Introduction to Automata Theory, Languages, and Computation

Solutions for Chapter 2

Revised 9/6/01.

Solutions for Section 2.2

Exercise 2.2.1(a)

States correspond to the eight combinations of switch positions, and also must indicate whether the previous roll came out at *D*, i.e., whether the previous input was accepted. Let 0 represent a position to the left (as in the diagram) and 1 a position to the right. Each state can be represented by a sequence of three 0's or 1's, representing the directions of the three switches, in order from left to right. We follow these three bits by either *a* indicating it is an accepting state or *r*, indicating rejection. Of the 16 possible states, it turns out that only 13 are accessible from the initial state, 000r. Here is the transition table:

	A	**B**
->000r	100r	011r
*000a	100r	011r
*001a	101r	000a
010r	110r	001a
*010a	110r	001a
011r	111r	010a
100r	010r	111r
*100a	010r	111r
101r	011r	100a
*101a	011r	100a
110r	000a	101a
*110a	000a	101a
111r	001a	110a

Exercise 2.2.2

The statement to be proved is $\delta\text{-}hat(q,xy) = \delta\text{-}hat(\delta\text{-}hat(q,x),y)$, and we proceed by induction on the length of y.

Basis: If $y = \varepsilon$, then the statement is $\delta\text{-}hat(q,x) = \delta\text{-}hat(\delta\text{-}hat(q,x),\varepsilon)$. This statement follows from the basis in the definition of $\delta\text{-}hat$. Note that in applying this definition, we must treat $\delta\text{-}hat(q,x)$ as if it were just a state, say p. Then, the statement to be proved is $p = \delta\text{-}hat(p,\varepsilon)$, which is easy to recognize as the basis in the definition of $\delta\text{-}hat$.

Induction: Assume the statement for strings shorter than y, and break $y = za$, where a is the last symbol of y. The steps converting $\delta\text{-}hat(\delta\text{-}hat(q,x),y)$ to $\delta\text{-}hat(q,xy)$ are summarized in the following table:

Expression	Reason
δ-hat(δ-hat(q,x),y)	Start
δ-hat(δ-hat(q,x),za)	*y=za* by assumption
δ(δ-hat(δ-hat(q,x),z),a)	Definition of *δ-hat*, treating *δ-hat(q,x)* as a state
δ(δ-hat(q,xz),a)	Inductive hypothesis
δ-hat(q,xza)	Definition of *δ-hat*
δ-hat(q,xy)	*y=za*

Exercise 2.2.4(a)

The intuitive meanings of states *A*, *B*, and *C* are that the string seen so far ends in 0, 1, or at least 2 zeros.

	0	**1**
->A	B	A
B	C	A
*C	C	A

Exercise 2.2.6(a)

The trick is to realize that reading another bit either multiplies the number seen so far by 2 (if it is a 0), or multiplies by 2 and then adds 1 (if it is a 1). We don't need to remember the entire number seen --- just its remainder when divided by 5. That is, if we have any number of the form *5a+b*, where *b* is the remainder, between 0 and 4, then *2(5a+b) = 10a+2b*. Since 10*a* is surely divisible by 5, the remainder of *10a+2b* is the same as the remainder of 2b when divided by 5. Since *b*, is 0, 1, 2, 3, or 4, we can tabulate the answers easily. The same idea holds if we want to consider what happens to *5a+b* if we multiply by 2 and add 1.

The table below shows this automaton. State *qi* means that the input seen so far has remainder *i* when divided by 5.

	0	**1**
->*q0	q0	q1
q1	q2	q3
q2	q4	q0
q3	q1	q2
q4	q3	q4

There is a small matter, however, that this automaton accepts strings with leading 0's. Since the problem calls for accepting only those strings that begin with 1, we need an additional state *s*, the start state, and an additional ``dead state'' *d*. If, in state *s*, we see a 1 first, we act like *q0*; i.e., we go to state *q1*. However, if the first input is 0, we should never accept, so we go to state *d*, which we never leave. The complete automaton is:

	0	**1**
->s	d	q1
*q0	q0	q1
q1	q2	q3

q2	q4	q0
q3	q1	q2
q4	q3	q4
d	d	d

Exercise 2.2.9
Part (a) is an easy induction on the length of *w*, starting at length 1.

Basis: |*w*| = 1. Then $\delta\text{-}hat(q_0,w) = \delta\text{-}hat(q_f,w)$, because *w* is a single symbol, and δ*-hat* agrees with δ on single symbols.

Induction: Let $w = za$, so the inductive hypothesis applies to *z*. Then $\delta\text{-}hat(q_0,w) = \delta\text{-}hat(q_0,za) = \delta(\delta\text{-}hat(q_0,z),a) = \delta(\delta\text{-}hat(q_f,z),a)$ [by the inductive hypothesis] = $\delta\text{-}hat(q_f,za) = \delta\text{-}hat(q_f,w)$.

For part (b), we know that $\delta\text{-}hat(q_0,x) = q_f$. Since $x\varepsilon$, we know by part (a) that $\delta\text{-}hat(q_f,x) = q_f$. It is then a simple induction on *k* to show that $\delta\text{-}hat(q_0,x^k) = q_f$.

Basis: For $k=1$ the statement is given.

Induction: Assume the statement for $k-1$; i.e., $\delta\text{-}hat(q_0,xSUP>k-1) = q_f$. Using Exercise 2.2.2, $\delta\text{-}hat(q_0,x^k) = \delta\text{-}hat(\delta\text{-}hat(q_0,x^{k-1}),x) = \delta\text{-}hat(q_f,x)$ [by the inductive hypothesis] = q_f [by (a)].

Exercise 2.2.10
The automaton tells whether the number of 1's seen is even (state *A*) or odd (state *B*), accepting in the latter case. It is an easy induction on |*w*| to show that $dh(A,w) = A$ if and only if *w* has an even number of 1's.

Basis: |*w*| = 0. Then *w*, the empty string surely has an even number of 1's, namely zero 1's, and $\delta\text{-}hat(A,w) = A$.

Induction: Assume the statement for strings shorter than *w*. Then $w = za$, where *a* is either 0 or 1.

Case 1: $a = 0$. If *w* has an even number of 1's, so does *z*. By the inductive hypothesis, $\delta\text{-}hat(A,z) = A$. The transitions of the DFA tell us $\delta\text{-}hat(A,w) = A$. If *w* has an odd number of 1's, then so does *z*. By the inductive hypothesis, $\delta\text{-}hat(A,z) = B$, and the transitions of the DFA tell us $\delta\text{-}hat(A,w) = B$. Thus, in this case, $\delta\text{-}hat(A,w) = A$ if and only if *w* has an even number of 1's.

Case 2: $a = 1$. If *w* has an even number of 1's, then *z* has an odd number of 1's. By the inductive hypothesis, $\delta\text{-}hat(A,z) = B$. The transitions of the DFA tell us $\delta\text{-}hat(A,w) = A$. If *w* has an odd number of 1's, then *z* has an even number of 1's. By the inductive hypothesis, $\delta\text{-}hat(A,z) = A$, and the transitions of the DFA tell us $\delta\text{-}hat(A,w) = B$. Thus, in this case as well, $\delta\text{-}hat(A,w) = A$ if and only if *w* has an even number of 1's.

Solutions for Section 2.3

Exercise 2.3.1
Here are the sets of NFA states represented by each of the DFA states A through H: A = {p}; B = {p,q}; C = {p,r}; D = {p,q,r}; E = {p,q,s}; F = {p,q,r,s}; G = {p,r,s}; H = {p,s}.

	0	1
->A	B	A
B	D	C
C	E	A
D	F	C
*E	F	G
*F	F	G
*G	E	H
*H	E	H

Exercise 2.3.4(a)

The idea is to use a state *qi*, for *i* = 0,1,...,9 to represent the idea that we have seen an input *i* and guessed that this is the repeated digit at the end. We also have state *qs*, the initial state, and *qf*, the final state. We stay in state *qs* all the time; it represents no guess having been made. The transition table:

	0	1	...	9
->qs	{qs,q0}	{qs,q1}	...	{qs,q9}
q0	{qf}	{q0}	...	{q0}
q1	{q1}	{qf}	...	{q1}
...	...	...	...	...
q9	{q9}	{q9}	...	{qf}
*qf	{}	{}	...	{}

Solutions for Section 2.4

Exercise 2.4.1(a)

We'll use q0 as the start state. q1, q2, and q3 will recognize *abc*; q4, q5, and q6 will recognize *abd*, and q7 through q10 will recognize *aacd*. The transition table is:

	a	b	c	d
->q0	{q0,q1,q4,q7}	{q0}	{q0}	{q0}
q1	{}	{q2}	{}	{}
q2	{}	{}	{q3}	{}
*q3	{}	{}	{}	{}
q4	{}	{q5}	{}	{}
q5	{}	{}	{}	{q6}
*q6	{}	{}	{}	{}
q7	{q8}	{}	{}	{}

q8	{}	{}	{q9}	{}
q9	{}	{}	{}	{q10}
*q10	{}	{}	{}	{}

Exercise 2.4.2(a)
The subset construction gives us the following states, each representing the subset of the NFA states indicated: *A = {q0}; B = {q0,q1,q4,q7}; C = {q0,q1,q4,q7,q8}; D = {q0,q2,q5}; E = {q0,q9}; F = {q0,q3}; G = {q0,q6}; H = {q0,q10}*. Note that *F, G* and *H* can be combined into one accepting state, or we can use these three state to signal the recognition of *abc, abd*, and *aacd*, respectively.

	a	**b**	**c**	**d**
->A	B	A	A	A
B	C	D	A	A
C	C	D	E	A
D	B	A	F	G
E	B	A	A	H
*F	B	A	A	A
*G	B	A	A	A
*H	B	A	A	A

Solutions for Section 2.5

Exercise 2.5.1
For part (a): the closure of *p* is just *{p}*; for *q* it is *{p,q}*, and for *r* it is *{p,q,r}*.

For (b), begin by noticing that *a* always leaves the state unchanged. Thus, we can think of the effect of strings of *b*'s and *c*'s only. To begin, notice that the only ways to get from *p* to *r* for the first time, using only *b*, *c*, and ε-transitions are *bb*, *bc*, and *c*. After getting to *r*, we can return to *r* reading either *b* or *c*. Thus, every string of length 3 or less, consisting of *b*'s and *c*'s only, is accepted, with the exception of the string *b*. However, we have to allow *a*'s as well. When we try to insert *a*'s in these strings, yet keeping the length to 3 or less, we find that every string of *a*'s *b*'s, and *c*'s with at most one *a* is accepted. Also, the strings consisting of one *c* and up to 2 *a*'s are accepted; other strings are rejected.

There are three DFA states accessible from the initial state, which is the ε closure of *p*, or *{p}*. Let *A = {p}, B = {p,q}*, and *C = {p,q,r}*. Then the transition table is:

	a	**b**	**c**
->A	A	B	C
B	B	C	C
*C	C	C	C

Solutions for Chapter 3

Solutions for Section 3.1

Exercise 3.1.1(a)
The simplest approach is to consider those strings in which the first *a* precedes the first *b* separately from those where the opposite occurs. The expression: **c*a(a+c)*b(a+b+c)* + c*b(b+c)*a(a+b+c)***

Exercise 3.1.2(a)
(Revised 9/5/05) The trick is to start by writing an expression for the set of strings that have no two adjacent 1's. Here is one such expression: **(10+0)*(ε+1)**

To see why this expression works, the first part consists of all strings in which every 1 is followed by a 0. To that, we have only to add the possibility that there is a 1 at the end, which will not be followed by a 0. That is the job of (ε+1).

Now, we can rethink the question as asking for strings that have a prefix with no adjacent 1's followed by a suffix with no adjacent 0's. The former is the expression we developed, and the latter is the same expression, with 0 and 1 interchanged. Thus, a solution to this problem is **(10+0)*(ε+1)(01+1)*(ε+0)**. Note that the ε+1 term in the middle is actually unnecessary, as a 1 matching that factor can be obtained from the **(01+1)*** factor instead.

Exercise 3.1.4(a)
This expression is another way to write ``no adjacent 1's.'' You should compare it with the different-looking expression we developed in the solution to Exercise 3.1.2(a). The argument for why it works is similar. **(00*1)*** says every 1 is preceded by at least one 0. **0*** at the end allows 0's after the final 1, and (ε+**1**) at the beginning allows an initial 1, which must be either the only symbol of the string or followed by a 0.

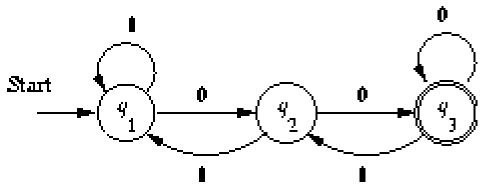
Exercise 3.1.5
The language of the regular expression ε. Note that ε* denotes the language of strings consisting of any number of empty strings, concatenated, but that is just the set containing the empty string.

Solutions for Section 3.2

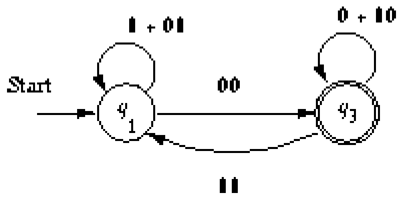
Exercise 3.2.1
Part (a): The following are all R^0 expressions; we list only the subscripts. R11 = ε+**1**; R12 = **0**; R13 = phi; R21 = **1**; R22 = ε; R23 = **0**; R31 = phi; R32 = **1**; R33 = ε+**0**.

Part (b): Here all expression names are $R^{(1)}$; we again list only the subscripts. R11 = **1***; R12 = **1*0**; R13 = phi; R21 = **11***; R22 = ε+**11*0**; R23 = **0**; R31 = phi; R32 = **1**; R33 = ε+**0**.

Part (e): Here is the transition diagram:

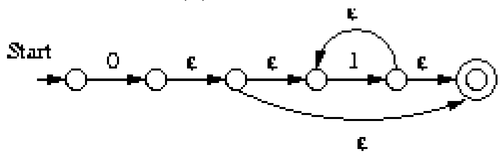


If we eliminate state *q2* we get:



Applying the formula in the text, the expression for the ways to get from *q1* to *q3* is: **[1 + 01 + 00(0+10)*11]*00(0+10)***

Exercise 3.2.4(a)



Exercise 3.2.6(a)

(Revised 1/16/02) LL^* or L^+.

Exercise 3.2.6(b)

The set of suffixes of strings in *L*.

Exercise 3.2.8

Let $R^{(k)}_{ijm}$ be the number of paths from state *i* to state *j* of length *m* that go through no state numbered higher than *k*. We can compute these numbers, for all states *i* and *j*, and for *m* no greater than *n*, by induction on *k*.

Basis: R^{0}_{ij1} is the number of arcs (or more precisely, arc labels) from state *i* to state *j*. $R^{0}_{ii0} = 1$, and all other R^{0}_{ijm}'s are 0.

Induction: $R^{(k)}_{ijm}$ is the sum of $R^{(k-1)}_{ijm}$ and the sum over all lists *(p1,p2,...,pr)* of positive integers that sum to *m*, of $R^{(k-1)}_{ikp1} * R^{(k-1)}_{kkp2} * R^{(k-1)}_{kkp3} * ... * R^{(k-1)}_{kkp(r-1)} * R^{(k-1)}_{kjpr}$. Note *r* must be at least 2.

The answer is the sum of $R^{(k)}_{1jn}$, where *k* is the number of states, 1 is the start state, and *j* is any accepting state.

Solutions for Section 3.4

Exercise 3.4.1(a)

Replace R by $\{a\}$ and S by $\{b\}$. Then the left and right sides become $\{a\}$ *union* $\{b\} = \{b\}$ *union* $\{a\}$. That is, $\{a,b\} = \{b,a\}$. Since order is irrelevant in sets, both languages are the same: the language consisting of the strings a and b.

Exercise 3.4.1(f)

Replace R by $\{a\}$. The right side becomes $\{a\}^*$, that is, all strings of a's, including the empty string. The left side is $(\{a\}^*)^*$, that is, all strings consisting of the concatenation of strings of a's. But that is just the set of strings of a's, and is therefore equal to the right side.

Exercise 3.4.2(a)

Not the same. Replace R by $\{a\}$ and S by $\{b\}$. The left side becomes all strings of a's and b's (mixed), while the right side consists only of strings of a's (alone) and strings of b's (alone). A string like ab is in the language of the left side but not the right.

Exercise 3.4.2(c)

Also not the same. Replace R by $\{a\}$ and S by $\{b\}$. The right side consists of all strings composed of zero or more occurrences of strings of the form $a...ab$, that is, one or more a's ended by one b. However, every string in the language of the left side has to end in ab. Thus, for instance, ε is in the language on the right, but not on the left.

Solutions for Chapter 4

Solutions for Section 4.1

Exercise 4.1.1(c)

Let n be the pumping-lemma constant (note this n is unrelated to the n that is a local variable in the definition of the language L). Pick $w = 0^n10^n$. Then when we write $w = xyz$, we know that $|xy| <= n$, and therefore y consists of only 0's. Thus, xz, which must be in L if L is regular, consists of fewer than n 0's, followed by a 1 and exactly n 0's. That string is not in L, so we contradict the assumption that L is regular.

Exercise 4.1.2(a)

Let n be the pumping-lemma constant and pick $w = 0^{n^2}$, that is, n^2 0's. When we write $w = xyz$, we know that y consists of between 1 and n 0's. Thus, $xyyz$ has length between $n^2 + 1$ and $n^2 + n$. Since the next perfect square after n^2 is $(n+1)^2 = n^2 + 2n + 1$, we know that the length of $xyyz$ lies strictly between the consecutive perfect squares n^2 and $(n+1)^2$. Thus, the length of $xyyz$ cannot be a perfect square. But if the language were regular, then $xyyz$ would be in the language, which contradicts the assumption that the language of strings of 0's whose length is a perfect square is a regular language.

Exercise 4.1.4(a)
We cannot pick w from the empty language.

Exercise 4.1.4(b)
If the adversary picks $n = 3$, then we cannot pick a w of length at least n.

Exercise 4.1.4(c)
The adversary can pick an $n > 0$, so we have to pick a nonempty w. Since w must consist of pairs 00 and 11, the adversary can pick y to be one of those pairs. Then whatever i we pick, xy^iz will consist of pairs 00 and 11, and so belongs in the language.

Solutions for Section 4.2

Exercise 4.2.1(a)
aabbaa.

Exercise 4.2.1(c)
The language of regular expression **a(ab)*ba**.

Exercise 4.2.1(e)
Each b must come from either 1 or 2. However, if the first b comes from 2 and the second comes from 1, then they will both need the a between them as part of $h(2)$ and $h(1)$, respectively. Thus, the inverse homomorphism consists of the strings $\{110, 102, 022\}$.

Exercise 4.2.2
Start with a DFA A for L. Construct a new DFA B, that is exactly the same as A, except that state q is an accepting state of B if and only if $\delta(q,a)$ is an accepting state of A. Then B accepts input string w if and only if A accepts wa; that is, $L(B) = L/a$.

Exercise 4.2.5(b)
We shall use D_a for ``the derivative with respect to a." The key observation is that if *epsilon* is not in $L(R)$, then the derivative of RS will always remove an a from the portion of a string that comes from R. However, if *epsilon* is in $L(R)$, then the string might have nothing from R and will remove a from the beginning of a string in $L(S)$ (which is also a string in $L(RS)$. Thus, the rule we want is:

If *epsilon* is not in $L(R)$, then $D_a(RS) = (D_a(R))S$. Otherwise, $D_a(RS) = D_a(R)S + D_a(S)$.

Exercise 4.2.5(e)
L may have no string that begins with 0.

Exercise 4.2.5(f)
This condition says that whenever $0w$ is in L, then w is in L, and vice-versa. Thus, L must be of the form $L(0^*)M$ for some language M (not necessarily a regular language) that has no string beginning with 0.

In proof, notice first that $D_0(L(0^*)M = D_0(L(0^*))M$ *union* $D_0(M) = L(0^*)M$. There are two reasons for the last step. First, observe that D_0 applied to the language of all strings of 0's gives all strings of 0's, that is, $L(0^*)$. Second, observe that because M has no string that begins with 0, $D_0(M)$ is the empty set [that's part (e)].

We also need to show that every language N that is unchanged by D_0 is of this form. Let M be the set of strings in N that do not begin with 0. If N is unchanged by D_0, it follows that for every string w in M, $00...0w$ is in N; thus, N includes all the strings of $L(0^*)M$. However, N cannot include a string that is not in $L(0^*)M$. If x were such a string, then we can remove all the 0's at the beginning of x and get some string y that is also in N. But y must also be in M.

Exercise 4.2.8
Let A be a DFA for L. We construct DFA B for $half(L)$. The state of B is of the form $[q,S]$, where:

- q is the state A would be in after reading whatever input B has read so far.
- S is the set of states of A such that A can get from exactly these states to an accepting state by reading any input string whose length is the same as the length of the string B has read so far.

It is important to realize that it is not necessary for B to know how many inputs it has read so far; it keeps this information up-to-date each time it reads a new symbol. The rule that keeps things up to date is: $\delta_B([q,S],a) = [\delta_A(q,a),T]$, where T is the set of states p of A such that there is a transition from p to any state of S on any input symbol. In this manner, the first component continues to simulate A, while the second component now represents states that can reach an accepting state following a path that is one longer than the paths represented by S.

To complete the construction of B, we have only to specify:

- The initial state is $[q_0,F]$, that is, the initial state of A and the accepting states of A. This choice reflects the situation when A has read 0 inputs: it is still in its initial state, and the accepting states are exactly the ones that can reach an accepting state on a path of length 0.
- The accepting states of B are those states $[q,S]$ such that q is in S. The justification is that it is exactly these states that are reached by some string of length n, and there is some other string of length n that will take state q to an accepting state.

Exercise 4.2.13(a)
Start out by complementing this language. The result is the language consisting of all strings of 0's and 1's that are *not* in 0*1*, plus the strings in L_{0n1n}. If we intersect with 0*1*, the result is exactly L_{0n1n}. Since complementation and intersection with a regular set preserve regularity, if the given language were regular then so would be L_{0n1n}. Since we know the latter is false, we conclude the given language is not regular.

Exercise 4.2.14(c)
Change the accepting states to be those for which the first component is an accepting state of A_L and the second is a nonaccepting state of A_M. Then the resulting DFA accepts if and only if the input is in *L* - *M*.

Solutions for Section 4.3

Exercise 4.3.1
Let *n* be the pumping-lemma constant. Test all strings of length between *n* and 2*n*-1 for membership in *L*. If we find even one such string, then *L* is infinite. The reason is that the pumping lemma applies to such a string, and it can be ``pumped" to show an infinite sequence of strings are in *L*.

Suppose, however, that there are no strings in *L* whose length is in the range *n* to 2*n*-1. We claim there are no strings in *L* of length 2*n* or more, and thus there are only a finite number of strings in *L*. In proof, suppose *w* is a string in *L* of length at least 2*n*, and *w* is as short as any string in *L* that has length at least 2*n*. Then the pumping lemma applies to *w*, and we can write *w* = *xyz*, where *xz* is also in *L*. How long could *xz* be? It can't be as long as 2*n*, because it is shorter than *w*, and *w* is as short as any string in *L* of length 2*n* or more. n, because *xz* is at most *n* shorter than *w*. Thus, *xz* is of length between *n* and 2*n*-1, which is a contradiction, since we assumed there were no strings in *L* with a length in that range.

Solutions for Section 4.4

Exercise 4.4.1
Revised 10/23/01.

```
B|x
C|x x
D|x x x
E|x x   x
F|x   x x x
G|  x x x x x
H|x x x x x x x
 --------------
  A B C D E F G
```

	0	**1**
->AG	BF	AG
BF	AG	CE
CE	D	BF
*D	D	AG
H	AG	D

Note, however, that state *H* is inaccessible, so it should be removed, leaving the first four states as the minimum-state DFA.

Solutions for Chapter 5

Revised 11/11/01.

Solutions for Section 5.1

Exercise 5.1.1(a)

```
S -> 0S1 | 01
```

Exercise 5.1.1(b)

```
S -> AB | CD
A -> aA | ε
B -> bBc | E | cD
C -> aCb | E | aA
D -> cD | ε
E -> bE | b
```

To understand how this grammar works, observe the following:

- *A* generates zero or more *a*'s.
- *D* generates zero or more *c*'s.
- *E* generates one or more *b*'s.
- *B* first generates an equal number of *b*'s and *c*'s, then produces either one or more *b*'s (via *E*) or one or more *c*'s (via *cD*). That is, *B* generates strings in *b*c** with an unequal number of *b*'s and *c*'s.
- Similarly, *C* generates unequal numbers of *a*'s then *b*'s.
- Thus, *AB* generates strings in *a*b*c** with an unequal numbers of *b*'s and *c*'s, while *CD* generates strings in *a*b*c** with an unequal number of *a*'s and *b*'s.

Exercise 5.1.2(a)

Leftmost: S => A1B => 0A1B => 00A1B => 001B => 0010B => 00101B => 00101

Rightmost: S => A1B => A10B => A101B => A101 => 0A101 => 00A101 => 00101

Exercise 5.1.5

```
S -> S+S | SS | S* | (S) | 0 | 1 | phi | e
```

The idea is that these productions for *S* allow any expression to be, respectively, the sum (union) of two expressions, the concatenation of two expressions, the star of an expression, a parenthesized expression, or one of the four basis cases of expressions: 0, 1, phi, and ε.

Solutions for Section 5.2

Exercise 5.2.1(a)

```
      S
    / | \
   A  1  B
 / |   / |
0  A  0  B
 / |   / |
0  A  1  B
   |     |
   e     e
```

In the above tree, *e* stands for ε.

Solutions for Section 5.3

Exercise 5.3.2

```
B -> BB | (B) | [B] | ε
```

Exercise 5.3.4(a)

Change production (5) to:

```
ListItem -> <LI> Doc </LI>
```

Solutions for Section 5.4

Exercise 5.4.1

Here are the parse trees:

```
   S                  S
 / |              / / | \
a  S             a S  b  S
 / | \ \           | \   |
a  S  b S          a  S  e
   |    |             |
   e    e             e
```

The two leftmost derivations are: S => aS => aaSbS => aabS => aab and S => aSbS => aaSbS => aabS => aab.

The two rightmost derivations are: S => aS => aaSbS => aaSb => aab and S => aSbS => aSb => aaSb => aab.

Exercise 5.4.3

The idea is to introduce another nonterminal *T* that cannot generate an unbalanced *a*. That strategy corresponds to the usual rule in programming languages that an ``else'' is associated with the closest previous, unmatched ``then.'' Here, we force a *b* to match the previous unmatched *a*. The grammar:

```
S -> aS | aTbS | ε
T -> aTbT | ε
```

Exercise 5.4.6

Alas, it is not. We need to have three nonterminals, corresponding to the three possible ``strengths'' of expressions:

1. A *factor* cannot be broken by any operator. These are the basis expressions, parenthesized expressions, and these expressions followed by one or more *'s.
2. A *term* can be broken only by a *. For example, consider 01, where the 0 and 1 are concatenated, but if we follow it by a *, it becomes 0(1*), and the concatenation has been ``broken'' by the *.
3. An *expression* can be broken by concatenation or *, but not by +. An example is the expression 0+1. Note that if we concatenate (say) 1 or follow by a *, we parse the expression 0+(11) or 0+(1*), and in either case the union has been broken.

The grammar:

```
E -> E+T | T
T -> TF | F
F -> F* | (E) | 0 | 1 | phi | e
```

Solutions for Chapter 6

Solutions for Section 6.1

Exercise 6.1.1(a)

$(q,01,Z_0) \vdash (q,1,XZ_0) \vdash (q,\varepsilon,XZ_0) \vdash (p,\varepsilon,Z_0)$
$\vdash (p,1,Z_0) \vdash (p,\varepsilon,\varepsilon)$

Solutions for Section 6.2

Exercise 6.2.1(a)

We shall accept by empty stack. Symbol *X* will be used to count the 0's on the input. In state *q*, the start state, where we have seen no 1's, we add an *X* to the stack for each 0 seen. The first *X* replaces Z_0, the start symbol. When we see a 1, we go to state *p*, and then only pop the stack, one *X* for each input 1. Formally, the PDA is $(\{q,p\},\{0,1\},\{X,Z_0\},\delta,q,Z_0)$. The rules:

1. $\delta(q,0,Z_0) = \{(q,X)\}$

2. $\delta(q,0,X) = \{(q,XX)\}$
3. $\delta(q,1,X) = \{(p,\varepsilon)\}$
4. $\delta(p,1,X) = \{(p,\varepsilon)\}$

Exercise 6.2.2(a)
Revised 6/20/02.

Begin in start state $q0$, with start symbol Z_0, and immediately guess whether to check for:

1. $i=j=0$ (state $q1$).
2. $i=j>0$ (state $q2$).
3. $j=k$ (state $q3$).

We shall accept by final state; as seen below, the accepting states are $q1$ and $q3$. The rules, and their explanations:

- $\delta(q0,\varepsilon,Z_0) = \{(q1,Z_0), (q2,Z_0), (q3,Z_0)\}$, the initial guess.
- $\delta(q1,c,Z_0) = \{(q1,Z_0)\}$. In case (1), we assume there are no *a*'s or *b*'s, and we consume all *c*'s. State $q1$ will be one of our accepting states.
- $\delta(q2,a,Z_0) = \{(q2,XZ_0)\}$, and $\delta(q2,a,X) = \{(q2,XX)\}$. These rules begin case (2). We use X to count the number of *a*'s read from the input, staying in state $q2$.
- $\delta(q2,b,X) = \delta(q4,b,X) = \{(q4,\varepsilon)\}$. When *b*'s are seen, we go to state $q4$ and pop X's against the *b*'s.
- $\delta(q4,\varepsilon,Z_0) = \{(q1,Z_0)\}$. If we reach the bottom-of-stack marker in state $q4$, we have seen an equal number of *a*'s and *b*'s. We go spontaneously to state $q1$, which will accept and consume all *c*'s, while continuing to accept.
- $\delta(q3,a,Z_0) = \{(q3,Z_0)\}$. This rule begins case (3). We consume all *a*'s from the input. Since $j=k=0$ is possible, state $q3$ must be an accepting state.
- $\delta(q3,b,Z_0) = \{(q5,XZ_0)\}$. When *b*'s arrive, we start counting them and go to state $q5$, which is not an accepting state.
- $\delta(q5,b,X) = \{(q5,XX)\}$. We continue counting *b*'s.
- $\delta(q5,c,X) = \delta(q6,c,X) = \{(q6,\varepsilon)\}$. When *c*'s arrive, we go to state $q6$ and match the *c*'s against the *b*'s.
- $\delta(q6,\varepsilon,Z_0) = \{(q3,\varepsilon)\}$. When the bottom-of-stack marker is exposed in state $q6$, we have seen an equal number of *b*'s and *c*'s. We spontaneously accept in state $q3$, but we pop the stack so we cannot accept after reading more *a*'s.

Exercise 6.2.4
Introduce a new state q, which becomes the initial state. On input ε and the start symbol of P, the new PDA has a choice of popping the stack (thus accepting ε), or going to the start state of P.

Exercise 6.2.5(a)
Revised 4/30/10

$(q0,bab,Z_0)$ |- $(q2,ab,BZ_0)$ |- $(q3,b,Z_0)$ |- $(q1,b,AZ_0)$ |- $(q1,\varepsilon,Z_0)$ |- $(q0,\varepsilon,Z_0)$ |- $(f,\varepsilon,\varepsilon)$

Exercise 6.2.8
Suppose that there is a rule that $(p,X_1X_2...X_k)$ is a choice in $\delta(q,a,Z)$. We create $k-2$ new states $r_1,r_2,...,r_{k-2}$ that simulate this rule but do so by adding one symbol at a time to the stack. That is, replace $(p,X_1X_2...X_k)$ in the rule by $(r_{k-2},X_{k-1}X_k)$. Then create new rules $\delta(r_{k-2},\varepsilon,X_{k-1}) = \{(r_{k-3},X_{k-2}X_{k-1})\}$, and so on, down to $\delta(r_2,\varepsilon,X_3) = \{(r_1,X_2X_3)\}$ and $\delta(r_1,\varepsilon,X_2) = \{(p,X_1X_2)\}$.

Solutions for Section 6.3

Exercise 6.3.1
$(\{q\},\{0,1),\{0,1,A,S\},\delta,q,S)$ where δ is defined by:

1. $\delta(q,\varepsilon,S) = \{(q,0S1), (q,A)\}$
2. $\delta(q,\varepsilon,A) = \{(q,1A0), (q,S), (q,\varepsilon)\}$
3. $\delta(q,0,0) = \{(q,\varepsilon)\}$
4. $\delta(q,1,1) = \{(q,\varepsilon)\}$

Exercise 6.3.3
In the following, S is the start symbol, e stands for the empty string, and Z is used in place of Z_0.

1. S -> $[qZq]$ | $[qZp]$

   The following four productions come from rule (1).

2. $[qZq]$ -> $1[qXq][qZq]$
3. $[qZq]$ -> $1[qXp][pZq]$
4. $[qZp]$ -> $1[qXq][qZp]$
5. $[qZp]$ -> $1[qXp][pZp]$

   The following four productions come from rule (2).

6. $[qXq]$ -> $1[qXq][qXq]$
7. $[qXq]$ -> $1[qXp][pXq]$
8. $[qXp]$ -> $1[qXq][qXp]$
9. $[qXp]$ -> $1[qXp][pXp]$

   The following two productions come from rule (3).

10. $[qXq]$ -> $0[pXq]$
11. $[qXp]$ -> $0[pXp]$

   The following production comes from rule (4).

12. *[qXq] -> e*

The following production comes from rule (5).

13. *[pXp] -> 1*

The following two productions come from rule (6).

14. *[pZq] -> 0[qZq]*
15. *[pZp] -> 0[qZp]*

Exercise 6.3.6
Convert *P* to a CFG, and then convert the CFG to a PDA, using the two constructions given in Section 6.3. The result is a one-state PDA equivalent to *P*.

Solutions for Section 6.4

Exercise 6.4.1(b)
Not a DPDA. For example, rules (3) and (4) give a choice, when in state *q*, with 1 as the next input symbol, and with *X* on top of the stack, of either using the 1 (making no other change) or making a move on ε input that pops the stack and going to state *p*.

Exercise 6.4.3(a)
Suppose a DPDA *P* accepts both *w* and *wx* by empty stack, where *x* is not ε (i.e., *N(P)* does not have the prefix property). Then (q_0,wxZ_0) |-* *(q,x,ε)* for some state *q*, where q_0 and Z_0 are the start state and symbol of *P*. It is not possible that *(q,x,ε)* |-* *(p,ε,ε)* for some state *p*, because we know *x* is not ε, and a PDA cannot have a move with an empty stack. This observation contradicts the assumption that *wx* is in *N(P)*.

Exercise 6.4.3(c)
Modify *P'* in the following ways to create DPDA *P*:

1. Add a new start state and a new start symbol. *P*, with this state and symbol, pushes the start symbol of *P'* on top of the stack and goes to the start state of *P'*. The purpose of the new start symbol is to make sure *P* doesn't accidentally accept by empty stack.
2. Add a new ``popping state'' to *P*. In this state, *P* pops every symbol it sees on the stack, using ε input.
3. If *P'* enters an accepting state, *P* enters the popping state instead.

As long as *L(P')* has the prefix property, then any string that *P'* accepts by final state, *P* will accept by empty stack.

Solutions for Chapter 7

Revised 2/18/05.

Solutions for Section 7.1

Exercise 7.1.1

A and *C* are clearly generating, since they have productions with terminal bodies. Then we can discover *S* is generating because of the production *S*->*CA*, whose body consists of only symbols that are generating. However, *B* is not generating. Eliminating *B*, leaves the grammar

```
S -> CA
A -> a
C -> b
```

Since *S*, *A*, and *C* are each reachable from *S*, all the remaining symbols are useful, and the above grammar is the answer to the question.

Exercise 7.1.2

Revised 6/27/02.

a)

Only *S* is nullable, so we must choose, at each point where *S* occurs in a body, to eliminate it or not. Since there is no body that consists only of *S*'s, we do not have to invoke the rule about not eliminating an entire body. The resulting grammar:

```
S -> ASB | AB
A -> aAS | aA | a
B -> SbS | bS | Sb | b | A | bb
```

b)

The only unit production is *B* -> *A*. Thus, it suffices to replace this body *A* by the bodies of all the *A*-productions. The result:

```
S -> ASB | AB
A -> aAS | aA | a
B -> SbS | bS | Sb | b | aAS | aA | a | bb
```

c)

Observe that *A* and *B* each derive terminal strings, and therefore so does *S*. Thus, there are no useless symbols.

d)

Introduce variables and productions *C* -> *a* and *D* -> *b*, and use the new variables in all bodies that are not a single terminal:

```
S -> ASB | AB
A -> CAS | CA | a
B -> SDS | DS | SD | b | CAS | CA | a | DD
C -> a
D -> b
```

Finally, there are bodies of length 3; one, *CAS*, appears twice. Introduce new variables *E*, *F*, and *G* to split these bodies, yielding the CNF grammar:

```
S -> AE | AB
A -> CF | CA | a
B -> SG | DS | SD | b | CF | CA | a | DD
C -> a
D -> b
E -> SB
F -> AS
G -> DS
```

Exercise 7.1.10

It's not possible. The reason is that an easy induction on the number of steps in a derivation shows that every sentential form has odd length. Thus, it is not possible to find such a grammar for a language as simple as {00}.

To see why, suppose we begin with start symbol *S* and try to pick a first production. If we pick a production with a single terminal as body, we derive a string of length 1 and are done. If we pick a body with three variables, then, since there is no way for a variable to derive epsilon, we are forced to derive a string of length 3 or more.

Exercise 7.1.11(b)

The statement of the entire construction may be a bit tricky, since you need to use the construction of part (c) in (b), although we are not publishing the solution to (c). The construction for (b) is by induction on *i*, but it needs to be of the stronger statement that if an A_i-production has a body beginning with A_j, then $j > i$ (i.e., we use part (c) to eliminate the possibility that $i=j$).

Basis: For $i = 1$ we simply apply the construction of (c) for $i = 1$.

Induction: If there is any production of the form $A_i \rightarrow A_1 \ldots$, use the construction of (a) to replace A_1. That gives us a situation where all A_i production bodies begin with at least A_2 or a terminal. Similarly, replace initial A_2's using (a), to make A_3 the lowest possible variable beginning an A_i-production. In this manner, we eventually guarantee that the body of each A_i-production either begins with a terminal or with A_j, for some $j >= i$. A use of the construction from (c) eliminates the possibility that $i = j$.

Exercise 7.1.11(d)

As per the hint, we do a backwards induction on *i*, that the bodies of A_i productions can be made to begin with terminals.

Basis: For $i = k$, there is nothing to do, since there are no variables with index higher than *k* to begin the body.

Induction: Assume the statement for indexes greater than *i*. If an A_i-production begins with a variable, it must be A_j for some $j > i$. By the induction hypothesis, the A_j-productions all have bodies beginning with terminals now. Thus, we may use the construction (a) to replace the initial A_j, yielding only A_i-productions whose bodies begin with terminals.

After fixing all the A_i-productions for all i, it is time to work on the B_i-productions. Since these have bodies that begin with either terminals or A_j for some j, and the latter variables have only bodies that begin with terminals, application of construction (a) fixes the B_j's.

Solutions for Section 7.2

Exercise 7.2.1(a)

Let n be the pumping-lemma constant and consider string $z = a^n b^{n+1} c^{n+2}$. We may write $z = uvwxy$, where v and x, may be ``pumped,'' and $|vwx| <= n$. If vwx does not have c's, then uv^3wx^3y has at least $n+2$ a's or b's, and thus could not be in the language.

If vwx has a c, then it could not have an a, because its length is limited to n. Thus, uwy has n a's, but no more than $2n+2$ b's and c's in total. Thus, it is not possible that uwy has more b's than a's and also has more c's than b's. We conclude that uwy is not in the language, and now have a contradiction no matter how z is broken into $uvwxy$.

Exercise 7.2.1(d)

Let n be the pumping-lemma constant and consider $z = 0^n 1^{n2}$. We break $Z = uvwxy$ according to the pumping lemma. If vwx consists only of 0's, then uwy has n^2 1's and fewer than n 0's; it is not in the language. If vwx has only 1's, then we derive a contradiction similarly. If either v or x has both 0's and 1's, then uv^2wx^2y is not in 0^*1^*, and thus could not be in the language.

Finally, consider the case where v consists of 0's only, say k 0's, and x consists of m 1's only, where k and m are both positive. Then for all i, $uv^{i+1}wx^{i+1}y$ consists of $n + ik$ 0's and $n^2 + im$ 1's. If the number of 1's is always to be the square of the number of 0's, we must have, for some positive k and m: $(n+ik)^2 = n^2 + im$, or $2ink + i^2k^2 = im$. But the left side grows quadratically in i, while the right side grows linearly, and so this equality for all i is impossible. We conclude that for at least some i, $uv^{i+1}wx^{i+1}y$ is not in the language and have thus derived a contradiction in all cases.

Exercise 7.2.2(b)

It could be that, when the adversary breaks $z = uvwxy$, $v = 0^k$ and $x = 1^k$. Then, for all i, uv^iwx^iy is in the language.

Exercise 7.2.2(c)

The adversary could choose $z = uvwxy$ so that v and x are single symbols, on either side of the center. That is, $|u| = |y|$, and w is either epsilon (if z is of even length) or the single, middle symbol (if z is of odd length). Since z is a palindrome, v and x will be the same symbol. Then uv^iwx^iy is always a palindrome.

Exercise 7.2.4

The hint turns out to be a bad one. The easiest way to prove this result starts with a string $z = 0^n1^n0^n1^n$ where the middle two blocks are distinguished. Note that vwx cannot include 1's from the second block and also 1's from the fourth block, because then vwx would have all n distinguished 0's and thus at least $n+1$ distinguished symbols. Likewise, it cannot have 0's from both blocks of 0's. Thus, when we pump v and x, we must get an imbalance between the blocks of 1's or the blocks of 0's, yielding a string not in the language.

Solutions for Section 7.3

Exercise 7.3.1(a)
For each variable A of the original grammar G, let A' be a new variable that generates *init* of what A generates. Thus, if S is the start symbol of G, we make S' the new start symbol.

If $A \to BC$ is a production of G, then in the new grammar we have $A \to BC$, $A' \to BC'$, and $A' \to B'$. If $A \to a$ is a production of G, then the new grammar has $A \to a$, $A' \to a$, and $A' \to$ *epsilon*.

Exercise 7.3.1(b)
The construction is similar to that of part (a), but now A' must be designed to generate string w if and only if A generates wa. That is, A''s language is the result of applying $/a$ to A's language.

If G has production $A \to BC$, then the new grammar has $A \to BC$ and $A' \to BC'$. If G has $A \to b$ for some $b != a$, then the new grammar has $A \to b$, but we do not add any production for A'. If G has $A \to a$, then the new grammar has $A \to a$ and $A' \to$ *epsilon*.

Exercise 7.3.3(a)
Consider the language $L = \{a^ib^jc^k \mid 1 <= i \text{ and } 1 <= j \text{ and } (i <= k \text{ or } j <= k)\}$. L is easily seen to be a CFL; you can design a PDA that guesses whether to compare the a's or b's with the c's. However, $min(L) = \{a^ib^jc^k \mid k = min(i,j)\}$. It is also easy to show, using the pumping lemma, that this language is not a CFL. Let n be the pumping-lemma constant, and consider $z = a^nb^nc^n$.

Exercise 7.3.4(b)
If we start with a string of the form 0^n1^n and intersperse any number of 0's, we can obtain any string of 0's and 1's that begins with at least as many 0's as there are 1's in the entire string.

Exercise 7.3.4(c)
Given DFA's for L_1 and L_2, we can construct an NFA for their shuffle by using the product of the two sets of states, that is, all states $[p,q]$ such that p is a state of the automaton for L_1, and q is a state of the automaton for L_2. The start state of the automaton for the shuffle consists of the start states of the two automata, and its accepting states consist of pairs of accepting states, one from each DFA.

The NFA for the shuffle guesses, at each input, whether it is from L_1 or L_2. More formally, $\delta([p,q],a) = \{[\delta_1(p,a),q], [p,\delta_2(q,a)]\}$, where δ_i is the transition function for the DFA for L_i (i = 1 or 2). It is then an easy induction on the length of w that $\delta\text{-}hat([p_0,q_0],w)$ contains $[p,q]$ if and only if w is the shuffle of some x and y, where $\delta_1\text{-}hat(p_0,x) = p$ and $\delta_2\text{-}hat(q_0,y) = q$.

Exercise 7.3.5

a)

Consider the language of regular expression **(0l)***. Its permutations consist of all strings with an equal number of 0's and 1's, which is easily shown not regular. In proof, use the pumping lemma for regular languages, let *n* be the pumping-lemma constant, and consider string 0^n1^n.

b)

The language of **(012)*** serves. Its permutations are all strings with an equal number of 0's 1's, and 2's. We can prove this language not to be a CFL by using the pumping lemma on $0^n1^n2^n$, where *n* is the pumping-lemma constant.

c)

Assume the alphabet of regular language *L* is {0,1}. We can design a PDA *P* to recognize *perm(L)*, as follows. *P* simulates the DFA *A* for *L* on an input string that it guesses. However, *P* must also check that its own input is a permutation of the guessed string. Thus, each time *P* guesses an input for *A*, it also reads one of its own symbols. *P* uses its stack to remember whether it has seen more 0's than it has guessed, or seen more 1's than it has guessed. It does so by keeping a stack string with a bottom-of-stack marker and either as many more 0's as it has seen than guessed, or as many more 1's as it has seen than guessed.

For instance, if *P* guesses 0 as an input for *A* but sees a 1 on its own input, then *P*:

1. If 0 is the top stack symbol, then push another 0.
2. If 1 is the top stack symbol, then pop the stack.
3. If Z_0, the bottom-of-stack marker is on top, push a 0.

In addition, if *P* exposes the bottom-of-stack marker, then it has guessed, as input to *A*, a permutation of the input *P* has seen. Thus, if *A* is in an accepting state, *P* has a choice of move to pop its stack on epsilon input, thus accepting by empty stack.

Solutions for Section 7.4

Exercise 7.4.1(a)

If there is any string at all that can be ``pumped," then the language is infinite. Thus, let *n* be the pumping-lemma constant. If there are no strings as long as *n*, then surely the language is finite. However, how do we tell if there is some string of length *n* or more? If we had to consider all such strings, we'd never get done, and that would not give us a decision algorithm.

The trick is to realize that if there is any string of length *n* or more, then there will be one whose length is in the range *n* through *2n-1*, inclusive. For suppose not. Let *z* be a string that is as short as possible, subject to the constraint that $|z| >= n$. If $|z| < 2n$, we are done; we have found a string in the desired length range. If $|z| >= 2n$, use the pumping lemma to write $z = uvwxy$. We know *uwy* is also in the language, but because $|vwx| <= n$, we know $|z| > |uwy| >= n$. That contradicts our assumption that *z* was as short as possible among strings of length *n* or more in the language.

We conclude that $|z| < 2n$. Thus, our algorithm to test finiteness is to test membership of all strings of length between *n* and *2n-1*. If we find one, the language is infinite, and if not, then the language is finite.

Exercise 7.4.3(a)
Here is the table:

```
   {S,A,C}
   {B}      {B}
   {B}      {S,C}    {B}
   {S,C}    {S,A}    {S,C}    {S,A}
   {A,C}    {B}      {A,C}    {B}      {A,C}
------------------------------------------
     a        b        a        b        a
```

Since *S* appears in the upper-left corner, *ababa* is in the language.

Exercise 7.4.4
The proof is an induction on *n* that if $A \Rightarrow^* w$, for any variable *A*, and $|w| = n$, then all parse trees with *A* at the root and yield *w* have $2n-1$ interior nodes.

Basis: $n = 1$. The parse tree must have a root with variable *A* and a leaf with one terminal. This tree has $2n-1 = 1$ interior node.

Induction: Assume the statement for strings of length less than *n*, and let $n > 1$. Then the parse tree begins with *A* at the root and two children labeled by variables *B* and *C*. Then we can write $w = xy$, where $B \Rightarrow^* x$ and $C \Rightarrow^* y$. Also, *x* and *y* are each shorter than length *n*, so the inductive hypothesis applies to them, and we know that the parse trees for these derivations have, respectively, $2|x|-1$ and $2|y|-1$ interior nodes.

Thus, the parse tree for $A \Rightarrow^* w$ has one (for the root) plus the sum of these two quantities number of interior nodes, or $2(|x|+|y|-1)$ interior nodes. Since $|x|+|y| = |w| = n$, we are done; the parse tree for $A \Rightarrow^* w$ has $2n-1$ interior nodes.

Solutions for Chapter 8

Solutions for Section 8.1

Exercise 8.1.1(a)
We need to take a program *P* and modify it so it:

1. Never halts unless we explicitly want it to, and
2. Halts whenever it prints `hello, world.`

For (1), we can add a loop such as `while(1){x=x;}` to the end of `main`, and also at any point where `main` returns. That change catches the normal ways a program can halt, although it doesn't address the problem of a program that halts because some exception such as division by 0 or an attempt to read an unavailable device. Technically, we'd have to replace all of the exception handlers in the run-time environment to cause a loop whenever an exception occurred.

For (2), we modify *P* to record in an array the first 12 characters printed. If we find that they are `hello, world.`, we halt by going to the end of `main` (past the point where the while-loop has been installed).

Solutions for Section 8.2

Exercise 8.2.1(a)
To make the ID's clearer in HTML, we'll use *[q0]* for q_0, and similarly for the other states.

[q0]00 |- *X[q1]0* |- *X0[q1]*

The TM halts at the above ID.

Exercise 8.2.2(a)
Here is the transition table for the TM:

state	**0**	**1**	**B**	**X**	**Y**
q0	(q2,X,R)	(q1,X,R)	(qf,B,R)	-	(q0,Y,R)
q1	(q3,Y,L)	(q1,1,R)	-	-	(q1,Y,R)
q2	(q2,0,R)	(q3,Y,L)	-	-	(q2,Y,R)
q3	(q3,0,L)	(q3,1,L)	-	(q0,X,R)	(q3,Y,L)
qf	-	-	-	-	-

In explanation, the TM makes repeated excursions back and forth along the tape. The symbols *X* and *Y* are used to replace 0's and 1's that have been cancelled one against another. The difference is that an *X* guarantees that there are no unmatched 0's and 1's to its left (so the head never moves left of an *X*), while a *Y* may have 0's or 1's to its left.

Initially in state *q0*, the TM picks up a 0 or 1, remembering it in its state (*q1* = found a 1; *q2* = found a 0), and cancels what it found with an *X*. As an exception, if the TM sees the blank in state *q0*, then all 0's and 1's have matched, so the input is accepted by going to state *qf*.

In state *q1*, the TM moves right, looking for a 0. If it finds it, the 0 is replaced by *Y*, and the TM enters state *q3* to move left an look for an *X*. Similarly, state *q2* looks for a 1 to match against a 0.

In state *q3*, the TM moves left until it finds the rightmost *X*. At that point, it enters state *q0* again, moving right over *Y*'s until it finds a 0, 1, or blank, and the cycle begins again.

Exercise 8.2.4
These constructions, while they can be carried out using the basic model of a TM are much clearer if we use some of the tricks of Sect. 8.3.

For part (a), given an input *[x,y]* use a second track to simulate the TM for *f* on the input *x*. When the TM halts, compare what it has written with *y*, to see if *y* is indeed *f(x)*. Accept if so.

For part (b), given *x* on the tape, we need to simulate the TM *M* that recognizes the graph of *f*. However, since this TM may not halt on some inputs, we cannot simply try all *[x,i}* to see which value of *i* leads to acceptance by *M*. The reason is that, should we work on some value of *i* for which *M* does not halt, we'll never advance to the correct value of *f(x)*. Rather, we consider, for various combinations of *i* and *j*, whether *M* accepts *[x,i]* in *j* steps. If we consider *(i,j)* pairs in order of their sum (i.e., (0,1), (1,0), (0,2), (1,1), (2,0), (0,3),...) then eventually we shall simulate *M* on *[x,f(x)]* for a sufficient number of steps that *M* reaches acceptance. We need only wait until we consider pairs whose sum is *f(x)* plus however many steps it takes *M* to accept *[x,f(x)]*. In this manner, we can discover what *f(x)* is, write it on the tape of the TM that we have designed to compute *f(x)*, and halt.

Now let us consider what happens if *f* is not defined for some arguments. Part (b) does not change, although the constructed TM will fail to discover *f(x)* and thus will continue searching forever. For part (a), if we are given *[x,y]*, and *f* is not defined on *x*, then the TM for *f* will never halt on *x*. However, there is nothing wrong with that. Since *f(x)* is undefined, surely *y* is not *f(x)*. Thus, we do not want the TM for the graph of *f* to accept *[x,y]* anyway.

Exercise 8.2.5(a)
This TM only moves right on its input. Moreover, it can only move right if it sees alternating 010101... on the input tape. Further, it alternates between states q_0 and q_1 and only accepts if it sees a blank in state q_1. That in turn occurs if it has just seen 0 and moved right, so the input must end in a 0. That is, the language is that of regular expression **(01)*0**.

Solutions for Section 8.3

Exercise 8.3.3
Here is the subroutine. Note that because of the technical requirements of the subroutine, and the fact that a TM is not allowed to keep its head stationary, when we see a non-0, we must enter state *q3*, move right, and then come back left in state *q4*, which is the ending state for the subroutine.

state	**0**	**1**	**B**
q1	(q2,0,R)	-	-
q2	(q2,0,R)	(q3,1,R)	(q3,B,R)
q3	(q4,0,L)	(q4,1,L)	(q4,B,L)

Now, we can use this subroutine in a TM that starts in state *q0*. If this TM ever sees the blank, it accepts in state *qf*. However, whenever it is in state *q0*, it knows only that it has not seen a 1 immediately to its right. If it is scanning a 0, it must check (in state *q5*) that it does not have a blank immediately to its right; if it does, it accepts. If it sees 0 in state *q5,* it comes back to the previous 0 and calls the subroutine to skip to the next non-0. If it sees 1 in state *q5*, then it has seen 01, and uses state *q6* to check that it doesn't have another 1 to the right.

In addition, the TM in state *q4* (the final state of the subroutine), accepts if it has reached a blank, and if it has reached a 1 enters state *q6* to make sure there is a 0 or blank following. Note that states *q4* and *q5* are really the same, except that in *q4* we are certain we are not scanning a 0. They could be combined into one state. Notice also that the subroutine is not a perfect match for what is needed, and there is some unnecessary jumping back and forth on the tape. Here is the remainder of the transition table.

state	0	1	B
q0	(q5,0,R)	(q6,1,R)	(qf,B,R)
q5	(q1,0,L)	(q6,1,R)	(qf,B,R)
q6	(q0,0,R)	-	(qf,B,R)
q4	-	(q6,1,R)	(qf,B,R)

Solutions for Section 8.4

Exercise 8.4.2(a)
For clarity, we put the state in square brackets below. Notice that in this example, we never branch. *[q_0]01* |- *1[q_0]1* |- *10[q_1]* |- *10B[q_2]*

Exercise 8.4.3(a)
We'll use a second tape, on which the guess *x* is stored. Scan the input from left yo right, and at each cell, guess whether to stay in the initial state (which does the scanning) or go to a new state that copies the next 100 symbols onto the second tape. The copying is done by a sequence of 100 state, so exactly 100 symbols can be placed on tape 2.

Once the copying is done, retract the head of tape 2 to the left end of the 100 symbols. Then, continue moving right on tape 1, and at each cell guess either to continue moving right or to guess that the second copy of *x* begins. In the latter case, compare the next 100 symbols on tape 1 with the 100 symbols on tape 2. If they all match, then move right on tape 1 and accept as soon as a blank is seen.

Exercise 8.4.5
For part (a), guess whether to move left or right, entering one of two different states, each responsible for moving in one direction. Each of these states proceeds in its direction, left or right, and if it sees a $ it enters state *p*. Technically, the head has to move off the $, entering another state, and then move back to the $, entering state *p* as it does so.

Part (b), doing the same thing deterministically, is trickier, since we might start off in the wrong direction and travel forever, never seeing the $. Thus, we have to oscillate, using left and right endmarkers *X* and *Y*, respectively, to mark how far we have traveled on a second track. Start moving one cell left and leave the *X*. Then, move two cells right and leave the *Y*. Repeatedly move left to the *X*, move the *X* one more cell left, go right to the *Y*, move it once cell right, and repeat.

Eventually, we shall see the $. At this time, we can move left or right to the other endmarker, erase it, move back to the end where the $ was found, erase the other endmarker, and wind up at the $.

Exercise 8.4.8(a)

There would be 10 tracks. Five of the tracks hold one of the symbols from the tape alphabet, so there are 7^5 ways to select these tracks. The other five tracks hold either *X* or blank, so these tracks can be selected in 2^5 ways. The total number of symbols is thus $7^5 * 2^5 = 537{,}824$.

Exercise 8.4.8(b)

The number of symbols is the same. The five tracks with tape symbols can still be chosen in 7^5 ways. The sixth track has to tell which subset of the five tapes have their head at that position. There are 2^5 possible subsets, and therefore 32 symbols are needed for the 6th track. Again the number of symbols is $7^5 * 2^5$.

Solutions for Section 8.5

Exercise 8.5.1(c)

In principle, any language that is recursively enumerable can be recognized by a 2-counter machine, but how do we design a comprehensible answer for a particular case? As the *a*'s are read, count them with both counters. Then, when *b*'s enter, compare them with one counter, and accept if they are the same. Continue accepting as long as *c*'s enter. If the numbers of *a*'s and *b*'s differ, then compare the second counter with the number of *c*'s, and accept if they match.

Solutions for Chapter 9

Revised 5/2/01.

Solutions for Section 9.1

Exercise 9.1.1(a)

37 in binary is 100101. Remove the leading 1 to get the string 00101, which is thus w_{37}.

Exercise 9.1.3(a)

Suppose this language were accepted by some TM *M*. We need to find an *i* such that $M = M_{2i}$. Fortunately, since all the codes for TM's end in a 0, that is not a problem; we just convert the specification for *M* to a code in the manner described in the section.

We then ask if w_i is accepted by M_{2i}? If so, then w_i is not accepted by *M*, and therefore not accepted by M_{2i}, which is the same TM. Similarly, if w_i is not accepted by M_{2i}, then w_i is accepted by *M*, and therefore by M_{2i}. Either way, we reach a contradiction, and conclude that *M* does not exist.

Solutions for Section 9.2

Exercise 9.2.2(a)
$A(2,1) = A(A(1,1),0)$ [rule 4] = $A(A(A(0,1),0),0)$ [rule 4] = $A(A(1,0),0)$ [rule 1] = $A(2,0)$ [rule 2] = 4 [rule 3].

Exercise 9.2.3(a)
Let's keep i, the integer in unary, whose square we have most recently printed on the output tape, on tape 1, and keep i^2 on tape 2, also in unary. Initially, $i = 0$. Repeatedly do the following:

1. Add 1 to tape 1; now we have $i+1$ there.
2. Copy tape 1 to tape 2 twice, and remove one to change i^2 to $i^2 + 2(i+1) - 1 = (i+1)^2$.
3. Copy tape 2 to the output as a block of 0's and append a 1.

Exercise 9.2.4
By symmetry, if we can prove L_1 is recursive, we can prove any of the languages to be recursive. Take TM's M_1, M_2,...,M_k for each of the languages L_1, L_2,...,L_k, respectively. Design a TM M with k tapes that accepts L_1 and always halts. M copies its input to all the tapes and simulates M_I on the ith tape. If M_1 accepts, then M accepts. If any of the other TM's accepts, M halts without accepting. Since exactly one of the M_i's will accept, M is sure to halt.

Exercise 9.2.5
Note that the new language defined in the displayed text should be *L'*; it is different from the given language *L*, of course. Also, we'll use -*L* for the complement of *L* in what follows.

Suppose *L'* were RE. Then we could design a TM *M* for -*L* as follows. Given input *w*, *M* changes its input to *1w* and simulates the hypothetical TM for *L'*. If that TM accepts, then *w* is in -*L*, so *M* should accept. If the TM for *L'* never accepts, then neither does *M*. Thus, *M* would accept exactly -*L*, which contradicts the fact that -*L* is not RE. We conclude that *L'* is not RE.

Exercise 9.2.6(a)
To test whether an input *w* is in the union of two recursive languages *L1* and *L2*, we design a TM to copy its input *w* onto a second tape. It then simulates the halting TM for *L1* on one tape and the halting TM for *L2* on the other. If either accepts, then we accept. If both halt without accepting, we halt without accepting. Thus, the union is accepted by a TM that always halts.

In the case where *L1* and *L2* are RE, do the same, and accept if either accepts. The resulting TM accepts the union, although it may not halt. We conclude that both the recursive languages and the RE languages are closed under union.

Exercise 9.2.6(e)
Consider the case where *L* is RE. Design a NTM *M* for *h(L)*, as follows. Suppose *w* is the input to *M*. On a second tape, *M* guesses some string *x* over the alphabet of *L*, checks that *h(x) = w*, and simulates the TM for *L* on *x*, if so. If *x* is accepted, then *M* accepts *w*. We conclude that the RE languages are closed under homomorphism.

However, the recursive languages are not closed under homomorphism. To see why, consider the particular language *L* consisting of strings of the form (M,w,c^i), where *M* is a coded Turing machine with binary input alphabet, *w* is a binary string, and *c* is a symbol not appearing elsewhere. The string is in *L* if and only if *M* accepts *w* after making at most *i* moves. Clearly *L* is recursive; we may simulate *M* on *w* for *i* moves and then decide whether or not to accept. However, if we apply to *L* the homomorphism that maps the symbols other than *c* to themselves, and maps *c* to ε, we find that *h(L)* is the universal language, which we called L_u. We know that L_u is not recursive.

Solutions for Section 9.3

Exercise 9.3.1
The property of languages ``contains all the palindromes'' is a nontrivial property, since some languages do and others don't. Thus, by Rice's theorem, the question is undecidable.

Exercise 9.3.4(d)
Revised 7/19/05

We shall reduce the problem L_e (does a TM accept the empty language?) to the question at hand: does a TM accept a language that is its own reverse? Given a TM *M*, we shall construct a nondeterministic TM *M'*, which accepts either the empty language (which is its own reverse), or the language {01} (which is not its own reverse). We shall make sure that if *L(M)* is empty, then *L(M')* is its own reverse (the empty language, in particular), and if *L(M)* is not empty, then *L(M')* is not its own reverse. *M'* works as follows:

1. First, check that its input is 01, and reject if not.
2. Guess an input *w* for *M*.
3. Simulate *M* on *w*. If *M* accepts, then *M'* accepts its own input, 01.

Thus, if *L(M)* is nonempty, *M'* will guess some string *M* accepts and therefore accept 01. If *L(M)* is empty, then all guesses by *M'* fail to lead to acceptance by *M*, so *M'* never accepts 01 or any other string.

Exercise 9.3.6(a)
After making *m* transitions (not *m+1* as suggested by the hint), the TM will have been in *m+1* different states. These states cannot all be different. Thus, we can find some repeating state, and the moves of the TM look like $[q_0]$ |-* *q* |-* *q* |-* ..., where the central |-* represents at least one move. Note that we assume the tape remains blank; if not then we know the TM eventually prints a nonblank. However, if it enters a loop without printing a nonblank, then it will remain forever in that loop and never print a nonblank. Thus, we can decide whether the TM ever prints a nonblank by simulating it for *m* moves, and saying ``yes'' if and only if it prints a nonblank during that sequence of moves.

Exercise 9.3.7(a)
We reduce the complement of L_u to this problem, which is the complement of the halting problem for Turing Machines. The crux of the argument is that we can convert any TM *M* into another TM *M'*, such that *M'* halts on input *w* if and only if *M* accepts *w*. The construction of *M'* from *M* is as follows:

1. Make sure that *M'* does not halt unless *M* accepts. Thus, add to the states of *M* a new state *p*, in which *M'* runs right, forever; i.e., $\delta(p,X) = (p,X,R)$ for all tape symbols *X*. If *M* would halt without accepting, say $\delta(q,Y)$ is undefined for some nonaccepting state *q*, then in *M'*, make $\delta(q,Y) = (p,Y,R)$; i.e., enter the right-moving state and make sure *M'* does not halt.
2. If *M* accepts, then *M'* must halt. Thus, if *q* is an accepting state of *M*, then in *M'*, $\delta(q,X)$ is made undefined for all tape symbols *X*.
3. Otherwise, the moves of *M'* are the same as those of *M*.

The above construction reduces the complement of L_u to the complement of the halting problem. That is, if *M* accepts *w*, then *M'* halts on *w*, and if not, then not. Since the complement of L_u is non-RE, so is the complement of the halting problem.

Exercise 9.3.8(a)
We'll show this problem not to be RE by reducing the problem of Exercise 9.3.7(a), the ``nonhalting" problem to it. Given a pair *(M,w)*, we must construct a TM *M'*, such that *M'* halts on every input if and only if *M* does not halt on *w*. Here is how *M'* works:

1. Given an input *x* of length *n*, *M'* simulates *M* on *w* for *n* steps.
2. If during that time, *M* halts, then *M'* enters a special looping state [as discussed in the solution to Exercise 9.3.7(a)] and *M'* does not halt on its own input *x*.
3. However, if *M* does not halt on *w* after *n* steps, then *M'* halts.

Thus, *M'* halts on all inputs if and only if *M* does not halt on *w*. Since we proved in the solution to Exercise 9.3.7(a) that the problem of telling whether *M* does not halt on *w* is non-RE, it follows that the question at hand --- whether a given TM halts on all inputs --- must not be RE either.

Exercise 9.3.8(d)
This language is the complement of the language of Exercise 9.3.8(a), so it is surely not recursive. But is it RE? We can show it isn't by a simple reduction from the nonhalting problem. Given *(M,w)*, construct *M'* as follows:

1. *M'* ignores its own input and simulates *M* on *w*.
2. If *M* halts, *M'* halts on its own input. However, if *M* never halts on *w*, then *M'* will never halt on its own input.

As a result, *M'* fails to halt on at least one input (in fact, on all inputs) if *M* fails to halt on *w*. If *M* halts on *w*, then *M'* halts on all inputs.

Solutions for Section 9.4

Exercise 9.4.1(a)

There is no solution. First, a solution would have to start with pair 1, because that is the only pair where one is a prefix of the other. THus, our partial solution starts:

```
A: 01
B: 011
```

Now, we need a pair whose *A*-string begins with 1, and that can only be pair 3. The partial solution becomes

```
A: 0110
B: 01100
```

Now, we need a pair whose *A*-string begins with 0, and either pair 1 or pair 2 might serve. However, trying to extend the solution with pair 1 gives us:

```
A: 011001
B: 01100011
```

while extending by pair 2 yields:

```
A: 0110001
B: 0110010
```

In both cases, there is a mismatch, and we conclude no solution exists.

Exercise 9.4.3

The problem is decidable by the following, fairly simple algorithm. First, if all the *A*-strings are strictly longer than their corresponding *B*-strings, then there is surely no solution. Neither is there a solution in the opposite case, where all the *B*-strings are strictly longer than their corresponding *A*-strings.

We claim that in all other cases, there is a solution. If any corresponding pair of strings are the same length, then they are identical, and so just that pair is a solution. The only possibility remains has at least one pair, say *i*, with the *A*-string longer than the *B*-string, say by *m* symbols, and another pair, say *j*, where the *B*-string is longer than the *A*-string, say by *n* symbols. Then $i^n j^m$, i.e., *n* uses of pair *i* followed by *m* uses of pair *j* is a solution. In proof, it is easy to check that both the *A*- and *B*-strings that result have the same length. Since there is only one symbol, these strings are therefore identical.

Solutions for Section 9.5

Exercise 9.5.1

Given an instance *(A,B)* of PCP, construct the grammar G_A as in the text. Also, construct a grammar G_{BR}, that is essentially G_B, but with the bodies of all productions reversed, so its language is the reverse of the language of G_B. Assume the start symbols of these grammars are *A* and *B*, they contain no variables in common, and that *c* is a terminal that does not appear in these grammars.

Construct a new grammar G with all the productions of G_A and G_{BR}, plus the production S -> AcB. Then a solution to the PCP instance yields a string y such that y is generated by G_A and y^R is generated by G_{BR}. Thus, G generates the palindrome ycy^R. However, any palindrome generated by G must have the c in the middle and thus implies a solution to the PCP instance, that is, a string y that appears in $L(G_A)$ while y^R appears in $L(G_{BR})$ [and therefore y appears in $L(G_B)$].

Solutions for Chapter 10

Revised 6/30/01.

Solutions for Section 10.1

Exercise 10.1.1(a)
The MWST would then be the line from 1 to 2 to 3 to 4.

Exercise 10.1.3
For every problem P in **NP** there would be some polynomial p that bounded the running time (and therefore the output length) of the reduction from P to the NP-complete problem in question. That would imply an algorithm for P that ran in time $O(p(n) + [p(n)]^{\log_2 p(n)})$. The first term, $p(n)$, can be neglected. The exponent is $k \log n$ for some constant k. Moreover, $p(n)^k$ is at most $n^{k'}$ for some other constant k'. Thus, we can say that there would be some constant c such that problem P could be solved in time $O(n^{c \log_2 n})$.

Exercise 10.1.5(a)
Given (G,A,B), construct $G1$ and $G2$ to be G, with start symbols A and B, respectively. Since this transformation is essentially copying, it can be performed in linear time, and is therefore surely a polynomial-time reduction.

Exercise 10.1.5(c)
Absolutely nothing! Part of any NP-completeness proof is a part that shows the problem to be in **NP**. These problems are, in fact, undecidable, and therefore surely not in **NP**.

Exercise 10.1.6(b)
Test for membership in one language and then, if the input is not in the first, test for membership in the second. The time taken is no more than the sum of the times taken to recognize each language. Since both ar in **P**, then can each be recognized in polynomial time, and the sum of polynomials is a polynomial. Thus, their union is in **P**.

Exercise 10.1.6(c)
Let $L1$ and $L2$ be languages in **P**, and suppose we want to recognize their concatenation. Suppose we are given an input of length n. For each i between 1 and $n-1$, test whether positions 1 through i holds a string in $L1$ and

positions *i+1* through *n* hold a string in *L2*. If so, accept; the input is in *L1L2*. If the test fails for all *i*, reject the input.

The running time of this test is at most *n* times the sum of the running times of the recognizers for *L1* and *L2*. Since the latter are both polynomials, so is the running time for the TM just described.

Exercise 10.1.6(f)
Given a polynomial-time TM *M* for *L*, we can modify *M* to accept the complement of *L* as follows:

1. Make each accepting state of *M* a nonaccepting state from which there are no moves. Thus, if *M* accepts, the new TM will halt without accepting.
2. Create a new state *q*, which is the only accepting state in the new TM. For each state-symbol combination that has no move, hte new TM enters state *q*, whereupon it accepts and halts.

Solutions for Section 10.2

Exercise 10.2.1(a)
Choosing *x = y = z = 1* makes the expression satisfiable. Thus, the expression is in SAT.

Exercise 10.2.2(a)
There are actually only three distinct Hamilton circuits, once we account for the differences in direction and differences in starting point. These three circuits are (1,2,3,4), (1,3,2,4), and (1,3,4,2). We can express the existence of one of these three circuits (using the simplified notation of Section 10.3) by: *x12x23x34x14 + x13x23x24x14 + x13x34x24x12*.

Solutions for Section 10.3

Exercise 10.3.1(a)
In what follows, *[-x]* stands for x-bar, the complement of *x*. We'll begin by using the construction to put it into CNF. *xy* is already the product of clauses *(x)(y)*, and *[-x]z* is the product of clauses *([-x])(z)*. When we use the OR construction to combine these, we get *(x+u)(y+u)([-x]+[-u])(z+[-u])*.

Now, to put this expression into 3-CNF, we have only to expand the four clauses, each of which has only two literals, by introducing four new variables and doubling the number of clauses. The result: *(x+u+v1)(x+u+[-v1])(y+u+v2)(y+u+[-v2])([-x]+[-u]+v3)([-x]+[-u]+[-v3])(z+[-u]+v4)(z+[-u]+[-v4])*.

Exercise 10.3.3(a)
It is satisfiable. Let any two variables be assigned TRUE, say *x1* and *x2*, and let the other two variables be assigned FALSE. Then in any set of three variables, there must be at least one true and at least one false. Thus, none of the clauses can be false.

Solutions for Section 10.4

Exercise 10.4.1

For part (a): There are triangles (3-cliques), such as {1,2,3}. However, there is no 4-clique, since there are only 4 nodes, and one edge is missing. Thus *k = 3* is the answer.

For part (b): All pairs of nodes must have an edge between them, and the number of pairs of *k* nodes is *k choose 2*, or *k(k-1)/2*.

For part (c): We reduce NC to CLIQUE as follows. Suppose we are given an instance *(G,k)* of NC. Construct the instance *(G',n-k)* of CLIQUE, where *n* is the total number of nodes of *G*, and *G'* is *G* with the set of edges complemented; that is, *G'* has edge *(u,v)* if and only if *G* does not have that edge.

We must show that *G* has a node cover of size *k* if and only if *G'* has a clique of size *n-k*. First, let *C* be a node cover of *G* of size *k*. We claim that *C'*, the complement of the nodes in *C*, is a clique in *G'* of size *n-k*. Surely *C'* is of size *n-k*. Suppose it is not a clique. Then there is a pair of nodes *(u,v)* that do not have an edge in *G'*. Thus this edge is in *G*. But neither *u* nor *v* is in *C*, contradicting the assumption that is is a node cover.

Conversely, if *C'* is a clique of size *n-k* in *G'*, then we claim that *C* the complement of *C'*, is a node cover of size *k* in *G*. The argument is similar: if *(u,v)* is an edge of *G* not covered by *C*, then both *u* and *v* are in *C'*, but the edge *(u,v)* is not in *G'*, contradicting the assumption that *C'* is a clique.

Exercise 10.4.2

For each clause, we add one node, and connect it so that it can only be colored in one of the *n+1* available colors if the clause is made true. Suppose the clause consists of literals with variables *xi*, *xj*, and *xk*, possibly negated. The node for the clause is connected to:

1. *xm* for all *m* = 0, 1,..., *n*, except for *i, j*, and *k*. Thus, the only possible colors for the nodes are the ones used for its literals.
2. If the literal with *xi* is positive (not negated), connect the node for the clause to the node for *xi*. If the literal is negated, connect the node for the clause to the node for *xi*-bar.
3. Connect to nodes for *xj* and *xk*, analogously.

Now, if at least one of the literals of the clause is made true by the assignment where the color *c0* corresponds to truth, then that literal will not be colored with the color for its variable, and we can use that color for the clause's node. However, if the truth assignment makes all three literals false, then the clause's node is connected to nodes of all *n+1* colors, and we cannot complete the coloring. Thus, coloring the complete graph with *n+1* colors is possible if and only if there is a satisfying truth assignment for the 3-CNF expression.

Exercise 10.4.3(a)

Yes; a Hamilton circuit can be found by going around the inner circle, say from 11 to 20, clockwise, then to 10, around the outer circle counterclockwise, to 1, and then back to 11.

Exercise 10.4.4(f)

Let *(G,k)* be an instance of the clique problem, and suppose *G* has *n* nodes. We produce an instance of the half-clique problem, as follows:

1. If $k = n/2$, just produce G. Note that G has a half-clique if and only if it has a clique of size k. in this case.
2. If $k > n/2$, add $2k - n$ isolated nodes (nodes with no incident edges). The resulting graph has a half-clique (whose size must be $(n + (2k-n))/2 = 2k$, if and only if G has a clique of size k.
3. If $k < n/2$, add $n - 2k$ nodes, and connect them in all possible ways to each other and to the original nodes of G. The new graph thus has $2(n-k)$ nodes. The new nodes, plus a clique of size k in G form a clique of size $(n-2k) + k = n-k$, which is half the number of nodes in the new graph. Conversely, if the new graph has a half-clique, then it must include at least $(n-k) - (n-2k) = k$ nodes of the graph G, implying that G has a clique of size k.

These steps complete a reduction of CLIQUE to HALF-CLIQUE. It is evidently performable in polynomial time, since the number of new nodes and edges is at most the square of the original number of nodes, and the rules for adding nodes and edges are simple to carry out.

Exercise 10.4.5(a)
Following the hint, pick any node x in graph G. Add a duplicate node y that is adjacent to exactly those nodes to which x is adjacent. Then, add new nodes u and v that are adjacent to x and y, respectively, and no other nodes. Call the resulting graph G'.

We claim G' has a Hamilton path if and only if G has a Hamilton circuit. If G has a Hamilton circuit, the following is a Hamilton path in G': start at u, go to x, follow the Hamilton circuit, but end at y instead of x, and then go to v.

If G' has a Hamilton path, it must start at u and end at v, or vice-versa (which is really the same path. Moreover, the path must go from xy, visiting all the nodes of G as it does. Thus, if we replace y by x along this path, we get a Hamilton circuit in G.

Exercise 10.4.5(c)
A spanning tree with two leaf nodes is a Hamilton path. Thus, the Hamilton path problem reduces to the question of whether a graph has a spanning tree with only 2 leaf nodes. Surely, then, Hamilton path reduces to the more general problem stated in the question, where the number of leaf nodes is a parameter of the problem.

Solutions for Chapter 11

Solutions for Section 11.1

Exercise 11.1.1(a)
The problem is in **NP**. We need only to test whether the expression is true when all variables are true (a polynomial-time, deterministic step) and then guess and check some other assignment. Notice that if an expression is not true when all variables are true, then it is surely not in TRUE-SAT.

The complement of TRUE-SAT consists of all inputs that are not well-formed expressions, inputs that are well-formed expressions but that are false when all variables are true, and well-formed expressions that are true only when all variables are true. We shall show TRUE-SAT is NP-complete, so it is unlikely that the complement is in **NP**.

To show TRUE-SAT is NP-complete, we reduce SAT to it. Suppose we are given an expression *E* with variables *x1, x2,..., xn*. Convert *E* to *E'* as follows:

1. First, test if *E* is true when all variables are true. If so, we know *E* is satisfiable, and so convert it to a specific expression *x+y* that we know is in TRUE-SAT.
2. Otherwise, let *E' = E + x1x2...xn*, surely a polynomial-time reduction. Surely *E'* is true when all variables are true. If *E* is in SAT, then it is satisfied by some truth assignment other all all-true, because we tested all-true and found *E* to be false. Thus, *E'* is in TRUE-SAT. Conversely, if *E'* is in TRUE-SAT, then since *x1x2...xn* is true only for the all-true assignment, *E* must be satisfiable.

Exercise 11.1.2
There are three things to show. The language is in **NP**, in co-**NP**, and not in **P**.

1. To show the language is in **NP**, guess *z*, compute *f(z)* deterministically in polynomial time, and test whether *f(z) = x*. When the guess of *z* is correct, we have $f^{-1}(x)$. Compare it with *y*, and accept the pair *(x,y)* if *z < y*.
2. To show the language to be in co-**NP**, we have to show the complement --- the set of inputs that are not of the form *(x,y)*, where $f^{-1}(x) < y$, is in **NP**. It is easy to check for ill-formed inputs, so the hard part is checking whether $f^{-1}(x) >= y$. However, the trick from part (1) works. Guess *z*, compute *f(z)*, test if *f(z) = x*, and then test if *z >= y*. If both tests are met, then we have established that $f^{-1}(x) >= y$, so *(x,y)* is in the complement language.
3. Finally, we must show that the language is not in **P**. We can show that if it were in **P**, then with *n* tests for membership in the language, we could binary-search to find the exact value of $f^{-1}(x)$. If one test takes time that is polynomial in *n*, then *n* times that amount is also polynomial in *n*. Start by testing the pair $(x,2^{n-1})$, i.e., the rough midpoint in the range of *n*-bit integers. If the answer is ``yes,'' next test $(x,2^{n-2})$; if the answer is ``no,'' test $(x,3*2^{n-2})$ next. In this manner, we can establish one bit of $f^{-1}(x)$ at each test, and after *n* tests, we know $f^{-1}(x)$ exactly.

Solutions for Section 11.3

Exercise 11.3.2

4. Suppose *M* is a TM with polynomial space bound *p(n)*, and *w* is an input to *M* of length *n*. We must show how to take *M* and *w*, and write down, in polynomial time, a regular expression *E* that is Sigma* if and only if *M* does not accept *w*.

   Technically, this construction reduces *L(M)* to the complement of the set in question, that is, to the set of regular expressions that are not equivalent to Sigma*. However, an easy consequence of Theorem 11.4 is that, since a deterministic, polynomial-time TM can be made to halt, **PS** is closed under complementation; just change the accepting states to halting, but nonaccepting states, add an accepting state, and make every halting, nonaccepting state transfer to that accepting state instead of halting immediately. Thus, we could assume that *M* is actually a TM for the complement of the language *L* in **PS** in question. Then, we are actually reducing *L* to the language of regular expressions equivalent to Sigma*, as requested.

To construct regular expression *E*, we shall write *E = F + G + H*, where the three subexpressions *E, F*, and *H* define sequences of ID's of *M* that do not ``start right,'' ``move right,'' and ``finish right,'' respectively. Think of an accepting computation of *M* as a sequence of symbols that are the concatenation of ID's of *M*, each preceeded by a special marker symbol #. The alphabet Sigma for *E* will be # plus all the tape and state symbols of *M*, which we can assume without loss of generality are disjoint. Each ID is exactly *p(n)+1* symbols long, since it includes the state and exactly *p(n)* tape symbols, even if many at the end are blank.

1. *H*: Finishes wrong. *M* fails to accept if the sequence has no accepting ID. Thus, let *H = (Sigma - Qf)**, where *Qf* is the set of accepting states of *M*.
2. *F*: Starts wrong. Any string in which the first *p(n)+2* symbols are not #, *q_0* (the start state), *w*, and *p(n) - n* blanks, is not the beginning of an accepting computation, and so should be in *L(E)*. We can write *F* as the sum of the terms:
   - *(Sigma-{#})Sigma**, i.e., all strings that do not begin with #.
   - *Sigma(Sigma-{q_0})Sigma**, i.e., all strings that do not have *q_0* as their second symbol.
   - $Sigma^{i+1}$*(Sigma-{a_i})Sigma**, where *a_i* is the *i*th position of *w*. Note $Sigma^{k}$ stands for Sigma written *k* times, but this expression takes only polynomial time to write.
   - $Sigma^{i}$*(Sigma-{B})Sigma**, for all *n+3 <= i <= p(n)+1*. Here, *B* represents the blank, and this expression covers all strings that fail to have a blank where the first ID must have one. Note that these terms require us to write Sigma as many as *p(n)+1* times, but that still takes only polynomial time. Also, there are polynomially many terms, so the total work is polynomial.
   - $(Sigma + \varepsilon)^{p(n)+1}$. This term covers all strings that are shorter than *p(n)+2* symbols, and therefore cannot have an initial ID, regardless of the symbols found there. As in the previous set of terms, the time taken to write this term is large, but polynomial.
3. *G*: moves wrong. We need to capture all strings that have some point at which symbols separated by distance roughly *p(n)* do not reflect a move of *M*. The idea is similar to that used in Cook's theorem (Theorem 10.9). Each position of an ID is determined by the symbol at that position in the previous ID and the two neighboring positions. Thus, *G* is the sum of terms *(Sigma*)UVW*($Sigma^{p(n)}$)*X(Sigma*)*, where *U, V, W, X* are four symbols of Sigma such that if *UVW* were three consecutive symbols of an ID of *M* (*U* may be # if the ID is just beginning, and *W* may be # if the ID is ending), then *X* would **not** be the symbol in the same position as *V* in the next ID. For example, if none of *U, V, W* are a state, then *X* could be any symbol but *V*. Again, we can write this large expression in polynomial time, even though it requires us to write Sigma *p(n)* times.

If *M* accepts *w*, then there is some accepting computation, and the string representing that computation fails to match any of the regular expressions described above. Thus, *E != Sigma**. However, any string that is not an accepting computation of *M* on *w* will surely fail meet one of the conditions ``starts wrong," ``moves wrong," or ``finished wrong," and therefore will be in *L(E)*. Thus, if *M* does not accept *w*, then *E = Sigma**.

Solutions for Section 11.5

Exercise 11.5.1(a)

5. A simple way to to observe that 9 - 11 = -2, and -2 modulo 13 is 11, since 13 -2 = 11. Or, we may treat the subtraction as addition of a negative number, say that -11 modulo 13 is 2, and 9 + 2 modulo 13 is 11.

Exercise 11.5.1(c)

6. We need to compute the inverse of 8 modulo 13. Exploring the possibilities, from 2 to 12, we find that 8*5 = 40, which is 1 modulo 13. Thus, 1/8 = 5 modulo 13, and 5/8 = 5*5 = 25 = 12 modulo 13. Thus, 5/8 = 12.

Exercise 11.5.3(a)

7. From the table of Fig. 11.9, we see that 2*2 = 4, 3*3 = 2, 4*4 = 2, 5*5 = 4, and 6*6 = 1. We also know that 1*1 = 1. Thus, 1, 2, and 4 are the quadratic residues modulo 7.