Answers to Practice Problems

Chapter 2

Section 2.2.2

1.

Step	*Operation*
1	Get values for *x*, *y*, and *z*
2	Set the value of *average* to $(x + y + z)/3$
3	Print the value of *average*
4	Stop

2.

Step	*Operation*
1	Get a value for *r*, the radius of the circle
2	Set the value of *circumference* to $2 \times \pi \times r$
3	Set the value of *area* to $\pi \times r^2$
4	Print the values of *circumference* and *area*
5	Stop

3.

Step	*Operation*
1	Get values for *amount*, the amount of electricity used, and for *cost*, the cost per kilowatt-hour
2	Set the value of *subtotal* to *amount* × *cost*
3	Set the value of *tax* to 0.08 × *subtotal*
4	Set the value of *total* to *subtotal* + *tax*
5	Print the value of *total*
6	Stop

4.

Step	*Operation*
1	Get values for *balance*, the current credit card balance, for *purchases*, the total dollar amount of new purchases, and for *payment*, the total dollar amount of all payments
2	Set the value of *unpaid* to *balance* + *purchases* − *payment*
3	Set the value of *interest* to *unpaid* × 0.12
4	Set the value of *newbalance* to *unpaid* + *interest*
5	Print the value of *newbalance*
6	Stop

5.

Step	*Operation*
1	Get values for *length* and *width* in feet
2	Set the value of *length-in-yards* to *length*/3

3	Set the value of *width-in-yards* to *width*/3
4	Set the value of *area* to *length-in-yards* × *width-in-yards*
5	Set the value of *cost* to *area* × 23
6	Print the value of *cost*
7	Stop

6.

Step	***Operation***
1	Get values for *first*, *second*, and *third*
2	Set the value of *points earned* to (5 × *first*) + (3 × *second*) + (1 × *third*)
3	Print the value of *points earned*
4	Stop

Section 2.2.3

1. If $x \geq 0$ then

    Set the value of *y* to 1

Else

    Set the value of *y* to 2

2. Get values for *x*, *y*, and *z*

If $x > 0$ then

    Set the value of *average* to (*x* + *y* + *z*)/3

    Print the value of *average*

Else

    Print the message 'Bad Data'

Stop

3. Get values for *balance*, *purchases*, *payment*

Set the value of *unpaid* to *balance* + *purchases* − *payment*

If *unpaid* < 100 then

    Set the value of *interest* to *unpaid* × 0.08

Else

    If *unpaid* ≤ 500 then

        Set the value of *interest* to *unpaid* × 0.12

    Else

        Set the value of *interest* to *unpaid* × 0.16

Set the value of *newbalance* to *unpaid* + *interest*

Print the value of *newbalance*

Stop

4. Get a value for *x*

While $x \neq 999$ do

Set the value of *a* to x^2

Set the value of *b* to sin(*x*)

Set the value of *c* to 1/*x*

Print the values of *a*, *b*, and *c*

Get a value for *x*

End of the loop

Stop

5. Get values for *length*, *width*, *price*

Set the value of *length-in-yards* to *length*/3

Set the value of *width-in-yards* to *width*/3

Set the value of *area* to *length-in-yards* × *width-in-yards*

Set the value of *cost* to *area* × *price*

If *cost* ≤ 500 then

Print the message 'You can afford this carpet'

Else

Print the message 'This carpet is too expensive'

Stop

6. Add the following statement to the end of the solution, just before the Stop statement:

(Assume that we have stored the cost of the carpet in the variable *cost*.)

If (*cost* ≤ 250) then

Print the message 'This is a particularly good deal.'

7. Get a value for *x*

While ($x \neq 999$) do

Set the value of *a* to x^2

Print the value of *a*

Set the value of *b* to sin(*x*)

Print the value of *b*

If $x = 0$ then

Print the error message 'Cannot compute 1/x'

Else

Set the value of *c* to 1/*x*

Print the value of *c*

Get a value for *x*

End of the loop

Stop

Section 2.3.1

1. Initial values

$a = 2$ $b = 4$ *count* $= 0$ *product* $= 0$

After pass 1

$a = 2$ $b = 4$ *count* $= 1$ *product* $= 2$

After pass 2

$a = 2$ $b = 4$ *count* $= 2$ *product* $= 4$

After pass 3

$a = 2$ $b = 4$ *count* $= 3$ *product* $= 6$

After pass 4

$a = 2$ $b = 4$ *count* $= 4$ *product* $= 8$

2. Yes, the algorithm still works correctly. On Line 1, we input the two values $a = 0$, $b = 0$. On Line 2, the Boolean expression is true, so we set *product* $= 0$ and skip the entire else clause. The next line executed is "print the value of product" which outputs a 0, the correct answer to the problem 0×0.

3. case 1 ($a = -2$, $b = 4$):
The value of *product* will be -8, which is correct.

case 2 ($a = 2$, $b = -4$):
The value of *product* will be 0 (the while loop does not execute at all), which is incorrect.

4. The original algorithm fails when $b < 0$ because *count* is never less than *b*. If $b < 0$, change the value of *b* to $-b$, but set the value of a variable called *bnegative* to YES to remember that *b* was negative. After the product is computed, the sign of *product* will be incorrect if *b* was negative, so change the sign. Here is a pseudocode version that works for all integer values of *a* and *b*:

Get values for *a* and *b*

Set the value of *bnegative* to NO

If (either *a* = 0 or *b* = 0) then

  Set the value of *product to* 0

Else

If $b < 0$ then
  Set the value of b to $-b$
  Set the value of *bnegative* to YES
Set the value of *count* to 0
Set the value of *product* to 0
While (*count* $<$ b) do
  Set the value of *product* to (*product* + a)
  Set the value of *count* to (*count* + 1)
End of the loop
If (*bnegative* = YES) then
  Print the value of $-$*product*
Else
  Print the value of *product*
Stop

5. It is a highly inefficient algorithm because of the number of steps required to find the answer. For example, to add 234 + 567, we will have to repeat the addition of the value 234 to a running sum 567 separate times. When we do multiplication the "traditional" way that we learned in grade school, we have to carry out far fewer operations. (We will learn much more about how to evaluate the efficiency of algorithms in Chapter 3.)

6. Get values for a and b
If (either $a > 10{,}000$ or $b > 10{,}000$) then
  Print 'Please use a more efficient multiplication algorithm'
Else
  If (either $a = 0$ or $b = 0$) then
    Set the value of *product* to 0
  Else
    Set the value of *count* to 0
    Set the value of *product* to 0
    While (*count* $<$ b) do
      Set the value of *product* to (*product* + a)
      Set the value of *count* to (*count* + 1)
    End of the loop
  Print the value of *product*
Stop

Section 2.3.3

1. We would need to change Line 1 so that the algorithm would input 1 million numbers and names rather than 10,000. We would need to change Line 3 so that the loop would repeat a maximum of 1 million times rather than a maximum of 10,000. Actually, that is not a lot of change given that the problem is 100 times bigger. We only needed to change two lines. In fact, if the phone book were 1 billion numbers in length, then we would only need to change the same two lines.

2. ***Step*** ***Operation***

```
1     Get values for NUMBER, T1, … , T10,000 and N1, … , N10,000
2     Set the value of i to 1 and the value of Found to NO
3     Do
4           If NUMBER is equal to the ith number on the list, Ti , then
5                 Print the name of the corresponding person, Ni
6                 Set the value of Found to YES
            Else
7                 Add 1 to the value of i
8     While (Found = NO) and (i ≤ 10,000)
9     If (Found = NO) then
10          Print the message 'Sorry, this number is not in the directory'
11    Stop
```

3. You must change the operation on Line 7 from a greater-than (>) to a less-than (<) sign. That line will now read as follows:

   If $A_i <$ *largest so far then* ...

   That is the only required change. However, to avoid confusion about what the algorithm is doing, you probably should also change the name of the variable *largest so far* to something like *smallest so far* on Lines 3, 7, 8, and 12. Otherwise, a casual reading of the algorithm might lead someone to think incorrectly that it is still an algorithm to find the largest value rather than the smallest.

4. If $n = 0$ (the list is empty), then there are no values for $A_1, A_2, \ldots, A_n$. In particular, setting the value of *largest so far* to A_1 gives a meaningless value to *largest so far*. The while loop will not execute at all because i has the value 2 and n has the value 0, so the condition $i \leq n$ is false. The algorithm will print out nonsense values for *largest so far* and *location*.

   The algorithm can be fixed by putting a conditional statement after Line 1. If $n = 0$, then the algorithm should print a message that says the list is empty. The "else" case will be the rest of the current algorithm.

5. If $n = 1$, then *largest so far* is set to A_1, the only list element, and *location* is set to 1. Also i is set to 2, so the while loop will not execute because it is false that $i \leq n$ (i.e., it is false that $2 \leq 1$). The correct values of *largest so far* and *location* are printed.

6. Yes, it would still work correctly in one sense—it would find the largest numerical value in the list. However, if two numbers were equal, this change would cause the algorithm to find the last occurrence of that number rather than the first. So the value of *largest so far* would not change but the value of *location* might be different.

Section 2.3.4

1. a. NUMBER = (771) 921-5281

i	*Operation*	*Found*
1	Compare (771) 921-5281 to T_1, (648) 555-1285. No match.	No
2	Compare (771) 921-5281 to T_2, (247) 834-6543. No match.	No
3	Compare (771) 921-5281 to T_3, (771) 921-5281. Match.	Yes

Output = Adams

b. NUMBER = (488) 351-1673

i	*Operation*	*Found*
1	Compare (488) 351-1673 to T_1, (648) 555-1285. No match.	No
2	Compare (488) 351-1673 to T_2, (247) 834-6543. No match.	No
3	Compare (488) 351-1673 to T_3, (771) 921-5281. No match.	No
4	Compare (488) 351-1673 to T_4, (356) 327-8900. No match.	No

Output = 'Sorry, this number is not in the directory'

2. $n = 7$, $A_1 = 22$, $A_2 = 18$, $A_3 = 23$, $A_4 = 17$, $A_5 = 25$, $A_6 = 30$, $A_7 = 2$

Largest So Far	*Location*	*i*	*Operation*
22	1	2	Compare A_2 and *largest so far*. Is 18 > 22? No
22	1	3	Compare A_3 and *largest so far*. Is 23 > 22? Yes, so reset values
23	3	4	Compare A_4 and *largest so far*. Is 17 > 23? No
23	3	5	Compare A_5 and *largest so far*. Is 25 > 23? Yes, so reset values
25	5	6	Compare A_6 and *largest so far*. Is 30 > 25? Yes, so reset values
30	6	7	Compare A_7 and *largest so far*. Is 2 > 30? No

Output: Largest = 30. Location = 6.

3. Pattern = an $m = 2$. The pattern has 2 characters.
Text = A man and a woman $n = 17$. The text has 17 characters.

k	*i*	*Mismatch*	*Operation*
1	1	No	Compare P_1, the "a", to T_1, the "A". No match.
		Yes	End of the check for a match at position 1 of the text.
2	1	No	Compare P_1, the "a", to T_2, the blank. No match.
		Yes	End of the check for a match at position 2 of the text.
3	1	No	Compare P_1, the "a", to T_3, the "m". No match.
		Yes	End of the check for a match at position 3 of the text.
4	1	No	Compare P_1, the "a", to T_4, the "a". Match.
4	2	No	Compare P_2, the "n", to T_5, the "n". Match.
4	3	No	i (3) is greater than m (2), so we exit the loop.

Output: There is a match at position 4

In a similar way, the program will produce the following two additional lines of output:
There is a match at position 7
There is a match at position 16

4. If $m > n$, then $n - m + 1 \leq 0$. Because the value of k is set to 1 right before the outer while loop, the condition $k \leq (n - m + 1)$ is false, the loop is not executed, and the algorithm terminates with no output.

5. If we had incorrectly initialized i to 0 rather than 1, then the algorithm would halt with a fatal error on Line 8. In that line, we reference the variable P_i. If i is 0, this would be P_0, but there is no such value, as the pattern begins with the element P_1. This shows how important it is to be very careful about not only the "big issues" but the little details as well. Even a very tiny initialization error can cause an entire algorithm to fail.

6. Yes, it will work correctly. The quantity $(n - m + 1)$ evaluates to 1. Since k is initialized to 1, the test $k \leq (n - m + 1)$ will initially evaluate to True and the loop will be executed. The text and the pattern will be matched beginning at position 1. Then when k is incremented to 2, the test will become false and we will exit the loop, as desired.

Chapter 3

Section 3.2

The numbers from 1 to n, where n is even, can be grouped into $n/2$ pairs of the form

$$1 + n = n + 1$$

$$2 + (n - 1) = n + 1$$

$$\ldots$$

$$n/2 + (n/2 + 1) = n + 1$$

giving a sum of $(n/2)(n + 1)$. This formula gives the correct sum for all cases shown, whether n is even or odd.

Section 3.3.2

N	*Best Case*	*Worst Case*	*Average Case*
10	1	10	5
50	1	50	25
100	1	100	50
1,000	1	1,000	500
10,000	1	10,000	5,000
100,000	1	100,000	50,000

Section 3.3.3

1. a. 4, 8, 2, 6
4, 6, 2, 8
4, 2, 6, 8
2, 4, 6, 8

b. 12, 3, 6, 8, 2, 5, 7
7, 3, 6, 8, 2, 5, 12
7, 3, 6, 5, 2, 8, 12
2, 3, 6, 5, 7, 8, 12
2, 3, 5, 6, 7, 8, 12

c. D, B, G, F, A, C, E, H
D, B, E, F, A, C, G, H
D, B, E, C, A, F, G, H
D, B, A, C, E, F, G, H
C, B, A, D, E, F, G, H
A, B, C, D, E, F, G, H

d. The list is already in sorted order. Although the selection sort algorithm will go through all of its operations, the largest element will always be at the back of the unsorted section and will be exchanged with itself, resulting in no change.

2. This question can be answered using the formulas from Section 3.3.3:

Number of comparisons $= \frac{1}{2}n^2 - \frac{1}{2}n$ Number of exchanges $= n$

a. Comparisons: 6 Exchanges: 4
b. Comparisons: 21 Exchanges: 7
c. Comparisons: 28 Exchanges: 8
d. Comparisons: 10 Exchanges: 5

Section 3.3.4

The basic shape of the curve as n gets large is still n^2 because as n gets large, the n^2 term dominates the other two terms.

Section 3.4.1

1. *legit* = 3

2	4	1	1

2.

2	0	4	1

2	4	1

3. *legit* = 3

2	1	4	1

4. For example,

1	2	0	0

Section 3.4.2

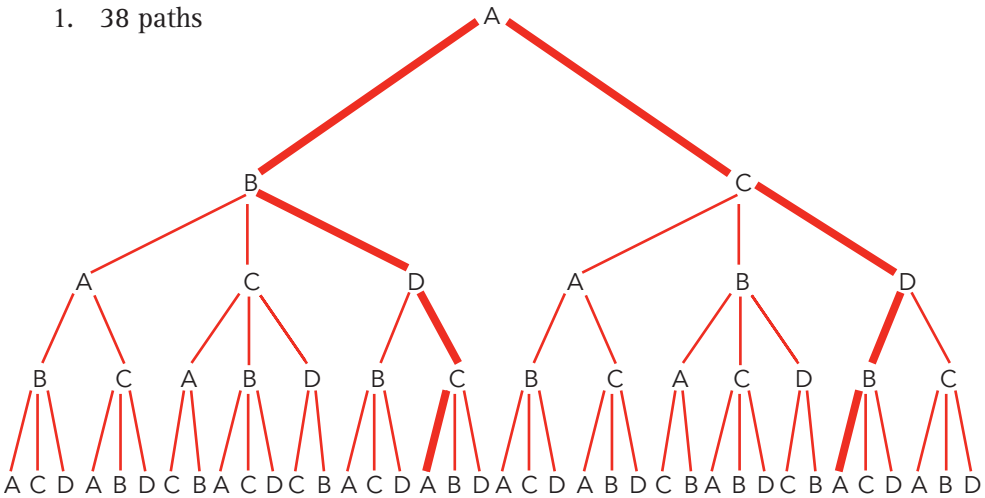
1. 5656170224, 7719215281, 6485551285
2. 5656170224, 7719215281, 8796562127
3. If the data file is sorted numerically by SSN, then the binary search algorithm can be used. The worst case is 29 (2^{28} = 268,435,456 and 2^{29} = 536,870,912). If the data file is sorted alphabetically by the individual's last name, then as far as searching for a SSN is concerned, the file is unsorted. The sequential search algorithm must be used, with a worst case of 453,700,000.

Section 3.4.3

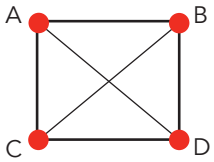
Pattern = AAAB; Text = AAAAAAAAA; $m = 4$; $n = 9$; $m \times n = 36$; the exact number of comparisons is $4 \times 6 = 24$.

Section 3.5

1. 38 paths



2.



3. Algorithms of $\Theta(n^2)$ are polynomial algorithms because n is raised to a constant power. An algorithm of $\Theta(2^n)$ is an exponential algorithm. It is the nonconstant exponent that makes this value grow so quickly, as opposed to polynomial algorithms. An algorithm of $\Theta(n^n)$ would still be considered an exponential algorithm because its exponent is n. In fact, such an algorithm grows even faster than one of order 2^n because of the nonconstant base.

Chapter 4

Section 4.2.1

1. a. $10101000 = (1 \times 2^3) + (1 \times 2^5) + (1 \times 2^7)$
$= 8 + 32 + 128$
$= 168$ as an unsigned integer value

   b. $10101000 = (1 \times 2^3) + (1 \times 2^5)$
$= 8 + 32$
$= 40$

   This is the value of the magnitude portion of the number. The leftmost bit represents the sign bit. In this example, it is a 1, which is a negative sign.

   $= -40$ as a signed integer value

2. To answer this question, you must represent the decimal value as a sum of powers of 2 and then convert that representation to binary.

   $99 = 64 + 32 + 2 + 1$
$= 2^6 + 2^5 + 2^1 + 2^0$
$= 1100011$

   However, this is only 7 bits and we need 8, so we must add one leading 0 to fill out the answer.

   $= 01100011$

3. The 10 bits would be represented as 9 bits for the magnitude and the leftmost bit for the sign. To represent the magnitude, we must rewrite 300 as the sum of powers of 2, as we did in the previous question.

   $300 = 256 + 32 + 8 + 4$
$= 2^8 + 2^5 + 2^3 + 2^2$
$= 100101100$ in 9 bits

To make it a negative value, we must add a 1 bit (the negative sign) to the leftmost position of the number.

$-300 = 1100101100$
$254 = 128 + 64 + 32 + 16 + 8 + 4 + 2$
$= 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1$
$= 011111110$ to 9 bits of accuracy for the magnitude

To make it a $+254$, we must add a 0 (the $+$ sign) to the leftmost position of the number.

$+254 = 0011111110$

4. Because the leftmost digit is a 1, the number is negative. The magnitude is represented by the remaining 7 digits, 0010111, which is $1 + 2 + 4 + 16 = 23$. Thus, the number is -23.

5. Writing the numbers 0000 through 1111 around a circle, the positive values count up from 0000 and the negative numbers count down from 1111.
   a. $+6$ is 0110
   b. -3 is 1101

6.
```
  1110      ← carry digit
  01110
+01011
 11001
```

7. a. To see what this value would look like in ASCII, we first look up the characters "X", "+", and "Y" in the ASCII conversion table to see what their internal representation is in decimal.

   "X" = 88
   "+" = 43
   "Y" = 89

   We then convert these decimal values to unsigned 8-bit binary values.

   "X" = 88 = 01011000
   "+" = 43 = 00101011
   "Y" = 89 = 01011001

   The internal representation of the three-character string 'X+Y' is formed by putting together all three of the preceding values, producing the following 24-bit string:

   010110000010101101011001

   which is how a computer stores 'X + Y' using ASCII encoding.

   b. From the Unicode tables,

   "X" = 0058
   "+" = 002B
   "Y" = 0059

where these representations are in hexidecimal (base 16) form. In base 16, digits run from 0–F rather than 0–9 as in the decimal system. Translating these representations into decimal, we get

$$\begin{aligned}\text{“X”} &= 0 \times 16^3 + 0 \times 16^2 + 5 \times 16^1 + 8 \times 16^0 \\ &= 80 + 8 \\ &= 88\end{aligned}$$

$$\begin{aligned}\text{“+”} &= 0 \times 16^3 + 0 \times 16^2 + 2 \times 16^1 + 11 \times 16^0 \\ &= 32 + 11 \\ &= 43\end{aligned}$$

$$\begin{aligned}\text{“Y”} &= 0 \times 16^3 + 0 \times 16^2 + 5 \times 16^1 + 9 \times 16^0 \\ &= 80 + 9 \\ &= 89\end{aligned}$$

These are the same decimal values as under ASCII encoding (Unicode for common characters agrees with ASCII encoding) but will be written in 16-bit binary form, with extra spaces for readability.

“X” = 88 = 0000 0000 0101 1000
“+” = 43 = 0000 0000 0010 1011
“Y” = 89 = 0000 0000 0101 1001

Putting these together produces the following 48-bit string for ‘X+ Y’:

0000 0000 0101 1000 0000 0000 0010 1011 0000 0000 0101 1001

8. a. $+0.25 = 0.01$ in binary
$= 0.1 \times 2^{-1}$ in scientific notation

so the mantissa is $+0.1$ and the exponent is -1.

$= 0\ \underbrace{100000000}_{\text{mantissa}}\quad\underbrace{1\ 00001}_{\text{exponent}}$

b. $-32\ 1/16 = -100000.0001$
$= -0.1000000001 \times 2^6$
$= 1\ \underbrace{100000000}_{\text{mantissa}}\quad\underbrace{0\ 00110}_{\text{exponent}}$

Note that the last 1 in the mantissa was not stored because there was not enough room. The loss of accuracy that results from limiting the number of digits available is called a *truncation error*.

9.
$$\begin{array}{rl} 1111 & \leftarrow \text{carry digits} \\ 00001 & \\ +01111 & \\ \hline 10000 & \end{array}$$

Here $15 + 1 = -16$

10. You are adding the two values $+7$ and $+14$, which should produce the value $+21$. But when you add the two values together, you get 10101, which is a negative value because the leftmost digit is a 1. In fact, it is the value -5. What happened is that we got an *arithmetic overflow*. The value $+21$ is too big to represent in sign/magnitude with only five digits.

Section 4.2.2

1. 44,100 samples/second × 16 bits/sample × 3 minutes × 60 sec/minute = 127 million bits

   Compressed at a ratio of 4:1, this becomes about 32 million bits.

2. 66,000 samples/sec × 24 bits/sample × 180 seconds = 285,120,000 bits

3. 2,100,000 pixels × 24 bits/pixel = 50,400,000 bits or 6,300,000 bytes

4. To reduce 6,300,000 bytes to 1,000,000 bytes requires a compression ratio of almost 7:1.

   To reduce 6,300,000 bytes to 256,000 bytes requires a compression ratio of almost 25:1.

5. Compression ratio = original size / compressed size = 9.6 / 3 = 3.2

6. The fixed-length 4-bit representation of ALOHA requires 5 × 4 = 20 bits. Using the variable-length code requires the following:

A	L	O	H	A
00	1111111	0111	010	00

   which is a total of 18 bits. The compression ratio is 20 / 18 = 1.11.

7. The number of bits required per song is:

   44,100 samples/sec × 16 bits/sample × 180 sec/song × 0.1 (compression) = 12,700,800 bits/song = 1,587,600 bytes/song (at 8 bits/byte)

   If there are 16 GB available on your smartphone for music storage, then the total number of songs that can be stored is approximately:

   $16 \times 10^9 / 1.587 \times 10^6 = 10{,}080$ songs

Section 4.3.1

1. a. $(x = 1)$ AND $(y = 3)$
   True AND False
   False — The final answer is False.

   b. $(x < y)$ OR $(x > 1)$
   True OR False
   True — The final answer is True.

   c. NOT $[(x = 1)$ AND $(y = 2)]$
   NOT [True AND True]
   NOT [True]
   False — The final answer is False.

2.

A	*B*	*C*	*((NOT A) AND B) OR C*
F	F	F	F
F	F	T	T
F	T	F	T
F	T	T	T
T	F	F	F
T	F	T	T
T	T	F	F
T	T	T	T

3. $(x = 5)$ AND $(y = 11)$ OR $([x + y] = z)$
True AND False OR True

   We now must make an assumption about which of the two logical operations to do first, the AND or the OR. If we assume the AND goes first, then we get

   False OR True
   True

   If we assume that the OR goes first, then the expression would be evaluated as follows:

   True AND True
   True

   In this case, the answer is the same, but we arrive at the answer in different ways.

4. $(x \geq 0)$ AND $(x \leq 100)$ AND $(y \geq 0)$ AND $(y \leq 100)$ AND (NOT$(x = y)$)

5. NOT[$(score \geq 200)$ AND $(score \leq 800)$]

6. The expression (*A* OR *B*) OR (NOT *A*) is true for all values of *A* and *B* and, therefore, is never false. A Boolean expression that is always true is called a *tautology*.

7. $(year = 4)$ AND (($major$ = Chemistry) OR ($major$ = Physics))

Section 4.4.2

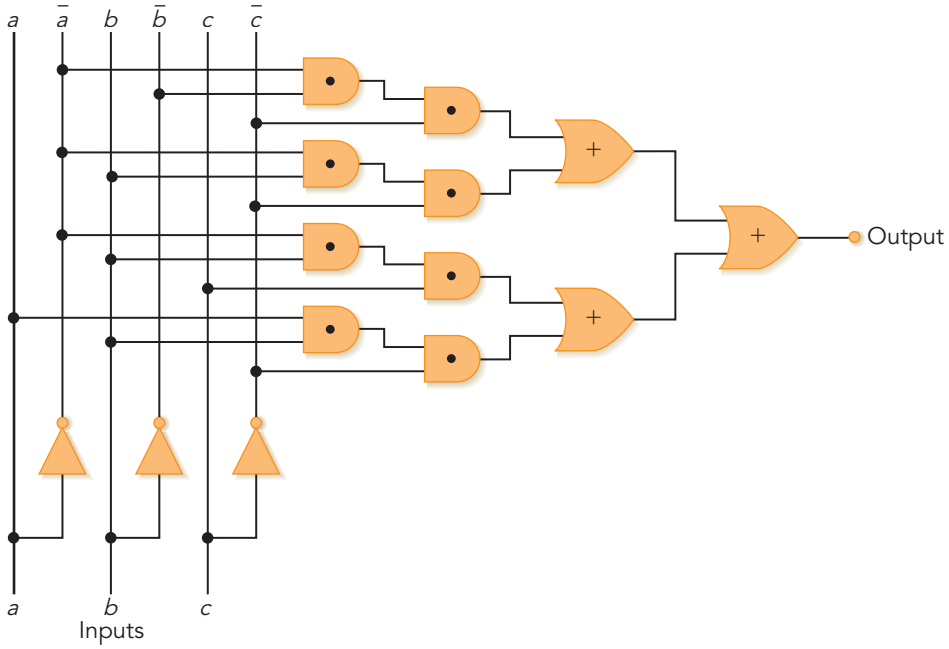
1. The four separate cases are

   $\bar{a}\cdot\bar{b}\cdot\bar{c}$ $\bar{a}\cdot b\cdot\bar{c}$ $\bar{a}\cdot b\cdot c$ $a\cdot b\cdot\bar{c}$

   Combining them by using the OR operator produces the following Boolean expression:

   $\bar{a}\cdot\bar{b}\cdot\bar{c} + \bar{a}\cdot b\cdot\bar{c} + \bar{a}\cdot b\cdot c + a\cdot b\cdot\bar{c}$

When this Boolean expression is represented as a Boolean diagram, it appears as follows:



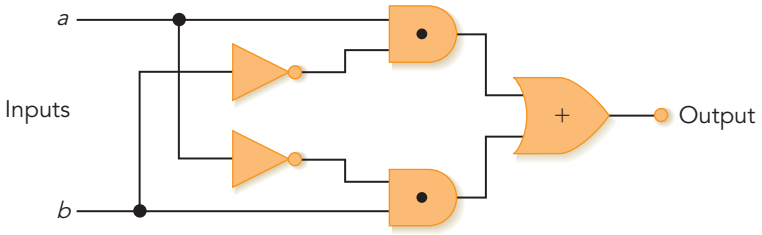
2. The Boolean expressions for the two cases are

   $\bar{a} \cdot b$
   $a \cdot \bar{b}$

   Combining these two by using the OR operator produces

   $\bar{a} \cdot b + a \cdot \bar{b}$

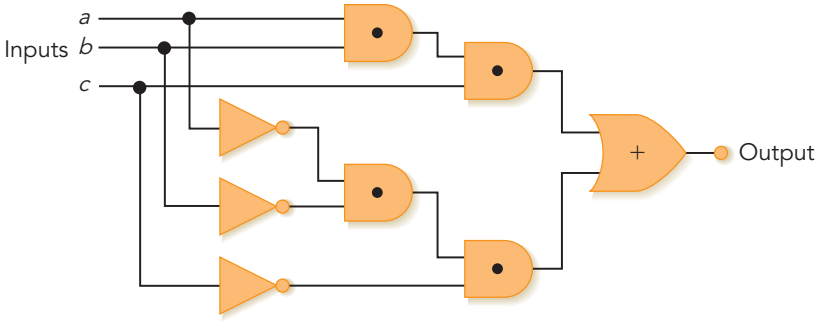
   Pictorially, the corresponding circuit diagram is



3. The Boolean expression for this is

   $\bar{a} \cdot \bar{b} \cdot \bar{c} + a \cdot b \cdot c$

   Pictorially, the corresponding circuit diagram is



4. If we use the circuit construction algorithm described in this chapter, we produce the circuit: $\bar{a}\cdot\bar{b} + \bar{a}\cdot b$. When this expression is implemented as a logic circuit, it takes two NOT gates, two AND gates, and one OR gate for a total of five gates. However, looking carefully at the truth table, we see that the output is a 1 whenever a is a 0, and the output is a 0 whenever a is a 1. The output is not affected by the value of b. Thus, an equivalent circuit is $\bar{a}$, which takes only 1 gate—an improvement of 80%. This is a good example of how much optimization can improve a preliminary design, and how important it can be to the efficiency of a computer system.

5. Since the truth table for C has 4 distinct inputs, there would be $2^4 = 16$ rows, representing the 16 possible inputs, 0000 to 1111. There are 4 input columns (for *a*, *b*, *c*, *d*) and 3 output columns (for *output-1*, *output-2*, and *output-3*), for a total of 7 columns. Thus the dimensions of the truth table for circuit C would be 16×7.

Section 4.4.3

1. Each 1-CE circuit contains two NOT gates, two AND gates, and one OR gate. This will require $(2 \times 1) + (2 \times 3) + (1 \times 3) = 2 + 6 + 3 = 11$ transistors. The 32-bit compare-for-equality circuit of Figure 4.28 has the following components:

   32 1-CE circuits $= 32 \times 11 = 352$ transistors
   31 AND gates $= 31 \times 3 = 93$ transistors
   Total $= 352 + 93 = 445$ transistors

2. Bit-compare $a > b$, where both a and b are 1 bit in length.

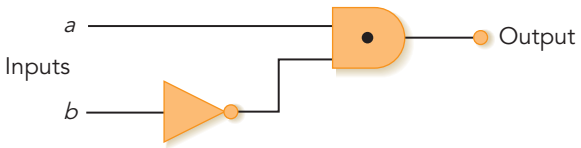
   The truth table for this circuit would be as follows:

a	*b*	*Output*	
0	0	0	
0	1	0	
1	0	1	(because *a* is greater than *b*)
1	1	0	

There is only one case where there is a 1 bit in the output. It is in the third row and corresponds to the following Boolean expression:

$a \cdot \bar{b}$

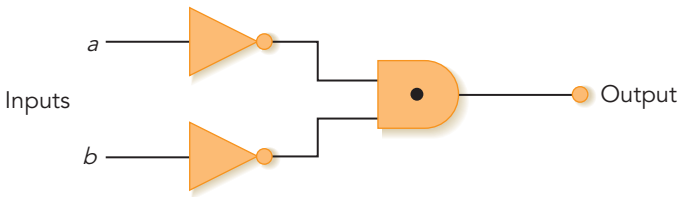
We can skip Step 3 because with only one case, there is no combining of Boolean expressions. Thus, the circuit diagram for this circuit is



3. The truth table is already given, and again there is only one case with a 1 bit in the output. This occurs in the first row and corresponds to the Boolean expression

   $\bar{a} \cdot \bar{b}$

   Given this single subexpression, we can proceed immediately to draw the circuit diagram:

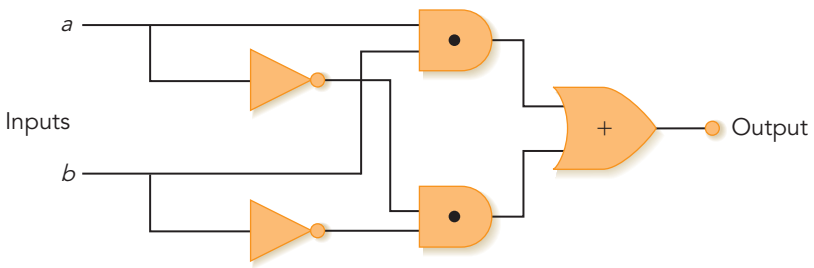


4. The truth table is already given. There are two cases with a 1 bit in the output, occurring in rows 1 and 4. The corresponding subexpressions are:

   Case 1: $\bar{a} \cdot \bar{b}$

   Case 2: $a \cdot b$

   The final Boolean expression is $\bar{a} \cdot \bar{b} + a \cdot b$ and the circuit diagram is



Chapter 5

Section 5.2.1

1. If the memory unit is a two-dimensional grid 1,024 (2^{10}) by 1,024 (2^{10}), then it contains a total of 1,048,576 (2^{20}) memory cells. We need a total of 20 bits to represent all the possible memory addresses, which range from 0 to $2^{20} - 1$.
2. Because there are $2^{10} = 1{,}024$ row lines and $2^{10} = 1{,}024$ column lines, we would need to send 10 (of the 20) bits in the MAR to the row decoder and 10 bits to the column decoder.
3. The instructions that are fetched from memory are usually much larger than 8 bits in size, often 16 or 32 bits. If the MDR were only 8 bits, then to fetch a 16-bit instruction we would have to go back to memory twice. If the instruction were 32 bits, then we would have to go back to memory four separate times. This would greatly slow down the fetch phase of the instruction cycle. The MDR should be at least as large as the largest instruction that we fetch.
4. Average access time $= (0.80 \times 10) + 0.20 \times (10 + 25) = 15$ nsec
5. Average access time $= (0.90 \times 10) + 0.10 \times (10 + 25) = 9 + 3.5 = 12.5$ nsec
6. With a cache hit rate of 92%, the average access time is

   $(0.92 \times 10) + 0.08 \times (10 + 25) = 12.0$ nsec
7. Each byte contains 8 bits, so 8 bytes contain 64 bits. Using two's complement representation, the largest positive value would be:

   $$+2^{63} - 1 = +9{,}223{,}372{,}036{,}854{,}775{,}807$$

   while the largest negative value would be:

   $$-2^{63} = -9{,}223{,}372{,}036{,}854{,}775{,}808$$
8. For example: If we think that human memory consists of "cells" in which information is stored and from which it can be retrieved, then human memory does not seem to have the property of a uniform time to access every cell. "Short-term memories" seem to be more quickly accessible than long-term memories.

Section 5.2.2

1. The total number of characters (ch) is

   2 surfaces/disk $\times$ 50 tracks/surface $\times$ 20 sectors/track $\times$ 1024 ch/sector, which is 2,048,000 characters on a single disk.
2. The seek time depends on the number of tracks over which the read head must move. This could range from 0, if the arm does not need to move,

to a worst case of the arm having to move from the far inside track to the far outside track, a total of 49 tracks. The average, as stated in the problem, is a move across 20 tracks. The best-case rotational delay is 0, whereas the worst case is one complete revolution. The rotational speed is 2,400 rev/min = 40 rev/sec = 25 msec/rev. On the average, we will wait about one-half of a revolution. Finally, the transfer time is the same in all cases, the time it takes for one sector (1/20 of a track) to rotate under the read/write head, which is 1/20 rev × 25 msec/rev = 1.25 msec. Putting all this together in a table produces the following values for the time (in msec) required for each task:

	Best Case	*Average Case*	*Worst Case*
Seek time	0.0	20 × 0.4 = 8.0	49 × 0.4 = 19.6
Latency	0.0	0.5 × 25 = 12.5	1 × 25 = 25.0
Transfer	1.25	1.25	1.25
Total	1.25	21.75	45.85

3. The new rotational speed is 7,200 rev/min = 120 rev/sec = 8.33 msec/rev. The seek time is unaffected by the rotational speed. The new average latency time is 0.5 rev × 8.33 msec/rev = 4.17 msec. The new transfer time is 1/20 rev × 8.33 msec/rev = 0.42 msec. The total average case time = 12.59 msec.

4. average seek time = 20 tracks × 0.2 msec/track = 4 msec
average latency = 0.5 rev × 8.33 msec/rev = 4.17 msec
transfer time = 1/20 rev × 8.33 msec = 0.42 msec

   So the average access time = 4 + 4.17 + 0.42 = 8.59 msec.

5. If many pieces of the file are on the same track, then no movement of the read head arm is required and seek time is 0.

6.

	Best Case	*Average Case*	*Worst Case*
Seek time	0.0	0.0	0.0
Latency	0.0	0.5 × 25 = 12.5	1 × 25 = 25.0
Transfer	1.25	1.25	1.25
Total	1.25	13.75	26.25

Section 5.2.4

Assuming that variables *a*, *b*, *c*, and *d* are stored in memory locations 100, 101, 102, and 103, respectively:

1.

Memory Location	*Op Code*	*Address Field*	*Comment*
50	LOAD	101	Register R now contains the value of *b*
51	ADD	102	R now contains the sum *b* + *c*
52	ADD	103	R now contains the sum *b* + *c* + *d*
53	STORE	100	And we store that sum into *a*

There are many other possible solutions to the previous and the following problems, depending on which instructions you choose to use. The previous solution uses the one-address format. The two- and three-address formats would lead to different sequences.

2.

Memory Location	Op Code	Address Field	Comment
50	LOAD	101	Register R contains the value of b
51	MULTIPLY	103	R now contains the product $b \times d$
52	STORE	101	b now has the value $b \times d$
53	LOAD	102	R contains the value of c
54	DIVIDE	103	R now contains the quotient c/d
55	STORE	103	d now has the value c/d
56	LOAD	101	Load the product $b \times d$ back into R
57	SUBTRACT	103	R now contains $(b \times d) - (c/d)$
58	STORE	100	Store the result into a

3.

Memory Location	Op Code	Address Field	Comment
50	LOAD	100	Register R now contains the value of a
51	SUBTRACT	104	R now contains the value $a - 1$
52	STORE	100	Store the result into a

4.

Memory Location	Op Code	Address Field	Comment
50	COMPARE	100, 101	Compare a and b and set condition codes
51	JUMPNEQ	54	If they are not equal, go to address 54
52	LOAD	103	Otherwise load R with the value of d
53	STORE	102	And store it into c
54			The next instruction begins here

5.

Memory Location	Op Code	Address Field	Comment
50	COMPARE	100, 101	Compare a and b and set condition codes
51	JUMPGT	55	Jump to address 55 if $a >$ b
52	LOAD	103	Load R with the value of d
53	STORE	102	And store it into c
54	JUMP	58	Jump to address 58
55	LOAD	103	Load R with the value of d
56	ADD	103	R now contains $2d$
57	STORE	102	And store that result into c
58			The next instruction begins here

6.

Memory Location	Op Code	Address Field	Comment
50	LOAD	103	R contains the value *d*
51	STORE	100	And store it into *a*
52	COMPARE	100, 102	Compare *a* and *c*, set condition codes
53	JUMPGT	58	Jump to address 58 if $a > c$
54	LOAD	100	R now contains the (current) value of *a*
55	ADD	101	R now contains the value $a + b$
56	STORE	100	And store that sum into *a*
57	JUMP	52	Jump back to test loop condition again
58			The next instruction begins here

Chapter 6

Section 6.3.1

1. Initial values $R = 20$ memory location 80 = 43, memory location 81 = 97

	Operation	Final Contents of Register R	Final Contents of Mem Loc 80	Final Contents of Mem Loc 81
a.	LOAD 80	43	43	97
b.	STORE 81	20	43	20
c.	COMPARE 80	20 (and the GT indicator goes ON)	43	97
d.	ADD 81	117	43	97
e.	IN 80	20	Whatever value is entered by the user	97
f.	OUT 81	20	43	97

2. Initial value memory location 50 = 4

	Operation	Final Contents of Register R
a.	LOAD 50	4
b.	LOAD 4	A copy of the contents of memory cell 4.
c.	LOAD L	Because *L* is equivalent to 50, this operation is equivalent to LOAD 50, which is the same as part (a).
d.	LOAD L+1	A copy of the contents of memory cell 51. This operation means LOAD ($L + 1$), which is equivalent to LOAD 51. LOAD $L + 1$ does arithmetic on addresses, not contents.

3. The HALT operation tells the CPU to stop program execution. If the program is organized as in Figure 6.6, then without the HALT instruction the CPU will fetch the data value stored in the next memory location after the last instruction and attempt to execute it. The .END pseudo-op tells the assembler to stop the translation process. The assembler is a piece of software that is acting on the source code, loaded into memory, as its "data"; without the .END pseudo-op, the assembler will try to translate whatever might be stored in memory after the last legitimate source code statement.

4. The first instruction will cause the memory location labeled L to be loaded into register R. Because L contains the data value +1, this will go into R, overwriting whatever was there previously. After completing one instruction, a processor will go on to the next one unless told to do otherwise—that is the essence of the Fetch/Decode/Execute cycle. Thus, the processor will next try to execute the "instruction" +1. As we explained in the text, this will be incorrectly interpreted as the op code 0 and address field of 1, which is a LOAD 1. Thus, the value +1 in register R will be overwritten with the contents of memory location 1.

5. a. The SUBTRACT op code is 0101. The binary representation for the unsigned integer 20 using 12 bits is 000000010100. Putting this together results in:

   0101000000010100

   b. The LOAD op code is 0000. The binary representation for the unsigned integer 31 using 12 bits is 000000011111. Putting this together results in:

   0000000000011111

   c. The HALT op code is 1111. HALT does not require an address but something has to be put into those 12 bits. Typically, an assembler will set all the bits to 0, although it does not really matter. Assuming we put zeroes into the address field we will end up with:

   1111000000000000

Section 6.3.2

1. a.
```
        INCREMENT X
        .
        .
        .
X:      .DATA       0
```
Another way to do the same thing is
```
        LOAD        X
        ADD         ONE
```

```
                STORE     X
                .
                .
                .
ONE:            .DATA     1
X:              .DATA     0
```

However, the first way is much more efficient. It takes two fewer instructions and one fewer DATA pseudo-op.

b.
```
                LOAD      X
                ADD       FIFTY
                STORE     X
                .
                .
                .
FIFTY:          .DATA     50
X:              .DATA     0
```

c.
```
                LOAD      Y         --Load the value of Y into register R
                ADD       Z         --R now holds the sum (Y + Z)
                SUBTRACT  TWO       --R now holds (Y + Z - 2)
                STORE     X         --Store the result in X
                .
                .
                .
X:              .DATA     0
Y:              .DATA     0
Z:              .DATA     0
TWO:            .DATA     2
```

d.
```
                LOAD      FIFTY     --R holds the constant 50
                COMPARE   X
                JUMPGT    THEN      --if X > 50 go to label THEN
                IN        X         --input a new value
                JUMP      DONE      --and jump to done because we are
                                    --all finished
THEN:           OUT       X
DONE:                               --the next statement goes here
                .
                .
                .
X:              .DATA     0
FIFTY:          .DATA     50
```

```
e.          LOAD      ZERO      --Put 0 in R
            STORE     SUM       --Initialize SUM to 0
            STORE     I         --Initialize loop counter to 0
     LOOP:  LOAD      FIFTY     --Put 50 in R
            COMPARE   I
            JUMPEQ    DONE      --I equals 50, exit loop
            LOAD      SUM       --Put SUM in R
            ADD       I         --R now holds (SUM + I)
            STORE     SUM       --Store result in SUM
            INCREMENT I         --Add 1 to I
            JUMP      LOOP      --end of loop body
     DONE:                      --the next statement goes here
            .
            .
            .
     I:     .DATA     0
     SUM:   .DATA     0
     ZERO:  .DATA     0
     FIFTY: .DATA     50
```

2.

```
            .BEGIN
            IN        NUMBER    --get first number
LOOP:       LOAD      ZERO
            COMPARE   NUMBER    --see whether number < 0
            JUMPLT    DONE      --the number is negative so
                                --go to DONE
            INCREMENT COUNT     --it is nonnegative so
                                --increment count
            IN        NUMBER    --get next number
            JUMP      LOOP      --and repeat the loop
DONE:       OUT       COUNT     --print out the final count
            HALT
COUNT:      .DATA     0         --count of number of
                                --nonnegative values
ZERO:       .DATA     0         --the constant 0 used for
                                --comparison
NUMBER:     .DATA     0         --place to store the input
                                --value
            .END
```

3.
```
            .BEGIN
            IN          NUMBER      --get first number
LOOP:       LOAD        ZERO
            COMPARE     NUMBER      --see whether number < 0
            JUMPLT      DONE        --the number is negative so
                                    --go to DONE
            INCREMENT   COUNT       --number is nonnegative so
                                    --increment count
            LOAD        HUNDRED
            COMPARE     COUNT
            JUMPEQ      DONE        --count = 100 so go to DONE
            IN          NUMBER      --count < 100 so get next
                                    --number
            JUMP        LOOP        --and repeat the loop
DONE:       OUT         COUNT       --count is either 100 or a
                                    --legitimate count of the less
                                    --than 100 nonnegative
                                    --numbers read, so print
                                    --count in either case
            HALT
COUNT:      .DATA       0           --count of number of
                                    --nonnegative values
ZERO:       .DATA       0           --the constant 0 used for
                                    --comparison
NUMBER:     .DATA       0           --place to store the input
                                    --value
HUNDRED:    .DATA       100         --the constant 100
            .END
```

4. The most important thing was that you were looking at a program that contained English words and familiar mathematical terminology. That made it much easier to understand what was going on and where the modifications were to be made. Also, the use of labels allowed you to make changes in the program without having to make changes to binary memory addresses.

Section 6.3.3

1.
```
            .BEGIN
            CLEAR       NEGCOUNT    --Step 1. Not really necessary
                                    --because
                                    --negcount is already set to 0
```

```
            LOAD        ONE         --Step 2. Set i to 1. Also not
                                    --really
            STORE       I           --necessary because i is
                                    --initialized to 1
LOOP:       LOAD        FIFTY       --Step 3. Check whether i > 50
                                    --and if so
                                    --terminate the loop
            COMPARE     I
            JUMPGT      ENDLOOP
            IN          N           --Step 4. Read a value
            LOAD        ZERO        --Step 5. Increment
                                    --negcount if
            COMPARE     N           --N is less than zero
            JUMPGE      SKIP
            INCREMENT   NEGCOUNT
SKIP:       INCREMENT   I           --Step 6. Count one more
                                    --loop iteration
            JUMP        LOOP        --Step 7. and start the loop
                                    --over
ENDLOOP:    OUT         NEGCOUNT    --Step 8. Produce the final
                                    --answer
            HALT                    --Step 9. and halt
NEGCOUNT:   .DATA       0
I:          .DATA       1
N:          .DATA       0
ONE:        .DATA       1
FIFTY:      .DATA       50
ZERO:       .DATA       0
            .END
```

2. a. COMPARE = 0111 Y = decimal 10 = 0000 0000 1010
   instruction = 0111 0000 0000 1010

   b. JUMPNEQ = 1100 DONE = decimal 7 = 0000 0000 0111
   instruction = 1100 0000 0000 0111

   c. DECREMENT = 0110 LOOP = decimal 0 = 0000 0000 0000
   instruction = 0110 0000 0000 0000

3. LOOP is the address of an instruction (IN X), but decrement is treating this instruction as though it were a piece of data and subtracting 1 from it. Thus, what this instruction is doing is "computing" (IN X) – 1, which is meaningless. However, the computer will be very happy to carry out this meaningless operation.

4. The address values that you come up with will depend entirely on your solution. The symbol table for the program in Practice Problem 1 would look like the following example. (*Note*: The solution assumes that each instruction occupies one memory location.)

Symbol	*Address*
LOOP	3
SKIP	11
ENDLOOP	13
NEGCOUNT	15
I	16
N	17
ONE	18
FIFTY	19
ZERO	20

5. No, it is not illegal, just highly misleading. You can label a piece of data with any name you want, but you should pick something that is helpful to the people who have to read and modify the program at some future time. By labeling the constant +1 with the symbolic name TWO, you will confuse and mislead the people looking at your code.

Section 6.4.1

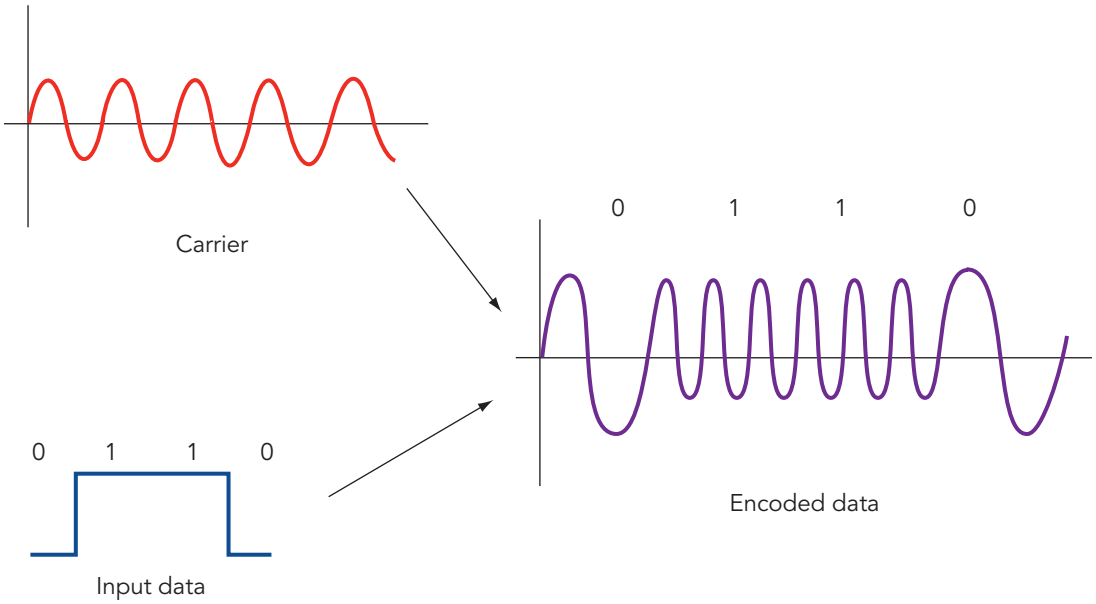
1. If there is one chance in four that a program is blocked waiting for input/output, then there is a (1/4) × (1/4) = 1 chance in 16 that both of the two programs in memory are simultaneously blocked waiting for I/O. Therefore, the processor will be busy 15/16, or about 94%, of the time. This is the processor utilization. If we increase the number of programs in memory to four, then the probability that all four of these programs are blocked at the same time waiting for I/O is (1/4) × (1/4) × (1/4) × (1/4) = 1 chance in 256. Now the utilization of the processor is 255/256, or about 99.6%. We can see clearly now why it is helpful to have more programs in memory. It increases the likelihood that at least one program will always be ready to run.

2. Passwords are not secure because people often write them down on a piece of paper put in plain sight for all to see. Furthermore, when choosing passwords people will often pick something that can be easily guessed, like their child's name or birthday. You can reduce the risk of using passwords by requiring passwords to (1) be at least 8–10 characters in length, (2) include at least one special character like /$*&, and (3) not be something found in a dictionary. You can also use *personal information* rather than a memorized password. Personal information is a fact that you would know without having to write it down, but which an acquaintance or coworker would likely not know—maybe the name of your first pet or your maternal grandfather's first name.

Chapter 7

Section 7.2.1

1. The figure shows the representation of a binary signal using frequency modulation of a carrier wave.



2. The number of bits in the image is $1{,}200 \times 780 \times 8 = 7{,}488{,}000$. To transmit this in 1 second requires a transmission speed of 7,488,000 bps, or nearly 7.5 Mbps.
3. The number of bits in the image is $1{,}200 \times 780 \times 8 = 7{,}488{,}000$. 7.488 million bits / 1 billion bits/sec = 0.007488 seconds or 7.48 milliseconds

Section 7.2.2

1. Because the message is to be broadcast, there is no need to include a destination address. Each node reads the message.
2. Node A sends the message on LAN1, where every node receives it, but only bridge B1 keeps it. All other nodes discard it because it is not addressed to them. Bridge B1 removes the message from LAN1 and rebroadcasts it on LAN2. Every node on LAN2 receives the message, but again, only bridge B2 keeps it. All others discard it. Bridge B2 knows that node B is located on LAN3, so it rebroadcasts the message on LAN3, where node B receives it, recognizes its own address, and removes the message from the network. The message has arrived at its intended destination.
3. The main security concern is that in an Ethernet-like bus topology every node receives a copy of every message sent. A node is supposed to look at the address field of the message to see if it is the desired recipient

and, if not, discard that message. However, a malicious node could keep, read, and record every message on the network, even those that are not intended for it.

Section 7.3.2

1. A has resent M(3), presumably because B's ACK(3) message never reached A. When B receives the second copy of M(3), it should again send the ACK(3) to A but discard the message because it is a duplicate.
2. B should disassemble M(4) and check that it was correctly transmitted. If so, B should send an ACK(4) message to A; if not, B should discard the faulty message and wait for A to resend it.
3. B should disassemble M(5) and check that it was correctly transmitted. If so, B should send an ACK(5) message to A; if not, B should discard the faulty message and wait for A to resend it. Although this message has been received out of sequence (presumably because M(4) was lost), no special action is required at this time; B did not send an ACK(4) so A should in time resend M(4).
4. This statement is false. Two nodes can have a collision and then pick the same random waiting time. There will then be a second collision, and it is possible that they will again pick the same waiting time, leading to a third collision. Theoretically, it is possible to have an infinite number of collisions, although the probability of that happening is vanishingly small.
5. Collisions can occur in a ring topology because all nodes share the ring and a node may want to send a message at the same time that a message from another node is passing by. In a star topology, collisions are avoided because each node has a direct line (via the hub node) to any other node.
6. Sending a 100-character message, 800 bits, across a 100 million bit per second (100 Mpbs) Ethernet takes about 8×10^{-6} seconds, or 8 μsec. The likelihood that one of the other 19 users on the system would transmit during that tiny window of time is quite small, so the probability of a collision would be small.

Section 7.3.3

1. There are four distinct paths from node A to node D, and their total weights are

ABCD	Weight = 16
ABFD	Weight = 14
AEFBCD	Weight = 25
AEFD	Weight = 15

   So the shortest path is ABFD, found by computing the weight of every possible path and then picking the smallest. This is essentially a "brute force" approach to the problem.

2. This approach would not work with larger graphs such as one with 26 nodes and 50 links. The number of possible paths would grow much too large for us to enumerate and evaluate them all in a reasonable amount of time. We must use a more clever algorithm.

3. If the link connecting node F to node D fails, then the paths ABFD and AEFD will not work and their weights become "infinite." Of the two paths remaining, path ABCD with weight 16 now becomes the shortest path. No one link in the network will disconnect nodes A and D. We can see that clearly by noting that the two paths ABCD and AEFD do not share any links in common. Therefore, if a link along one of these paths fails, we can use the other path.

4. Yes, the shortest path would change. The shortest path from A to E would now be the route AEFD with a cost of $4 + 2 + 7 = 13$ units. It had been ABFD with a cost of $3 + 4 + 7 = 14$ units.

5. If node B failed, no other nodes in the network would be brought down, although their communication delays might increase. Nodes A, C, D, E, F could all send and receive messages from the other active nodes.

Chapter 8

Section 8.2.1

1. **Step 1:** chjbup5 → 3 8 10 2 21 16 5

   Step 2: $3 + 8 + 10 + 2 + 21 + 16 + 5 = 65$

   Step 3: 65 / 7 leaves a remainder of 2 ($65 = 9 \times 7$ with 2 left over)

   Steps 4 and 5: are identical to the example.

2. ed

3. No; from Problem 2, the encrypted form of Judy's password "judy" is *ed*, but "mike" hashes to *fc*. The encrypted versions do not match.

4. The file is not legitimate. Using the hash function described, the final step is to change each digit back to a letter. The maximum digit is 9, which would correspond to *i*, so no letters beyond *i* can appear in an encrypted password.

Section 8.3.2

1. STB NX YMJ MTZW

2. Because there are 26 letters in the alphabet, a shift of $s = 26$ encrypts each character as itself, so you should not trust the messenger.

3. a. Step 1. Apply the S mapping to (M Q) to get (13 17).

   Step 2. Multiply result times M: $(13\ 17) \times \begin{bmatrix} 3 & 5 \\ 2 & 3 \end{bmatrix}$

   $= (13 \times 3 + 17 \times 2 \quad 13 \times 5 + 17 \times 3) = (73\ 116) \rightarrow (21\ 12)$

   Step 3. Apply S' to (21 12) to get (U L).

   b. Step 1. Apply the S mapping to (U L) to get (21 12).

   Step 2. Multiply result by M': $(21\ 12) \times \begin{bmatrix} 23 & 5 \\ 2 & 23 \end{bmatrix}$

   $= (21 \times 23 + 12 \times 2 \quad 21 \times 5 + 12 \times 23) = (507\ 381) \rightarrow (13\ 17)$

   Step 3. Apply S' to (13 17) to get (M Q)

Section 8.3.3

In the example, where $n = 21$ and $e = 5$, then $d = 5$ as well. To decode, compute

$C^d = 17^5 = 1419857 = 67612 \times 21 + 5 \rightarrow 5$

Therefore the original numeric message was 5.

Chapter 10

Section 10.2.2

1. `ITIME .LE. 7`

2.
```
    IF (X .LT. 3) THEN
        A = 2
    ELSE
        A = 1
        B = 3
    ENDIF
30 ...
```

3. Because X has the value 2 and $2 < 3$, control transfers to statement 10, which sets the value of A to 2. The next statement transfers control to statement 30, at which point A has the value 2 and B still has the value 1 because statement 20 was never executed. Therefore the new value of A is $2 + 1 = 3$.

Section 10.2.3

1. In Ada:
```
outputNumber := inputNumber;
sumOfValues  := sumOfValues + inputNumber;
```

In C++, C#, or Java:

```
outputNumber = inputNumber;
sumOf Values = sumOfValues + inputNumber;
```

In Python:

```
outputNumber = inputNumber
sumOfValues = sumOfValues + inputNumber
```

Section 10.2.4

1. *Rate* refers to the contents of the memory cell called *Rate*; &*Rate* refers to the address of that cell.
2. 10
3. 500; 29

Section 10.2.5

The program prints the numbers from 1 through 10 on a single line with a blank space between them.

Section 10.2.6

```
The answer is7
```

Section 10.2.7

The Python version is answer c,

```
print("Hello World")
```

Section 10.2.8

The name entered in the text box will be displayed in the label.

Section 10.3.3

1. It results in the names of all vendors from Chicago.
2. These are the *times* that try **men's souls**

Section 10.4.1

1. a. `(2 3 4)`
   b. `5`
2.
```
(define (threeplus x)
(+ 3 x))
```

Section 10.4.2

1. before(jefferson, kennedy).
   false
2. president(X, lewisandclark).
   X = jefferson
3. precedes(jefferson, X).
   X = lincoln
   X = fdr
   X = kennedy
   X = nixon

Section 10.4.3

1. One processor could compute A + B while another computes C + D. A third processor could then take the two quantities A + B and C + D and compute their sum. Parallel processing uses a total of two time slots: one to simultaneously do the two additions A + B and C + D, then one to do the addition (A + B) + (C + D). Sequential processing would require a total of three time slots: (A + B), then (A + B) + C, then ((A + B) + C) + D.
2. One solution would be to have the input/output processor set a timer for the maximum possible response time from each of the search processors and if that time is exceeded, it reports that NUMBER was not found. This is not foolproof, because it could simply be that the communications link between the input/output processor and the one processor that found NUMBER is broken, so it was unable to report back the corresponding NAME. A better solution would be to have each search processor report back to the input/output processor when it cannot find NUMBER; when the input/output processor has received that message from all 1,000 search processors, it would then report that NUMBER was not found.

Chapter 11

Section 11.2.1

1. a.

Token	*Classification*
x	1
=	3
x	1
+	4
1	2
;	6

b.

Token	*Classification*
if	8
(	10
a	1
+	4
b42	1
==	7
0	2
)	11
a	1
=	3
zz	1
-	5
12	2
;	6

2. a. This would most likely be classified as three tokens: the name abc, the minus sign '-', and the name def. The reason is that a minus sign is rarely allowed to be part of a variable name.

   b. This would likely be classified as a single token with the name abc_def. In many programming languages, the underscore character is allowed to be part of a name to allow the different parts of the variable to stand out, for example, the names inches_per_yard or cost_per_square_meter.

   c. This would definitely be classified as two separate tokens, abc and def, because a space is always used to end one token and to identify the start of the next one.

3. A scanner is not involved in the determination of the legality or illegality of a programming language statement; that is done in the next two steps. A scanner is only concerned with finding and classifying tokens. So it would simply return the following six tokens:

Name	*Classification*
X	1
=	3
;	6
(	10
14	2
hello	1

4. The symbol "else", which was chosen as a variable name, will not be assigned symbol category 1 as we would expect. Instead, it will be assigned to category 9, the reserved word "else". In the next phase of translation, when the compiler analyzes the syntactic structure of this

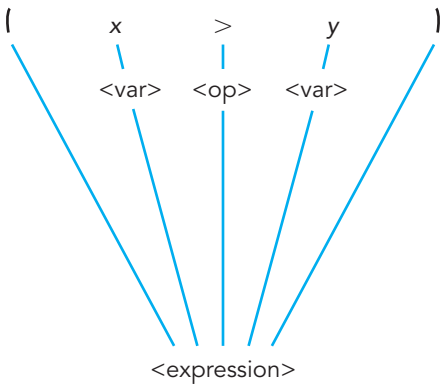
statement, it will not be able to understand what the user meant and will produce an error message. This is why in most programming languages you are not allowed to use reserved words (such as while, do, if, else) as names of variables.

Section 11.2.2 (Set 1)

1. <Boolean operator> ::= AND | OR | NOT

2. <identifier> ::= <first><second>
   <first> ::= i | j
   <second> ::= <letter> | <digit> | Λ
   <letter> ::= A | B | C | D | ... | Z
   <digit> ::= 0 | 1 | 2 | ... | 9
   <identifier> is the goal symbol

3. <expression> ::= (<var> <op> <var>)
   <var> ::= x | y | z
   <op> ::= < | == | >

4. 

5. You eventually reach the point in the parse where you have the following sequence:

   (<var> <op>)

   which does not match the right-hand side of any rule, and the parse fails.

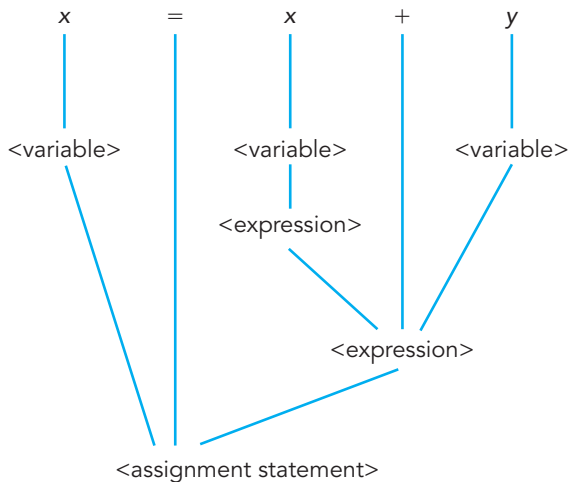
6. The first rule of Practice Problem 3 could be changed to

   <expression> ::= (<var> <op> <var>) | <var> <op> <var>

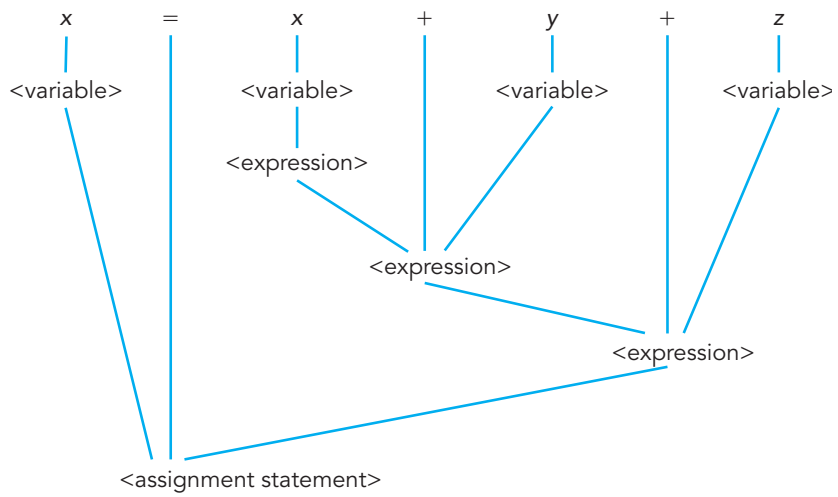
7. The symbol Λ and the blank are NOT interchangeable. Blank is a character, just like A or 5 or *. However, the symbol Λ means the absence of any character, including a blank space.

Section 11.2.2 (Set 2)

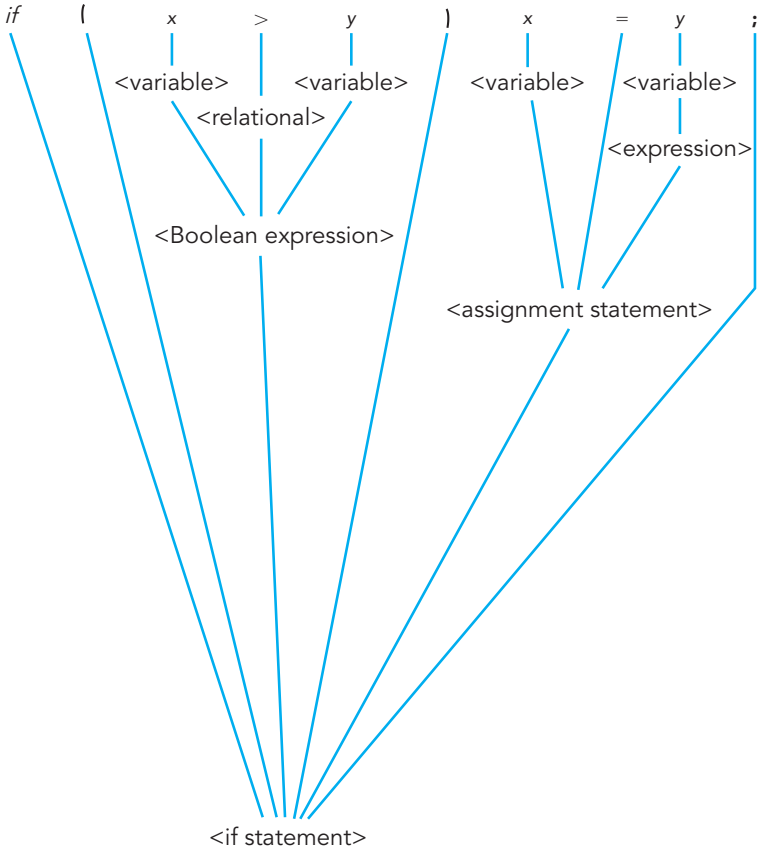
1.



2.



3.



4. The language consists of all strings of length 1 or more containing an arbitrary sequence of *a*s and *b*s.

5. <goal> ::= <pair> | <pair> <goal>
<pair> ::= AB

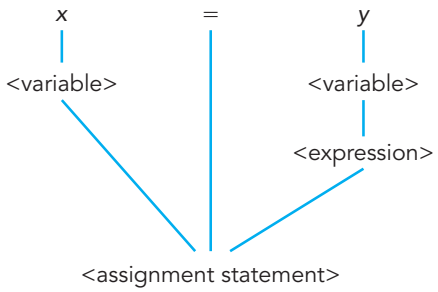
6. No, this grammar is not ambiguous. There are only four sentences that belong to the language defined by this grammar:

   *a**c, *a**d, *b**c, and *b**d*

   and all of these sentences have only a single possible parse tree.

Section 11.2.3

The parse tree for this expression is



During the construction of this parse tree, you will build four semantic records: two for <variable>, one for <expression>, and one for <assignment statement>.

The code generated is

```
                LOAD    Y
                STORE   TEMP
                LOAD    TEMP
                STORE   X
                .
                .
                .
X:              .DATA   0
Y:              .DATA   0
TEMP:           .DATA   0
```

Chapter 12

Section 12.2

1.

Rate in mph	Time in hours	Distance in miles
58	0.5	29
61	0.5	30.5
57	0.5	28.5
62	0.5	31
	Total	119

2. Piloting a boat, performing an operation, fighting a fire
3. Soil conditions, water supply, types of industrial waste. It could illustrate long-term effects of various waste-disposal policies. If it were inaccurate, policies based on the model might be pursued that would result in environmental damage.

Section 12.3.2

1. *b* 1 1 1 *b*
   ↑
   3
2. a. *b* 1 0 1 *b*
   ↑
   2

   b. *b* 1 1 1 *b*
   ↑
   2

c. $b\ 1\ 1\ 1\ b$
$\uparrow$
2

d. $b\ 0\ 1\ 1\ b$
$\uparrow$
1

3. $b\ 0\ 1\ 0\ b$
4. $b\ 0\ 0\ 0\ b$

Section 12.5.1

1. It is not equivalent. There is no transition from State 1 for an input symbol of 0, and no transition from State 2 for an input symbol of 1.
2. The machine would halt immediately because there are no instructions for what to do when looking at a blank cell.

Section 12.5.2

1. a. $b\ 1\ 1\ 0\ 1\ 0\ b$

Section 12.5.4

1. $b\ 1\ 1\ 1\ 1\ b\ 1\ 1\ 1\ 1\ 1\ b$ becomes
$b\ b\ b\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ b$
2. a. $(1,1,1,1,R)$
$(1,0,0,1,R)$
$(1,b,1,2,R)$

b. The machine for part (a) also solves the general problem.

Section 12.7

1. To prove that something is not true, assume that it is true and arrive at a contradiction. The assumption must then be wrong.

Chapter 13

Section 13.2.3

1. Without this instruction, server S_i is serving a newly arrived customer but is still marked as idle; the next newly arrived customer could try to go to that server.
2. Without this instruction, server S_i has finished serving a customer and there are no customers waiting in line, but the next newly arrived customer will not try to be served by S_i.

3. It most likely would make the conclusions less valid and much less usable. The idea of every customer taking the same amount of time is unrealistic. Therefore, making this assumption would cause our model to be a very poor abstraction of the real system. If your assumptions are wrong, then there is a far greater likelihood that your conclusions will be wrong as well. (We called this "garbage in, garbage out" in the text.)
4. There are many other possibilities, but here are a few:
   a. The possibility of mechanical breakdowns. The model could be designed to include the occasional breakdown of a critical component (e.g., the French fryer) to study how the system responds to these types of unexpected events.
   b. Rather than say no customer ever takes more than 5 minutes to be served, the model could be designed to allow for an occasional massive order—someone purchasing 100 hamburgers—to see what happens to our waiting times when this unusual event occurs.
   c. Some servers could be used only for special orders, rather than have all servers be identical.

Chapter 14

Section 14.2.4

1. For example:
   www.webmd.com
   https://en.wikipedia.org/wiki/Portal:Environment
   https://www.fantasysp.com
   www.petersons.com
   https://www.dawnbreaker.com/portals/altenergy
2. As an alternative, students could be asked to compare two such sites.

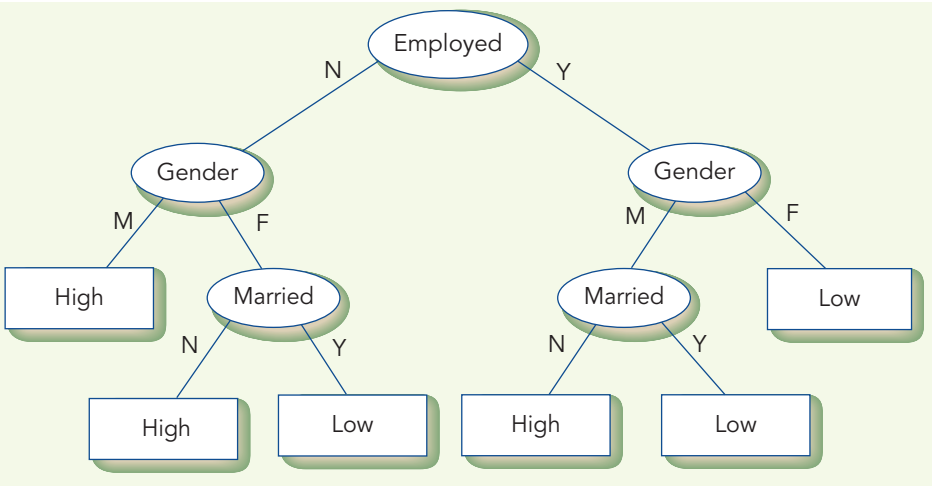
Section 14.3.3

1.
```
149      12.35
```
2.
```
SELECT LastName, FirstName, MonthlyCost
FROM Employees, InsurancePlans, InsurancePolicies
WHERE LastName = "Takasano"
AND ID = EmployeeID
AND InsurancePolicies.PlanType =
  InsurancePlans.PlanType;
```
3.
```
SELECT EmployeeID
FROM InsurancePolicies
WHERE PlanType = "B2";
```

4. Referential integrity. Removing this tuple leaves the InsurancePolicies table with tuples that have EmployeeID foreign key values no longer existing as a primary key value in the Employees table.

Section 14.4.1

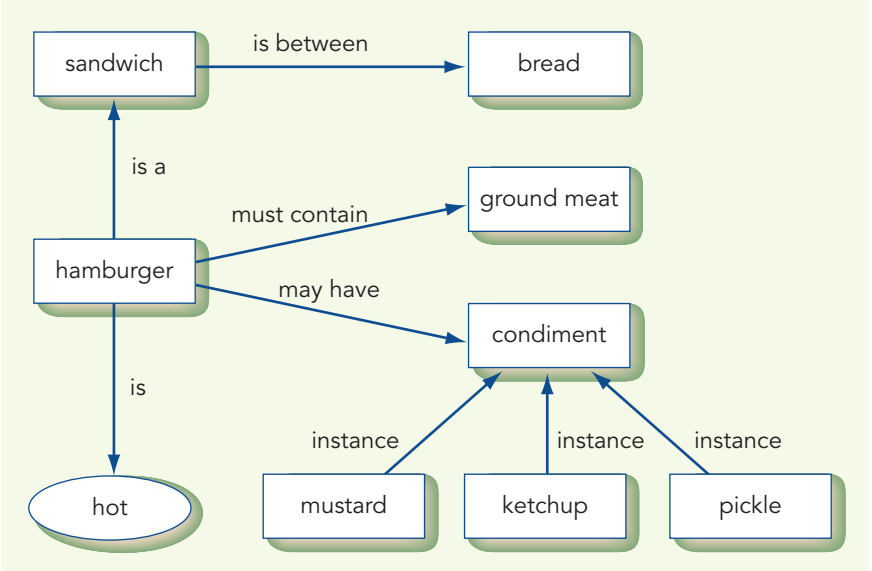
The decision tree after adding the new tuple would be



Chapter 15

Section 15.3

1. For example, a hamburger is a kind of sandwich. As such, it comes between two pieces of bread, but it is hot. It must contain ground meat, but it may also have various condiments, such as mustard, ketchup, and a pickle.



2. a. $(\forall x)(\text{hamburger}(x) \rightarrow \text{sandwich}(x))$
   b. $(\forall x)(\text{grilledCheese}(x) \rightarrow \text{sandwich}(x))$
   c. $(\forall x)(\text{sandwich}(x) \rightarrow \text{onBread}(x))$
   d. $(\forall x)(\text{grilledCheese}(x) \rightarrow \text{vegetarian}(x))$

Section 15.4

1. No. N1 and N4 fire, but N2 and N3 do not, so N5 does not.
2. It will never fire. The maximum incoming signal that node N5 can receive is $2 + 2 = 4$, which is not enough to make it fire if its threshold value is 5.

Section 15.5.4

1. Frank is tall. Knowing that Frank is tall does not necessarily mean that he is bald.
2. No conclusion can be inferred. Frank might or might not be tall.
3. No conclusion can be inferred. Frank might or might not be tall.
4. Frank is not tall (because if he were, he would be bald).

Chapter 16

Section 16.2.3

(*Note*: In the following answer, the *z* column has been omitted because it will contain all zeros.)

Vertex	*X*	*y*	*Connected To*
v_1	0	0	v_2 v_6 v_8 (Origin)
v_2	0	2.4	v_1 v_3 v_4
v_3	1.5	4.5	v_2 v_4 v_5
v_4	1.5	2.4	v_2 v_3 v_5 v_6 v_7
v_5	3	2.4	v_3 v_4 v_{10}
v_6	1	1.7	v_1 v_4 v_7 v_8 v_9
v_7	2	1.7	v_4 v_6 v_9 v_{10}
v_8	1	0	v_1 v_6 v_9
v_9	2	0	v_6 v_7 v_8 v_{10}
v_{10}	3	0	v_5 v_7 v_9

A total of 67 pieces of information are stored in this table.

Section 16.2.4

Because the motion takes place over a period of 2 seconds, we need to produce a total of 60 frames, given that 30 frames/second is the standard frame rate for video. These 60 frames represent 59 time intervals. So in each of the 58 in-between frames, we must move the triangle 1/59th of the total distance from its position in the first frame to its position in the last frame. This information allows us to compute the translation matrix.

Total x-distance moved = 4 units
$a = 1/59 \times 4 = 0.067796$

Total y-distance moved = 2 units
$b = 1/59 \times 2 = 0.033898$

Total z-distance moved = 0 units
$c = 1/59 \times 0 = 0$

Now, using the model shown in Figure 16.6, we can say that the translation matrix required to perform the desired motion is as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0.067796 \\ 0 & 1 & 0 & 0.033898 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$