of the elements can be guesses and verified to have the desired properties in polynomial time. We give a polynomial time reduction f from *3SAT* to *SET-SPLITTING* as follows.

Given a 3cnf ϕ, we set $S = \{x_1, \overline{x_1}, \ldots, x_m, \overline{x_m}, y\}$. In other words, S contains an element for each literal (two for each variable) and a special element y. For every clause c_i in ϕ, let C_i be a subset of S containing the elements corresponding to the literals in c_i and the special element $y \in S$. We also add subsets C_{x_i} for each literal, containing elements x_i and $\overline{x_i}$. Then $C = \{C_1, \ldots, C_l, C_{x_1}, \ldots, C_{x_l}\}$.

If ϕ is satisfiable, consider a satisfying assignment. If we color all the true literals red, all the false ones blue, and y blue, then every subset C_i of S has at least one red element (because c_i of ϕ is "turned on" by some literal) and it also contains one blue element (y). In addition, every subset C_{x_i} has exactly one element red and one blue, so that the system $\langle S, C\rangle \in$ *SET-SPLITTING*.

If $\langle S, C\rangle \in$ *SET-SPLITTING*, then look at the coloring for S. If we set the literals to true which are colored differently from y, and those to false which are the same color as y we obtain a satisfying assignment to ϕ.

7.28 Let $F = \{\langle a, b, c\rangle |\ a, b, c$ are binary integers and $a = pq$ for $b \leq p \leq c\}$. We see that $F \in$ NP because a nondeterministic algorithm can guess p and check that p divides a and that $b \leq p \leq c$ in polynomial time. If P = NP then $F \in$ P. In that case, we can locate a factor of a by successively running F on smaller and smaller intervals b, c, in effect performing a binary search on the interval 1 to a. The running time of that procedure is the log of the size of the interval, and thus is polynomial in the length of a.

7.29 Assuming P = NP, we have a polynomial time algorithm deciding *SAT*. To produce a satisfying assignment for a satisfiable formula ϕ, substitute $x_1 = 0$ and $x_1 = 1$ in ϕ and test the satisfiability of the two resulting formulas ϕ_0 and ϕ_1. At least one of these must be satisfiable because ϕ is satisfiable. At least one of these must be satisfiable because ϕ is satisfiable. If ϕ_0 is satisfiable, pick $x_1 = 0$; otherwise ϕ_1 must be satisfiable and pick $x_1 = 1$. That gives the value of x_1 in a satisfying assignment. Make that substitution permanent and determine a value for x_2 in that satisfying assignment in the same way. Continue until all variables been substituted.

7.30 If P = NP, then *CLIQUE* is recognizable in polynomial time. We show how to compute *MAX-CLIQUE* using *CLIQUE*. Let m be the number of nodes in the input graph. For each i from 1 to m, test whether there exists a clique of size i using the polynomial time algorithm for *CLIQUE*. Output the largest such i for which a clique exists.

To find the maximum clique, we start with i, the maximum clique size. Remove one node and see if there is still a clique of size i. If not, restore that node and remove another. If so, iterate the process until we are left with a graph of i nodes, which must be a clique. This process takes at most m trials to find which node to remove, and at most m nodes need to be removed. The total running time is therefore polynomial.

7.31 Test whether a path exists from the start state to each non-accepting state. [this problem is too easy—should have been only an exercise]

7.32 If we used 2×2 windows, we might not detect certain invalid computation histories where multiple heads occurred in the same row. For example, if the NTM had the nondeterministic choice of moving its head left or right when in a certain state reading some symbol, the next configuration must have one of these possibilities, but not both simultaneously. However, using a 2×2 window, a row that contained two state symbols corresponding to each possible next move would not appear incorrect, because each window would itself be legal.

Furthermore, once two heads appeared in a single row, they could collaborate to make the final row accepting, even though no actual accepting computation existed.

7.33 The given algorithm does use exponential time, but a different, polynomial time algorithm could conceivably exist. Hence, *SAT* might have polynomial time complexity. The error in the proof occurs in the sentence, "Thus *SAT* has exponential time complexity" because that conclusion doesn't follow logically from the previous statements.

7.34 *3COLOR* is in NP because a coloring can be verified in polynomial time. We show *3SAT* $\leq_P$ *3COLOR*. Let $\phi = c_1 \wedge c_2 \wedge \cdots \wedge c_l$ be a 3cnf formula over variables $x_1, x_2, \cdots, x_m$, where the c_i's are the clauses. We build a graph G_ϕ containing $2m + 6l + 3$ nodes: 2 nodes for each variable; 6 nodes for each clause; and 3 extra nodes. We describe G_ϕ in terms of the subgraph gadgets given in the Hint.

G_ϕ contains a variable gadget for each variable x_i, two OR-gadgets for each clause, and one palette gadget. The four bottom nodes of the OR-gadgets will be merged with other nodes in the graph, as follows. Label the nodes of the palette gadget T, F, and R. Label the nodes in each variable gadget + and − and connect each to the R node in the palette gadget. In each clause, connect the top of one of the OR-gadgets to the F node in the palette. Merge the bottom node of that OR-gadget with the top node of the other OR-gadget. Merge the three remaining bottom nodes of the two OR-gadgets with corresponding nodes in the variable gadgets so that if a clause contains the literal x_i, one of its bottom nodes is merged with the + node of x_i whereas if the clause contains the literal $\overline{x_i}$, one of its bottom nodes is merged with the − node of x_i.

To show the construction is correct, we first demonstrate that if ϕ is satisfiable, the graph is 3-colorable. The three colors are called T, F, and R. Color the palette with its labels. For each variable, color the + node T and the − node F if the variable is True in a satisfying assignment; otherwise reverse the colors. Because each clause has one True literal in the assignment, we can color the nodes of the OR-gadgets of that clause so that the node connected to the F node in the palette is not colored F. Hence we have a proper 3-coloring.

If we start out with a 3-coloring, we can obtain a satisfying assignment by taking the colors assigned to the + nodes of each variable. Observe that neither node of the variable gadget can be colored R, because all variable nodes are connected to the R node in the palette. Furthermore, if both bottom nodes of an OR-gadget are colored F, the top node must be colored F, and hence, each clause contain a true literal. Otherwise, the three bottom nodes that were merged with variable nodes would be colored F, and then both top nodes would be colored F, but one of the top nodes is connected to the F node in the palette.

7.36 We build an NFA in polynomial time that accepts all non-satisfying assignments. The NFA operates by guessing a clause is not satisfied by the assignment and then reading the input to check that the clause is indeed not

satisfied. More precisely, the NFA contains $l(m+1)+1$ states, where l is the number of clauses and m is the number of variables in ϕ. Each clause c_j is represented by $m+1$ states $q_{j,1}, \ldots, q_{j,m+1}$. The start state q_0 branches nondeterministically to each state $q_{1,j}$ on ϵ. Each state $q_{i,j}$ branches to $q_{i,j+1}$ for $j \leq m$. The label on that transition is 0 if x_i appears in clause c_j, and is 1 if $\overline{x_i}$ appears in c_j. If neither literals appear in c_j, the label on the transition is 0,1. Each state $q_{i,m+1}$ is an accept state.

If we could minimize this NFA in polynomial time, we would be able to determine whether it accepts all strings of length m, and hence whether ϕ is satisfiable, in polynomial time. That would imply P = NP.

7.37 The clause $(x \vee y)$ is logically equivalent to each of the expressions $(\overline{x} \rightarrow y)$ and $(\overline{y} \rightarrow x)$. We represent the 2cnf formula ϕ on the variables $x_1, \ldots, x_m$ by a directed graph G on $2m$ nodes labeled with the literals over these variables. For each clause in ϕ, place two edges in the graph corresponding to the the two implications above. Then we show that ϕ is satisfiable iff G doesn't contain a cycle containing both x_i and $\overline{x_i}$ for any i. We'll call such a cycle an ***inconsistency cycle***. Testing whether G contains an inconsistency cycle is easily done in polynomial time with a marking algorithm, or a depth-first search algorithm.

We first show that if G contains an inconsistency cycle, no satisfying assignment can exist. The sequences of implications in the inconsistency cycle yields the logical equivalence $x_i \leftrightarrow \overline{x_i}$ for some i, and that is contradictory. Thus ϕ is unsatisfiable.

Next, we show that if G doesn't contain an inconsistency cycle, ϕ is satisfiable. Write $x \stackrel{*}{\rightarrow} y$ if G contains a path from node x to node y. Because G contains the two designated edges for each clause in ϕ, we have $x \stackrel{*}{\rightarrow} y$ iff $\overline{y} \stackrel{*}{\rightarrow} \overline{x}$. Now we construct the satisfying assignment.

Pick any variable x_i. We cannot have both $x_i \stackrel{*}{\rightarrow} \overline{x_i}$ and $\overline{x_i} \stackrel{*}{\rightarrow} x_i$ because G doesn't contain an inconsistency cycle. Select literal x_i if $\overline{x_i} \stackrel{*}{\rightarrow} x_i$ is false, and otherwise select $\overline{x_i}$. Assign the selected literal and all implied literals (those reachable along paths from the selected node) to be True. Note that we never assign both x_j and $\overline{x_j}$ True because if $x_i \stackrel{*}{\rightarrow} x_j$ and $x_i \stackrel{*}{\rightarrow} \overline{x_j}$ then $x_j \stackrel{*}{\rightarrow} \overline{x_i}$ and hence $x_i \stackrel{*}{\rightarrow} \overline{x_i}$ thus we would not have selected literal x_i (similarly for $\overline{x_i}$). Then, we remove all nodes labeled with assigned literals or their complements from G, and repeat this paragraph until all variables are assigned.

The resulting assignment satisfies ϕ because it satisfies every clause. As soon as one of the literals in a clause is assigned False, the other must be assigned True because G contains an arrow from the negation of the falsified literal to that other literal.