Solutions to Selected Exercises

Chapter 1. Computer Science: The Mechanization of Abstraction

◇ Section 1.3

1.3.1: The static part of a data model consists of the values for the objects in the model; the dynamic part consists of the operations that can be applied to these values. For example, we can think of the set of integers with the operation addition as a data model. The static part is the set of integers and the dynamic part is the addition operator.

1.3.3: The data objects in a line-oriented text editor, such as *vi*, are files consisting of sequences of lines, where each line is a sequence of characters. A cursor identifies a position within a line. There are operators for positioning the cursor within a file. Typical operations on lines include inserting an additional line and deleting an existing line. A line may be modified by inserting, deleting, or changing characters within it. In addition, there are operators for creating, writing, and reading files.

◇ Section 1.4

1.4.1: An identifier can be one of the names for a box. For example, an identifier x in C may be attached to a box containing an integer by means of a variable declaration `int x;`. One of the names of that integer box is then x.

Chapter 2. Iteration, Induction, and Recursion

◇ Section 2.2

2.2.1(a): With 5 elements in the array, `SelectionSort` makes 4 iterations with the loop-index $i = 0, 1, 2, 3$. The first iteration makes 4 comparisons, the second 3, the third 2, the fourth 1, for a total of 10 comparisons. With the array $6, 8, 14, 17, 23$, there are no swaps (exchanges of elements) in any iteration.

2.2.1(b): On the array $17, 23, 14, 6, 8$, `SelectionSort` makes 4 iterations. The numbers of comparisons and swaps made during each iteration are summarized in the following table. We shall not regard a swap as having occurred if the selected element is already in its proper position. However, the reader should be aware that lines (6) – (8) of Fig. 2.2 are executed regardless of whether a swap is needed. Note that when $small = i$, these lines have no effect.

ITERATION	ARRAY AFTER ITERATION	NO OF COMPARISONS	NO OF SWAPS
Start	$17, 23, 14, 6, 8$	–	–
1	$6, 23, 14, 17, 8$	4	1
2	$6, 8, 14, 17, 23$	3	1
3	$6, 8, 14, 17, 23$	2	0
4	$6, 8, 14, 17, 23$	1	0

2.2.3: In what follows, we use the conventions and macros of Section 1.6. To begin, we use the cell/list macro to define linked lists of characters, as:

```
DefCell(char, CELL, LIST);
```

Here is the function `precedes`.

```
Boolean precedes(LIST L, LIST M) {
    if(M==NULL) return FALSE;
    if(L==NULL) return TRUE;
    if(L->element == M->element)
        return precedes(L->next,M->next);
    return (L->element < M->element);
}
```

2.2.5: If all n elements in the array `A` are the same, then `SelectionSort(A, n)` makes $n(n-1)/2$ comparisons but no swaps.

2.2.7: Let T be an arbitrary type. Define

```
typedef T TARRAY[MAX];
TARRAY A;
```

We modify `SelectionSort` as follows to sort elements of type T. The function `key(x)` returns the key of type K for the element `x`. We assume that the function `lt(u,v)` returns `TRUE` if `u` is "less than" `v` and `FALSE` otherwise, where `u` and `v` are elements of type K.

```
void SelectionSort(TARRAY A, int n) {
    int i, j, small;
    T temp;

    for(i=0; i<n-1; i++) {
        small = i;
        for(j=i+1; j<n; j++)
            if(lt(key(A[j]), key(A[small])))
                small=j;
        temp = A[small];
        A[small] = A[i];
        A[i] = temp;
    }
}
```

2.2.11:

a) $\sum_{i=1}^{189}(2i-1)$

b) $\sum_{i=1}^{n/2}(2i)^2$

c) $\prod_{i=3}^{k} 2^i$

◇ Section 2.3

2.3.1(a): We shall prove the following statement $S(n)$ by induction on n, for $n \geq 1$.

STATEMENT $S(n)$:

$$\sum_{i=1}^{n} i = n(n+1)/2$$

BASIS. The basis, $n = 1$, is obtained by substituting 1 for n in $S(n)$. Doing so, we get $\sum_{i=1}^{1} i = 1$. We thus see that $S(1)$ is true.

INDUCTION. Now assume that $n \geq 1$ and that $S(n)$ is true. We must prove $S(n+1)$, which is

$$\sum_{i=1}^{n+1} i = (n+1)(n+2)/2$$

We can rewrite the left-hand side as

$$(\sum_{i=1}^{n} i) + (n+1)$$

Then, using the inductive hypothesis to replace the first term, we get

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

which is the right-hand side of $S(n+1)$. We have now proven the inductive step and thus shown that $S(n+1)$ is true. We conclude that $S(n)$ holds for all $n \geq 1$.

2.3.1(b): We shall prove the following statement $S(n)$ by induction on n, for $n \geq 0$.

STATEMENT $S(n)$:

$$\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6, \text{ for all } n \geq 0$$

BASIS. $S(0)$, the basis, is $\sum_{i=1}^{0} i^2 = 0$, which is true by the definition of a sum of zero elements.

INDUCTION. Assume that $n \geq 0$ and that $S(n)$ is true. We now need to prove $S(n+1)$, which is

$$\sum_{i=1}^{n+1} i^2 = (n+1)(n+2)(2n+3)/6$$

We can rewrite the left-hand side as

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

Using the inductive hypothesis to replace the first term, we get

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

The last expression is the right-hand side of $S(n+1)$. We have now proven the inductive step. We therefore conclude $S(n)$ is true for all $n \geq 0$.

2.3.3:

a) 01101 has three 1's and is therefore of odd parity.

b) 111000111 has six 1's and is of even parity.

◇ Section 2.4

2.4.1: Our initial expression E is

$$(u+v)+((w+(x+y))+z)$$

Using the associative law for addition as in case (b), we can pull out u to get

$$(u+(v+((w+(x+y))+z))$$

We note that v happens to be pulled out as well. With another application of the associative law, we can pull out w:

$$u+(v+(w+((x+y))+z)))$$

The redundant parentheses around $x+y$ can be removed. Then, one more application of the associative law gives us our desired result:

$$u+(v+(w+(x+(y+z))))$$

2.4.3: We shall prove the following statement by complete induction on n, for $n \geq 0$.

STATEMENT $S(n)$: If an expression E has n operator occurrences, then E has $n+1$ operands.

BASIS. The basis is $n = 0$. Then E has no binary operators and one operand. Thus, $S(0)$ is true.

INDUCTION. We assume that $n \geq 0$ and that $S(j)$ is true for all $0 \leq j \leq n$. We want to prove $S(n+1)$. Let E be an expression with $n+1$ operator occurrences. Then, E is of the form $F\theta G$, where θ is a binary operator and F and G are expressions constituting the operands of θ. Let F have n_1 operator occurrences and G have n_2 operator occurrences. We know that $n_1+n_2 = n$, because the total number of operator occurrences, including θ, is $n+1$. Since n_1 and n_2 are each thus at most n, the inductive hypothesis applies. Thus, F has n_1+1 operands and G has n_2+1 operands. Therefore, E has $n+1$ operator occurrences and $n_1+1+n_2+1 = n+2$ operands, proving $S(n+1)$. We conclude $S(n)$ holds for all $n \geq 0$.

2.4.5: Matrix multiplication is associative but not commutative.

◇ Section 2.5

2.5.1: As in Fig. 2.12, we establish an invariant that is true at the top of the loop, that is, at the time when the program tests whether $i > n$ as part of the code for the for-statement. The invariant, which we prove by induction on the value of the variable `i`, is

STATEMENT $S(j)$: If we reach the test $i \leq n$ in the for-loop with the variable `i` having the value j, then the value of the variable `sum` is $j(j-1)/2$.

BASIS. The basis is $j = 1$, which occurs when we enter the for-loop for the first time. At this time, sum has its initialized value 0. Thus, $S(1)$ is true.

INDUCTION. Assume that $j \geq 1$ and that $S(h)$ is true for $1 \leq h \leq j$. We wish to prove $S(j+1)$. By the inductive hypothesis, $sum = j(j-1)/2$ as we began the jth iteration of the loop and i had the value j. After the assignment statement in the body of the loop was executed $sum = j(j-1)/2 + j = j(j+1)/2$. Thus, at the beginning of the $j+1$st iteration, sum has the value $j(j+1)/2$. Therefore, $S(j+1)$ is true.

After the nth iteration, the loop terminates with $sum = n(n+1)/2$. Thus, the program correctly evaluates $\sum_{i=1}^{n} i = n(n+1)/2$.

2.5.3: The loop invariant we shall prove by induction on k, the value of variable i, is

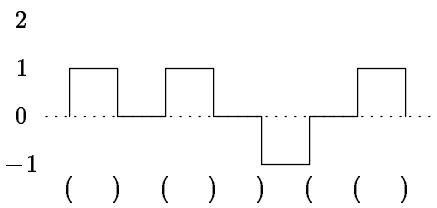
STATEMENT $S(k)$: If we reach the test $i \leq n$ in the for-loop with the variable i having the value k, then $x = 2^{2^{k-1}}$.

BASIS. The basis is $k = 1$. Upon entry to the loop, $x = 2$. Since $2^{2^{k-1}} = 2^{2^{1-1}} = 2^{2^0} = 2^1 = 2$, we see that $S(1)$ holds.

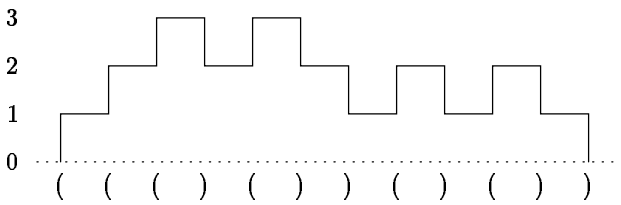
INDUCTION. Assume $k \geq 1$ and $S(i)$ is true for $1 \leq i \leq k$. We want to prove $S(k+1)$. By the inductive hypothesis, $x = 2^{2^{k-1}}$ as we entered the loop for the kth iteration and i had the value k. After the assignment statement x = x*x was executed, $x = 2^{2^{k-1}} * 2^{2^{k-1}} = 2^{2^k}$. Thus, at the beginning of the $k+1$st iteration, where i has the value $k+1$, x has the value 2^{2^k}. Therefore, $S(k+1)$ holds. After the nth iteration, when i gets the value $n+1$, the loop terminates and $x = 2^{2^n}$.

◇ Section 2.6

2.6.2(b):



2.6.2(c):



2.6.4:
a) $<$ is an infix binary operator.
b) & is a prefix, unary operator.
c) % is an infix, binary operator.

2.6.6:
a) By direct enumeration we can show that S starts off

$$0, 5, 7, 10, 12, 14, 15, 17, 19, 20, 22, 24, 25, 26, 27, 28, \cdots$$

We shall prove that 23 is the largest integer not in S.

b) We shall prove the following statement $T(n)$ by induction on n, for $n \geq 24$.

STATEMENT $T(n)$: If $n \geq 24$, then n is in S.

BASIS. The basis consists of the five integers 24, 25, 26, 27, 28, which are in S by part (a).

INDUCTION. Let us assume that $n \geq 28$ and that $T(i)$ holds for $T(24)$, $T(25)$, $T(26), \ldots, T(n)$. We want to prove $T(n+1)$.

Consider the integer $n-4$. Since $n-4 \geq 24$, by the inductive hypothesis $n-4$ is in S. By definition of S, $(n-4)+5$, or $n+1$, is in S. Therefore, $T(n+1)$ holds. We conclude that $T(n)$ holds for all $n \geq 24$.

2.6.9(a): An arithmetic expression with no operators is covered by the basis case. On the ith round we add those expressions whose trees have height i; that is, the longest path from the root to a leaf has $i+1$ nodes.

♢ Section 2.7

2.7.1:
a) Here is a C function to compute $sq(n) = n^2$, when n is a positive integer.

```
int sq(int n) {
    if(n==1) return 1;
    else return sq(n-1) + 2*n - 1;
}
```

b) We shall prove the following statement $S(n)$ by induction on n.

STATEMENT $S(n)$: $sq(n) = n^2$ when $n \geq 1$.

BASIS. When $n = 1$, the first line of `sq` returns 1.

INDUCTION. Assume that $n \geq 1$ and that $S(n)$ holds. We want to prove $S(n+1)$. From the else-statement, we know $sq(n+1) = sq(n) + 2 * n - 1$. By the inductive hypothesis, we know $sq(n) = n^2$. Therefore, $sq(n+1) = n^2 + 2(n+1) - 1 = (n+1)^2$. We have thus proved the inductive step. We conclude $S(n)$ is true for all $n \geq 1$.

2.7.3: The function `find1698` returns `TRUE` if the list contains the element 1698, and returns `FALSE` otherwise.

```
Boolean find1698(LIST L) {
    if(L==NULL) return FALSE;
    else if(L->element==1698) return TRUE;
    else return find1698(L->next);
}
```

2.7.5: The following procedure is adapted from Fig. 2.22. The array `A` and its cursor `i` is replaced by `L`, a pointer to a list of elements. The cursor `small`, indicating our current guess at the selected element, is replaced by a pointer `Small` that points to the cell of the current guess. Cursor `j`, used to run down the list of unsorted elements, is replaced by a pointer `J`, and `n`, the size of the array `A`, is implicit in the length of the given list.

```
void SelectionSort(LIST L) {

LIST J, Small;
    int temp;

    if(L!=NULL) {/* do nothing on the empty list */
        Small = L;
        J = L->next;
        while(J != NULL) {
            if(J->element < Small->element)
                Small = J;
            J = J->next;
        }
        /* now swap the elements in cells pointed to
            by L and Small */
        temp = L->element;
        L->element = Small->element;
        Small->element = temp;
        SelectionSort(L->next);
    }
}
```

2.7.7: Procedure $g(i)$ prints the remainder when i is divided by 2 and then calls itself recursively on the integer part of $i/2$. An easy inductive proof shows that

this works correctly on all positive integers. However, if $i = 0$, `g` prints nothing. Procedure `f` fixes up this problem by handling 0 as a special case.

```
void g(int i) {
    if(i>0) {
        printf("%d", i%2);
        g(i/2);
    }
}

void f(int i) {
    if(i==0) printf("0");
    else g(i);
}
```

◇ Section 2.8

2.8.1: The following table describes the sequence of events.

CALL	RETURN
`merge(1,2,3,4,5; 2,4,6,8,10)`	`1,2,2,3,4,4,5,6,8,10`
`merge(2,3,4,5; 2,4,6,8,10)`	`2,2,3,4,4,5,6,8,10`
`merge(3,4,5; 2,4,6,8,10)`	`2,3,4,4,5,6,8,10`
`merge(3,4,5; 4,6,8,10)`	`3,4,4,5,6,8,10`
`merge(4,5; 4,6,8,10)`	`4,4,5,6,8,10`
`merge(5; 4,6,8,10)`	`4,5,6,8,10`
`merge(5; 6,8,10)`	`5,6,8,10`
`merge(NULL; 6,8,10)`	`6,8,10`

2.8.5: In all of the parts, the trick is to identify an appropriate measure of size for the arguments that decreases with each recursive call.

a) A good size measure for `merge(L,M)` would be s the sum of the lengths of the lists `L` and `M`. We see that `merge` of size s calls `merge` of size $s-1$ which calls `merge` of size $s-2$, and so on, until one or the other of `L` or `M` becomes `NULL`.

c) A good size measure for `MakeList(i,n)` is $n-i$, the number of elements yet to be put on the newly created list. `MakeList` of size m calls `MakeList` of size $m-1$ which calls `MakeList` of size $m-2$ and so on until the size is 0.

◇ Section 2.9

2.9.1: Note: `PrintList` is in Fig. 2.31(a), not (b) as it states erroneously in the first printing. We shall prove by induction on i the following statement $S(i)$, for $i \geq 0$.

STATEMENT $S(i)$: If `L` is a list of length i, then `PrintList(L)` prints the elements of `L` in order.

BASIS. When $i = 0$, `L` is `NULL` and `PrintList(L)` returns without printing any elements.

INDUCTION. Assume that $i \geq 0$ and that $S(i)$ is true. We now wish to prove $S(i+1)$. Let L be a list of length $i+1$. We can write L as (a, M), where M is a list of length i.

`PrintList(L)` first prints `a` and then calls `PrintList(M)`. By the inductive hypothesis, `PrintList(M)` correctly prints the elements of `M` in order. We now have shown that `PrintList(L)` prints the elements of `L` in order. This proves the inductive step. We conclude $S(i)$ is true for all $i \geq 0$.

2.9.3: We prove by induction on i the following statement $S(i)$, for $i \geq 0$.

STATEMENT $S(i)$: If `L` is a list of length i, then `find0(L)` returns `TRUE` if 0 is an element on `L`, and returns `FALSE` otherwise.

BASIS. When $i = 0$, `L` is `NULL`. In this case, `find0` correctly returns `FALSE`.

INDUCTION. Assume that $i \geq 0$ and that $S(i)$ is true. We wish to prove $S(i+1)$. Let L be a list of length $i+1$. We can write L as (x, M), where M is a list of length i. If `x` is 0, then `find0` returns `TRUE`. If `x` is not 0, then `find0` calls `find0(M)` which, by the inductive hypothesis, returns `TRUE` if and only if `M` contains 0. Thus, `find0(L)` returns `TRUE` if 0 is an element of `L` and returns `FALSE` otherwise. This proves the inductive hypothesis. We conclude $S(i)$ is true for all $i \geq 0$.

2.9.5: The argument is essentially the same as that on p. 37 of the text, with the exception that there are two cases to the basis, $a = 0$ and $a = 1$.

Chapter 3. The Running Time of Programs

◇ Section 3.3

3.3.1: Lines (1) – (3) each time one time unit. For line (4), the test takes one unit, and it is executed n times. Lines (5) and (6) each take one unit and they are executed $n-1$ times. Line (7) takes one unit. Thus, the total time taken by the program in Fig. 2.13 is $3n+2$ units.

3.3.3: Program A takes less time than program B for all values of $n \leq 29$. For $n \geq 30$, program A takes more time than program B. At $n = 29$, program A takes 5.4×10^5 time and program B takes 8.4×10^5 time. At $n = 30$, program A takes 1.1×10^6 time and program B takes 9×10^5 time.

3.3.5: Program A takes more time than program B for $n \leq 3$, and less time thereafter.

Time Units	Maximum Problem Size Solvable with Program A	Maximum Problem Size Solvable with Program B
10^6	5	3
10^9	31	7
10^{12}	177	15

◇ Section 3.4

3.4.1: $f_1(n)$ is $O\big(f_2(n)\big)$, $O\big(f_3(n)\big)$, and $O\big(f_4(n)\big)$. In each case, we can use witnesses $c = 1$ and $n_0 = 0$.

$f_2(n)$ is not $O\big(f_1(n)\big)$, $O\big(f_3(n)\big)$, or $O\big(f_4(n)\big)$. To show that $f_2(n)$ is not $O\big(f_1(n)\big)$, suppose that it were. Then, there would be witnesses $c > 0$ and n_0 such that $n^3 \leq cn^2$ for all $n \geq n_0$. But this implies, $c \geq n$ for all $n \geq n_0$, contradicting our assumption that c is a constant.

$f_3(n)$ is not $O\big(f_1(n)\big)$ but it is $O\big(f_2(n)\big)$ and $O\big(f_4(n)\big)$. To show $f_3(n)$ is $O\big(f_4(n)\big)$, we can use $n_0 = 3$ and $c = 1$. Remember that every even number except 2 is composite.

$f_4(n)\big)$ is not $O\big(f_1(n)\big)$ but it is $O\big(f_2(n)\big)$ and $O\big(f_3(n)\big)$.

3.4.3: Choose $c = 2$ and $n_0 = 0$. Since $f(n) \leq g(n)$ for $n \geq 0$, we know that $f(n) + g(n) \leq 2g(n)$ for $n \geq 0$. Therefore, $f(n) + g(n)$ is $O\big(g(n)\big)$.

◇ Section 3.5

3.5.1:

a) Choose witnesses $c = 1$ and $n_0 = 1$. Because $a \leq b$, we know $n^a \leq n^b$ for $n \geq n_0$. Thus, n^a is $O(n^b)$.

b) Suppose there exist witnesses $c > 0$ and n_0 such that $n^a \leq cn^b$ for all $n \geq n_0$ when $a > b$. Let d be the larger of n_0 and $c^{1/(a-b)} + 1$. Because of the assumed big-oh relationship, we infer that $d^a \leq cd^b$ or $d^{a-b} \leq c$. But from our choice of d, we know that $d^{a-b} > c$, a contradiction. Thus, we conclude that n^a is not $O(n^b)$ if $a > b$.

3.5.3: Since $T(n)$ is $O\big(f(n)\big)$, we know that there exist witnesses $c > 0$ and $n_0 \geq 0$ such that $T(n) \leq cf(n)$ for all $n \geq n_0$. Since $g(n) \geq 0$ for all $n \geq 0$, we know $g(n)T(n) \leq cf(n)g(n)$, for $n \geq 0$. Thus, $g(n)T(n)$ is $O\big(g(n)f(n)\big)$.

3.5.5: Since $f(n)$ is $O\big(g(n)\big)$, there exist witnesses $c > 0$ and $n_0 \geq 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$. Choose $d = \max(c, 1)$. For any value of n, $\max\big(f(n), g(n)\big)$ is either $f(n)$ or $g(n)$. If $\max\big(f(n), g(n)\big)$ is $f(n)$, we know $f(n) \leq cg(n) \leq dg(n)$. If $\max\big(f(n), g(n)\big)$ is $g(n)$, we know $g(n) \leq dg(n)$. Thus, $\max\big(f(n), g(n)\big)$ is $O\big(g(n)\big)$.

◇ Section 3.6

3.6.1: The body of the for-loop

```
for(i=a; i<=b; i++)
```

is iterated $b - a + 1$ times, or 0 times if $a > b$.

The body of the for-loop

```
for(i=a; i<=b; i--)
```

is iterated $a - b + 1$ times, or 0 times if $b > a$.

3.6.3: If the condition C is false, the running time of the while-loop is $O(1)$. If the condition is true, the while-loop executes forever and the running time is not defined.

3.6.5: The running time is $O\big(g(n)\big)$. That is, the running time is that of the branch taken.

◇ Section 3.7

3.7.1: The tree is shown in Fig. S3.1. The assignment statements at the leaves (2), (4), (5), (6), (7), (10), (11) each take $O(1)$ time. The for-statement, (3)–(4), takes $O(n)$ time. The if-statement, (9)–(10), takes $O(1)$ time and the while-statement, (8)–(11), takes $O(n)$ time. The running time of the entire program represented by

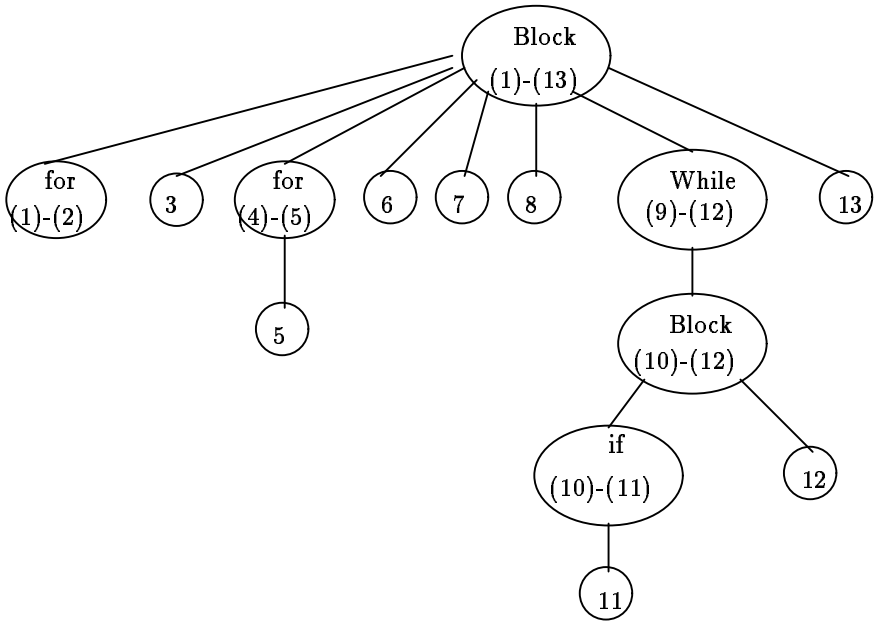


Fig. S3.1. Tree showing grouping of statements of program in Fig. 3.17.

the root is $O(n)$.

3.7.3: The tree is shown in Fig. S3.2. The assignment statements at the leaves (2), (3), (5), and (6) each take $O(1)$ time. As a function of i, the running time of the while-loop is $O(\log i)$ and the running time of the for-loop is $O(n \log n)$. As a function of n, the running time of the while-loop is $O(\log n)$ and the running time of the for-loop is $O(n \log n)$.

◇ Section 3.8

3.8.1: There are several ways to attack this problem. A proof by induction on n can be used to show that

$$\sum_{i=1}^{n}(i + n(n+1)/2) = (n^3 + 2n^2 + n)/2$$

for all $n \geq 0$. Perhaps simpler is to note that the left-hand side can be written as

$$\sum_{i=1}^{n} i + \sum_{i=1}^{n} n(n+1)/2$$

The first term sums to $n(n+1)/2$ as we saw in the introduction to Chapter 2. The expression in the second term is independent of i. The second term is thus $n^2(n+1)/2$. Adding these two sums, we get

$$n(n+1)/2 + n^2(n+1)/2 = (n^3 + 2n^2 + n)/2$$

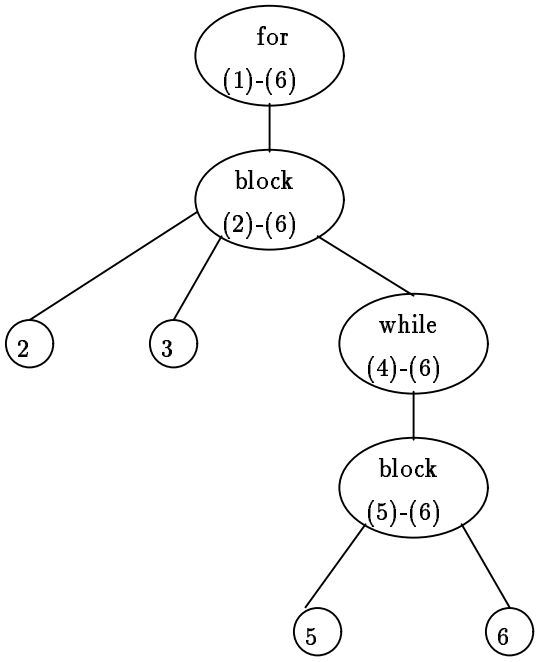


Fig. S3.2. Tree showing grouping of statements of program in Fig. 3.19.

which is the expression on the right-hand side of the equality we wanted to prove.

3.8.3: Each time we go around the loop we evaluate $f(n)$ and increment i. We also initialize i before the loop. Initialization and incrementation of i are each $O(1)$ operations, and we can neglect them. The body of the loop is iterated $f(n)$ times, taking $O(1)$ time per iteration, or $O\big(f(n)\big)$ time total. Thus, the running time of the loop is $O\big(f(n)\big)$ plus the time to evaluate $f(n)$ $f(n)+1$ times. For example, the answer to (a) is $O\big(n \times (n!+1)\big) + O(n!) = O(n \times n!)$.

3.8.5: Note that $bar(n,n) = (n^2+3n)/2$. The function `bar` takes $O(n)$ time as before. Line (8) of procedure `foo` takes $O(n)$ time and the for-loop of lines (7)–(8) is iterated $(n^2+3n)/2$ times. The evaluation of $bar(n,n)$ in the new line (7) takes $O(n)$ time and can be neglected. Thus, procedure `foo` now takes $O(n^3)$ time. The running time of `main` is dominated by the running time of `foo` and thus `main` takes $O(n^3)$ time.

◇ Section 3.9

3.9.1: Let $T(n)$ be the running time of $sum(L)$, where n is the length of the list L. We can define $T(n)$ by the following recurrence relation:

$$T(0) = O(1)$$
$$T(n) = O(1) + T(n-1)$$

Replacing the big-oh's by constants, we get

$T(0) = a$
$T(n) = b + T(n-1)$, for $n \geq 1$

As we saw in this section, the solution to this recurrence is

$T(n) = a + bn$, for $n \geq 0$

The running time of `sum` is therefore $O(n)$.

3.9.3: Let m be the number of elements yet to be sorted. Let $T(m)$ be the running time of `SelectionSort` applied to m elements. We can define the following recurrence for $T(m)$:

$T(1) = O(1)$
$T(m) = O(m) + T(m-1)$, for $m > 1$

Replacing the big-oh's by constants, we get

$T(1) = a$
$T(m) = bm + T(m-1)$, for $m \geq 1$

By repeated substitution, we find the solution to this recurrence is

$T(m) = b(m+2)(m-1)/2 + a$

The running time of `SelectionSort` is therefore $O(m^2)$.

3.9.5:

```
int gcd(int i, int j) {

int r;

    r = i % j;
    if(r != 0) return gcd(j,r);
    else return j;
}
```

For convenience, assume that $i > j$. (Note that this property always holds except possibly for the first invocation of `gcd`.) Let $T(i)$ be the running time of $gcd(i, j)$.

Suppose $gcd(i, j)$ calls $gcd(j, m)$ which calls $gcd(m, n)$. We shall show that $m \leq i/2$. There are two cases. First, if $j \leq i/2$, then $m \leq j \leq i/2$. Second, If $j > i/2$, then $m = i$ `MOD` $j = i - j \leq i/2$.

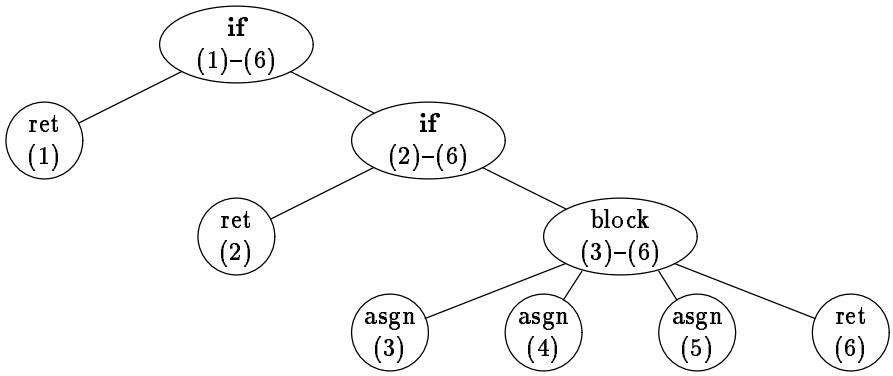
Thus, we conclude that after every two calls to `gcd`, the first argument is reduced by at least half. If we substitute the text of `gcd` for one invocation of the recursive call, we can model the running time of `gcd` by the recurrence

$T(i) \leq O(1) + T(i/2)$

The solution to this recurrence is $O(\log i)$. (See Exercise 3.11.3.)

◇ Section 3.10

3.10.1(a):



As in the text, let $T(n)$ be the time taken by `split` on list of length n. The running time of the assignments (1), (2), (3), (4), and (6) is each $O(1)$. The running time of assignment (5) is $T(n-2)$. The running time of the block-node representing lines (3)–(6) is $O(1) + T(n-2)$, as is the running time of the if-node representing lines (1)–(6).

◇ Section 3.11

3.11.1: We shall prove the following statement by induction on i.

STATEMENT $S(i)$: If $1 \leq i < n$, then

$$T(n) = T(n-i) + \sum_{j=0}^{i-1} g(n-j)$$

BASIS. The basis is $i = 1$. $S(1)$ states that

$$T(n) = T(n-1) + g(n)$$

This is known to be true by the definition of $T(n)$.

INDUCTION. If $i \geq n$, then $S(i+1)$ is true vacuously (the hypothesis of the statement S, that $1 \leq i \leq n$, is false). Assume that $1 \leq i < n$ and that $S(i)$ is true. We wish to prove $S(i+1)$. From the inductive step we know

$$T(n) = T(n-i) + \sum_{j=0}^{i-1} g(n-j)$$

From the definition of $T(n)$ we know

$$T(n-i) = T(n-i-1) + g(n-i)$$

Substituting this equation into the previous, we get

$$T(n) = T(n-i-1) + \sum_{j=0}^{i} g(n-j)$$

which is $S(i+1)$. We have thus proven the inductive step. We conclude $S(i)$ is true for $1 \leq i < n$.

3.11.3: The general form of the solution is

$$T(n) = a + \sum_{j=0}^{\log_2 n - 1} g(n/2^j)$$

a) When $g(n) = n^2$, the sum is $n^2 + n^2/2 + n^2/4 + \cdots + n^2/(n/2)$, which is upper bounded by the infinite geometric sum with first term n^2 and ratio 1/2. This sum is $2n^2$, so $T(n)$ is $O(n^2)$.
c) When $g(n) = 10$, $T(n)$ is $O(\log n)$.
e) When $g(n) = 2^n$, $T(n)$ is $O(2^n)$.

3.11.5: Suppose we guess that there is a constant c such that $G(n) \leq c2^n$, for $n \geq 1$. From the basis, we get the constraint $3 \leq 2c$. From the induction, we get the constraint

$$G(n) = (2^{n/2} + 1)G(n/2) \leq ((2^{n/2} + 1)c2^{n/2} = c2^n + c2^{n/2}$$

If $c2^n + c2^{n/2} \leq c2^n$, then $c \leq 0$, contradicting the basis constraint. Thus, if we guess that $G(n) \leq c2^n$, we fail to find a solution.

3.11.9:
a) Here, $c = 3$, $d = 2$, $k = 2$. Thus, $c < d^k$, so the solution to $T(n)$ is of the form $O(n^2)$.
b) Here, $c = 10$, $d = 3$, $k = 2$. Thus, $c > d^k$, so the solution to $T(n)$ is of the form $O(n^{\log_3 10})$.
c) Here, $c = 16$, $d = 4$, $k = 2$. Thus, $c = d^k$, so the solution to $T(n)$ is of the form $O(n^2 \log n)$.

Chapter 4. Combinatorics and Probability

Section 4.2

4.2.1(a): $4^3 = 64$.

4.2.3: First, note that we can choose input x so that each of the eight conditions is either true or false, as we wish. The reason is that each test asks whether x is divisible by a different prime. We may pick x to be the product of those primes for which we would like the test to be true. It is not possible that this product is divisible by any of the other primes.

Also note that different sets of true conditions lead to different values of n. The reason is that two different products of primes cannot yield the same value of n.

Now, we can compute the answer. We are asked to choose a "color," true or false, for each of eight conditions. We can do so in $2^8 = 256$ ways.

4.2.5: 10^n.

4.2.7: (a) 8K (c) 16M (e) 512P.

Section 4.3

4.3.1(a): $9! = 362880$.

4.3.3: There are at most five comparisons in any branch. This number is best possible, since four comparisons can only distinguish 16 different orders, and there are $4! = 24$ orders.

Given (a, b, c, d) to sort, the list is split into (a, c) and (b, d). The first thing that happens is (a, c) is sorted, resulting in the comparison of a and c. Then, (b, d) is sorted similarly. The third comparison is between the winners of the two comparisons. For example, if a and b are the winners, we compare these two. If, say, a wins, then the fourth comparison is between b and c. If b wins we are done, but if c wins, we need a fifth comparison of b against d. The first three levels of the decision tree are shown in Fig. S4.1.

Section 4.4

4.4.1(a): $26!/(26-3)! = 26!/23! = 26 \times 25 \times 24 = 15600$.

4.4.4:

a) This is a selection without replacement; there are $6^4 = 1296$ codes.

b) The number of codes without repetitions of color is $6!/(6-4)! = 6!/2! = 360$. Thus, the number of codes with a repetition is $1296 - 360 = 936$.

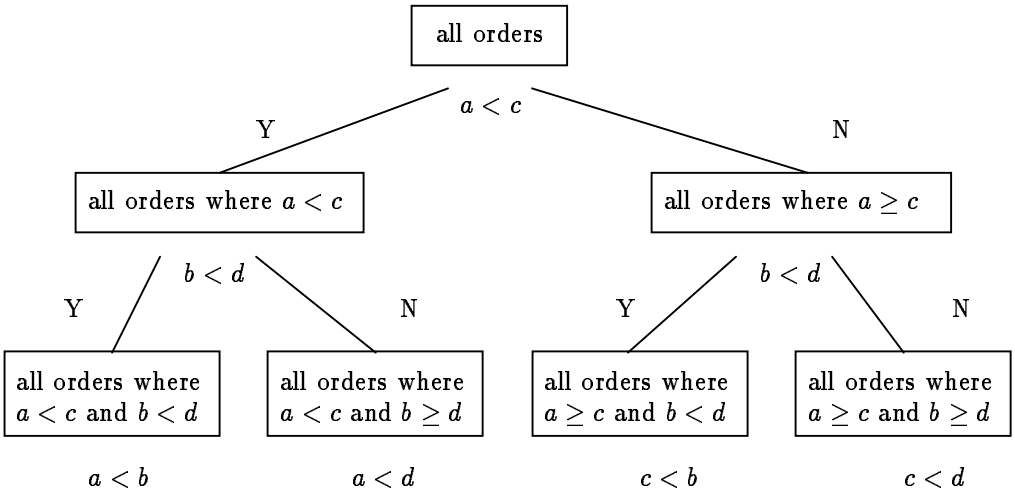


Fig. S4.1. Decision tree for Exercise 4.3.3..

c) $5^4 = 625$.

d) If there are only five colors, then the number of codes without repetition is $5!/(5-4)! = 5!/1! = 120$. Thus, the number with a repetition but no red peg is $625 - 120 = 505$.

◇ Section 4.5

4.5.1(a): $7!/\big(3! \times (7-3)!\big) = 7!/(3! \times 4!) = 5040/(6 \times 24) = 35$.

4.5.3(a): $\binom{7}{3}$, which, as we learned in Exercise 4.5.1(a), is 35.

4.5.5: To begin, we need to pick the positions that are vowels. There are $\binom{5}{2} = 10$ ways to do so. For each of these 10 ways, we can pick the 3 consonant positions in $21^3 = 9261$ ways. We can pick the two vowel positions in $5^2 = 25$ ways. Thus, the total number of words of length five with vowels in two particular positions is $9261 \times 25 = 231525$ ways. The number of words altogether is ten times this, or 2,315,250.

4.5.7:

```
c = 1.0;
for(i=n; i>n-m; i--) {
    c *= i;
    c /= (i-n+m);
}
```

◇ Section 4.6

4.6.1(a): $5!/(3! \times 1! \times 1!) = 120/(6 \times 1 \times 1) = 20$.

4.6.3: One way to look at this problem is that we wish to order 64 items, of which 3 are unique (the squares with pieces other than knights), two are indistinguishable from each other (the squares with the white knights) and the remaining 59 (the squares that do not have a piece on them) are also indistinguishable from each other. The number of orders is thus $64!/(59! \times 2! \times 1! \times 1! \times 1!) = 64!/(59! \times 2!) = 457{,}470{,}720$.

4.6.5: $(2n)!/(2! \times 2! \times \cdots \times 2!)$ (n times), or $(2n)!/2^n$.

◇ Section 4.7

4.7.1:

a) $(6+3)!/(6! \times 3!) = 9!/(6! \times 3!) = 84$.

c) $(6+3+4)!/(6! \times 3! \times 4!) = 13!/(6! \times 3! \times 4!) = 60060$.

4.7.3: Let us reserve one apple for each of the three children. Then, we may distribute the remaining four apples as we like. There are $(4+2)!/(4! \times 2!) = 15$ ways to do so.

◇ Section 4.8

4.8.1(a): Begin by picking the card that is not part of the two pairs. We can do so in 52 ways. Now, pick the ranks of the two pairs. There are only 12 remaining ranks, so we can do so in $\binom{12}{2} = 66$ ways. For each of the pairs, we can pick the two suits in $\binom{4}{2}$, or 6, ways. We now have the 5 cards of the hand without order, and the number of possibilities is $52 \times 66 \times 6 \times 6 = 123{,}552$.

4.8.2(a): We may pick the Ace four different ways, and we may pick the 10-point card in 16 ways. Thus, there are $4 \times 16 = 64$ different blackjacks.

4.8.4(a): $\binom{12}{9} + \binom{12}{10} + \binom{12}{11} + \binom{12}{12} = 220 + 66 + 12 + 1 = 299$.

4.8.7(a): First, we must pick the suit of which there is four. We can do so in 4 ways. Now, we pick the cards of the suit of four; there are $\binom{13}{4} = 715$ ways to do so. For each of the suits of three, we can select the cards in $\binom{13}{3} = 286$ ways. Thus, the number of hands is $4 \times 715 \times 286 \times 286 \times 286 = 66{,}905{,}856{,}160$ ways.

◇ Section 4.9

4.9.1(a): 5/36.

4.9.2(a): First, there are $52 \times 51 = 2652$ members of the probability space. To calculate the number of points with one or more Aces, it is easier to calculate the number with no Ace and subtract this probability from 1. The number of deals of two cards from the 48 that remain after removing the Aces is $48 \times 47 = 2256$. Thus, the probability of no Ace is 2256/2652, and the probability of at least one Ace is $1 - (2256/2652) = 396/2652 = 14.9\%$.

4.9.3(a): The area of a circle of radius 3 inches is $9\pi = 28.27$. The area of the entire square is 144 square inches. Thus, the probability of hitting the circle is $28.27/144 = 19.6\%$.

4.9.4(a): The probability is $\binom{5}{4} \times \binom{75}{16} / \binom{80}{20}$. If we cancel common factors in numerator and denominator, we get $5 \times 20 \times 19 \times 18 \times 17 \times 60 / (80 \times 79 \times 78 \times 77 \times 76)$, or 1.21%.

◇ Section 4.10

4.10.1:

a) Among the 18 points with an odd first die, 9 have an even second die. Thus, the probability is 9/18, or 50%.

c) There are six points that have 4 as the first die. Of these, four have a sum at least 7. Thus, the probability is 4/6, or 66.7%.

4.10.2(a): In the region of 120 points where there are three different numbers, the probability of two 1's is 0. In the region of 90 points with two of one number, 1/6 will have two 1's. In the region of six points with all three dice the same, the probability of at least two 1's is 1/6. Thus, the probability of at least two 1's is $0 \times (120/216) + (1/6) \times (90/216) + (1/6)(6/216) = 7.41\%$.

4.10.7: An appropriate probability space, in which all points are of equal probability, is one with six points, two for each of the choices of which prisoner to shoot. The distinction between the two points for prisoner A is the order in which the guard will consider the other two prisoners when asked for a prisoner (other than the questioner) who will not be shot. Thus, the two points for A can be thought of as (A, "B-before-C") and (A, C-before-B). The remaining four points, for B and C, are specified similarly.

Now, suppose that the guard answers "B" to A's question. There are three points that could have occurred:

(A, B-before-C), (C, A-before-B), and (C, B-before-A)

In only the first of these will A be shot, so the probability is still 1/3.

◇ Section 4.11

4.11.1(a): The probability that at least one of the events is at least the largest of the p_i's. The probability would be exactly $\max(p_1, p_2, \ldots, p_n)$ in the case that all the events were contained within the largest. The probability of at least one of the events is no greater than the sum of the p_i's, and of course it is no greater than 1. The probbility $\sum_{i=1}^{n} p_i$ would be reached in the case that all the events were disjoint.

4.12.2:

a) Nothing. It could be anything between 0 and 1.

b) The probability is $1 - p$.

4.12.3:

a) Between 0 and 0.3.

c) Cold must be contained in both High and Dropping. Thus, Cold cannot have probability greater than the smaller of High and Dropping, that is, 0.3. Its probability could be as low as 0, however, for example, if High and Dropping were disjoint.

◇ Section 4.12

4.12.1: Intuitively, the expected number of 1's on one die is 1/6. The tosses of dice are independent, so we expect 1/6 of a 1 from each. In three dice, we thus expect 1/2 a 1.

Alternatively, consider the $6^3 = 216$ tosses of three dice. The number of tosses with three 1's is 1. The number of tosses with two 1's is 15, since the other die can be any of five numbers, and the non-1 die can appear in any of three positions. The number of tosses with exactly one 1 is 75. In explanation, the 1 can appear in any of 3 positions. For each of the other two positions, there are 5 choices, for a total of $3 \times 5 \times 5 = 75$ tosses. The expected number of 1's is thus $(1 \times 3 + 15 \times 2 + 75 \times 1)/216 = (3 + 30 + 75)/216 = 108/216 = 1/2$.

4.12.2: Exercise 4.12.1 suggests that the average amount of our winnings is 50 cents. However, the average amount we lose is not 50 cents. It is one dollar times the probability that we lose. This probability is 125/216, the fraction of tosses that do not contain a 1. This expected value is -57.9 cents, so the expected value of our payout is -7.9 cents.

4.12.5: The game is fair. Generalizing Exercise 4.12.1, we have six independent tosses, each with an expected 1/6 of a 1, so the expected amount recieved in each toss is one dollar. Since we pay a dollar to play, our net expected payout is 0.

◇ Section 4.13

4.13.1: 383 is prime. $377 = 29 \times 13$, and $391 = 23 * 17$.

4.13.3: The number of tickets is $1 + 2 + \cdots + 11 = 66$. Of these tickets, 11 have a value of 1; i.e., the finishing place of the holder of those tickets is 1, counting from the bottom. 10 tickets have a value of 2, 9 have value 3, and so on. Thus, the expected value is $\sum_{i=1}^{11} i(12 - i)/66$. We can calculate this sum with the help of the formulas in Exercise 2.3.1(a) and (b). It is $286/66 = 4.33$. That is, the first pick goes to a team that is on average a little better than fourth-worst.

Chapter 5. The Tree Data Model

Section 5.2

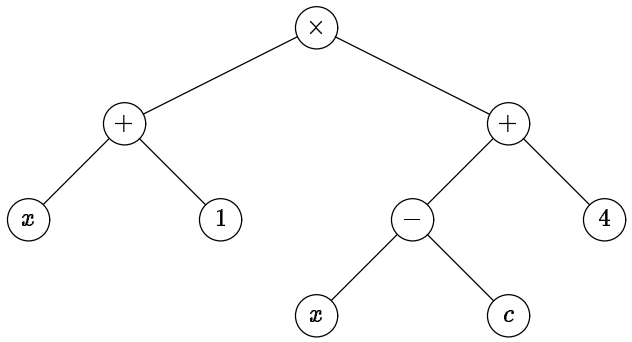
5.2.1: For the tree of Fig 5.5:

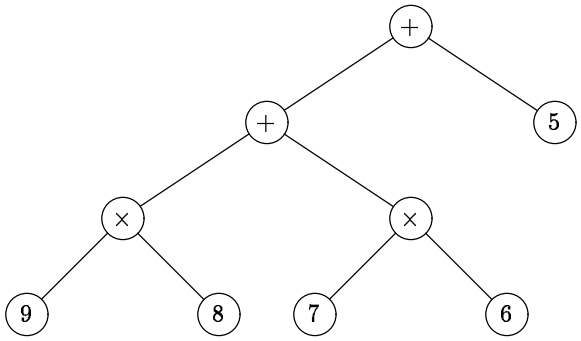
a) Node 1 is the root of the tree.
b) Nodes 6, 8, 9, 13, 15, 11, and 12 are the leaves.
c) Nodes 1–5, 7, 10, and 14 are the interior nodes.
d) Nodes 5 and 7 are the siblings of node 6.
e) The tree consisting of nodes 5, 10, 13, 14, and 15 is the subtree with root 5.
f) Nodes 1, 3, 5, and 10 are the ancestors of node 10.
g) Nodes 10, 13, 14, and 15 are the descendants of node 10.
h) Nodes 2, 4, 8, and 9 are to the left of node 10.
i) Nodes 6, 7, 11, and 12 are to the right of node 10.
j) Nodes 1, 3, 5, 10, 14, and 15 form the longest path (of length 5) in the tree.
k) Node 3 is of height 4.
l) Node 13 is the depth 4.
m) The tree is of height 5.

5.2.3: Let x and y be distinct leaves in a tree, and suppose x is an ancestor y. Then there exists a path $x = n_1, n_2, \ldots, n_k = y$ of length one or more from x to y. Thus, n_1 is a child of x, and x cannot be a leaf.

5.2.5: There is no edge connecting r to any of a, b, or c. If r is the root, then there is no way of reaching r from a, b, or c. Similarly, if one of a, b, or c is the root, then there is no way of reaching the root from r. Thus, the third property of the definition of a tree (a tree must be connected) is violated.

5.2.7(a): The expression tree for $(x + 1) \times (x - y + 4)$ is





NODE	LEFTMOST CHILD	RIGHT SIBLING
1	2	-
2	4	3
3	5	-
4	8	-
5	10	6
6	-	7
7	11	-
8	-	9
9	-	-
10	13	-
11	-	12
12	-	-
13	-	14
14	15	-
15	-	-

Fig. S5.1. Answer to Exercise 5.3.1..

5.2.7(c): The expression tree for $9 \times 8 + 7 \times 6 + 5$ is

◇ Section 5.3

5.3.1: For each node, the leftmost child and right sibling are shown in Fig. S5.1.

5.3.5: There are 10^7 nodes, each with 4 bytes of information and two 4-byte pointers, or 1.2×10^8 bytes. There would be $10^7 + 1$ `NULL` pointers (see Exercise 5.5.5).

◇ Section 5.4

5.4.1: Here is a surprisingly simple function that counts the nodes in a tree.

```
int count(pNODE n) {
    if(n != NULL)
        return(count(n->rightSibling) +
                count(n->leftmostChild) + 1);
    else return 0;
}
```

At first glance, the function doesn't seem to address the problem. However, the "inductive assertion" about `count` is that for any node n, $count(n)$ is the sum of the number of nodes in the subtree rooted at n *and all those subtrees rooted at siblings of* n *to the right of* n. The induction is straightforward, once we realize that the induction is on the length of the longest path of leftmost-child and right-sibling pointers extending from a node. Since the root has no siblings, the desired result appears at the root.

Another way to look at the `count` function above is that the leftmost-child and right-sibling pointers turn the tree into a binary tree with the same number of nodes. Surely, the rule that the number of nodes in a binary tree rooted at n is 1 plus the sum of the number of nodes in the left and right subtrees makes sense.

Incidentally, this technique applies to any computation on trees that can be expressed as an associative operator applied to the results of the children (the operator is + in the case of `count`), and a final, unary operator (add-one in this case). For example, the function to compute the height of a tree in Fig. 5.22 of the text can be replaced by a simpler (but less transparent) function that computes the height of n to be the larger of the height of the right sibling of n and 1 plus the height of the leftmost child of n. The height will be correct at the root, but in general, $height(n)$ is the largest of the height of n and any siblings of n to the right.

5.4.5: The node listings are

a) Preorder: 1, 2, 4, 8, 9, 3, 5, 10, 13, 14, 15, 6, 7, 11, 12
b) Postorder: 8, 9, 4, 2, 13, 15, 14, 10, 5, 6, 11, 12, 7, 3, 1

5.4.7: First, construct the expression tree. Then the infix and prefix expressions can be read off the tree.

a) Infix expression is $\big((a+b)*c\big)/(d-e)+f$.
b) Prefix expression is $+/*+abc-def$.

◇ Section 5.5

5.5.1(a): We shall prove by structural induction

STATEMENT $S(T)$: The procedure `preorder` when called on the root of T prints the labels of the nodes of T in preorder.

BASIS. When the tree T is a single node n, line (1) prints the label of the root n. At line (2) c is set to NULL, and consequently the body of the while-loop is not executed.

INDUCTION. Suppose we execute `preorder` on a tree T with root n and children $c_1, c_2, \ldots, c_k$. Line (1) prints the label of node n. Line (2) sets `c` to c_1. By the inductive hypothesis while-loop proceeds to print the labels of the subtrees rooted at $c_1, c_2, \ldots, c_k$ in preorder. This is the same as the definition of a preorder listing. We conclude that `preorder` prints the labels of a tree in preorder.

5.5.3: We shall prove by structural induction

STATEMENT $S(T)$: The number of nodes in T is 1 more than the sum of the degrees of the nodes.

BASIS. When T is a single node n, the degree of n is 0.

INDUCTION. Suppose n, the root of a tree T, has nodes $c_1, c_2, \ldots, c_k$ as children. Let $numnodes(c_i)$ be the number of nodes in the subtree rooted at c_i. Let $degree(c_i)$ be the sum of the degrees of the nodes in the subtree rooted at c_i. By the inductive hypothesis, we know that $numnodes(c_i) = degree(c_i) + 1$ for $1 \leq i \leq k$. The total number of nodes in T is $1 + \sum_{i=1}^{k} numnodes(c_i)$. The sum of the degrees of all the nodes in T is $\sum_{i=1}^{k} degree(c_i) + k$. We therefore have

$$1 + \sum_{i=1}^{k} numnodes(c_i) = 1 + \sum_{i=1}^{k} degree(c_i) + k$$

Since the root has degree k, the latter sum is 1 plus the sum of all the nodes in T, proving the induction.

5.5.5:

BASIS. A leaf has 2 `NULL` pointers and 1 node.

INDUCTION. Let T be a tree with root r. Let r have children $c_1, \ldots, c_k$, the roots of subtrees $T_1, \ldots, T_k$, respectively. Let T_i, as a tree by itself, have p_i `NULL` pointers and n_i nodes. By the inductive hypothesis, $p_i = n_i + 1$ for $i = 1, 2, \ldots, k$.

When we assemble T from r and the T_i's, we replace the `NULL` pointers in the `rightSibling` fields of $c_1, c_2, \ldots, c_{k-1}$ by non-`NULL` pointers. The root r has a non-`NULL` `leftmostChild` field and a `NULL` `rightSibling` field. The number of `NULL` pointers in T is thus $(\sum_{i=1}^{k} p_i) - (k-1) + 1$. Since $p_i = n_i + 1$ by the inductive hypothesis, the number of `NULL` pointers is $(\sum_{i=1}^{k} n_i) + 2$. This is one greater than the number of nodes in T, which is $(\sum_{i=1}^{k} n_i) + 1$.

◇ Section 5.6

5.6.1:

```
void inorder(TREE t) {
    if(t != NULL) {
        inorder(t->leftChild);
        printf("%d ", t->nodelabel);
        inorder(t->rightChild);
    }
}
```

5.6.3: In the code of Fig. S5.2, we assume the function `pr(x)` returns the precedence associated with the node label `x`. We assume leaf-operands have the highest precedence so no parentheses are put around them. When the left operand of the root t has lower precedence than the root, we put parentheses around the left operand, and the right operand is treated similarly.

```
void pinorder(TREE t) {
    if(t != NULL) {
        if(t->leftChild != NULL) {
            if(pr(t->leftChild.nodelabel) <= pr(t->nodelabel))
                pinorder(t->leftChild);
            else {
                printf("(");
                pinorder(t->leftChild);
                printf(")");
            }
        }
        printf("%d ", t->nodelabel);
        if(t->rightChild != NULL) {
            if(pr(t->rightChild.nodelabel) <= pr(t->nodelabel))
                pinorder(t->rightChild);
            else {
                printf("(");
                pinorder(t->rightChild);
                printf(")");
            }
        }
    }
}
```

Fig. S5.2. Solution to Exercise 5.6.3..

◇ Section 5.7

5.7.2: The trees are shown in Fig. S5.3.

5.7.4:

```
TREE insert(ETYPE x, TREE* pT)
{
    if ((*pT) == NULL) {
        (*pT) = (TREE) malloc(sizeof(struct NODE));
        (*pT)->element = x;
        (*pT)->leftChild = NULL;
        (*pT)->rightChild = NULL;
    }
    else if (x < (*pT)->element)
        (*pT)->leftChild = insert(x, &((*pT)->leftChild));
    else if (x > (*pT)->element)
        (*pT)->rightChild = insert(x, &((*pT)->rightChild));
}
```

◇ Section 5.8

5.8.1: The branching factor is the maximum number of children a node can have. The smallest tree of height h with branching factor b is a simple path of $h+1$ nodes. The largest tree of height h with branching factor $b > 1$ is a complete b-ary tree (all nodes on the first h levels have b children). There is one root, b children of the root, b^2 children of those, and so on to depth h. The total number of nodes is $\sum_{i=0}^{h} b^i$, or $(b^{h+1} - 1)/(b - 1)$ nodes.

◇ Section 5.9

5.9.1:

a) Sequence of steps to insert 3:

18 18 16 9 7 1 9 3 7 5 3	initially 3 goes into `A[11]`
18 18 16 9 7 1 9 3 7 5 3	after `bubbleUp(A,11)` (no change to `A`)

b) Insert 20:

18 18 16 9 7 1 9 3 7 5 3 20	initially 20 goes into `A[20]`
20 18 18 9 7 16 9 3 7 5 3 1	after `bubbleUp(A,12)`

c) Delete maximum element (replacing it by `A[12]`):

1 18 18 9 7 16 9 3 7 5 3	initial array
18 9 18 7 7 16 9 3 1 5 3	after calling `deletemax(A,11)`

d) Again, delete maximum element (replacing it by `A[11]`):

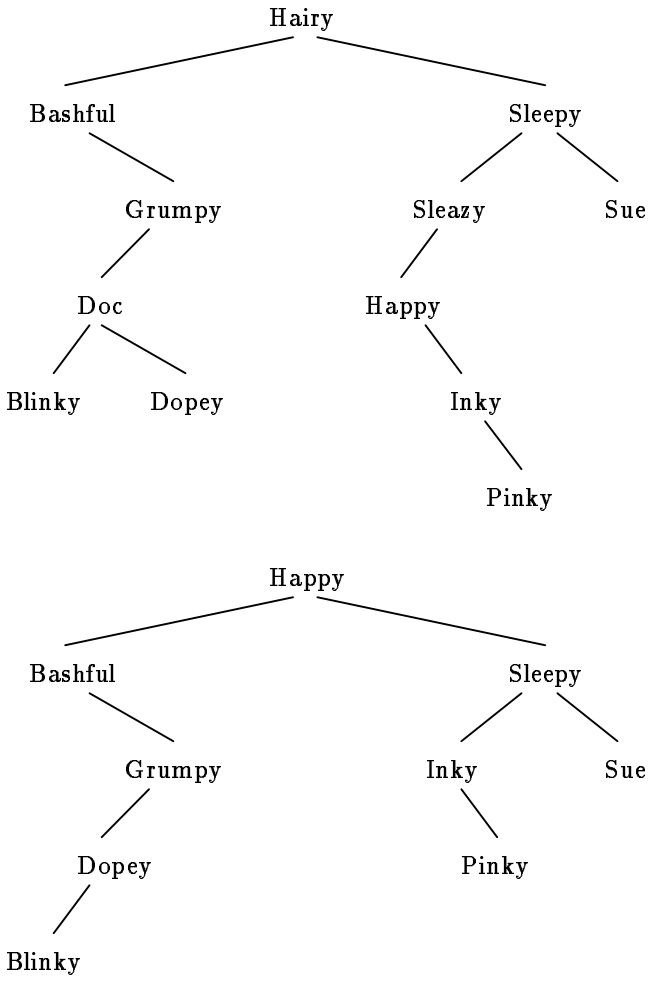


Fig. S5.3. Solutions to Exercise 5.7.2..

3 9 18 7 7 16 9 3 1 5	initial array
18 9 16 7 7 3 9 3 1 5	after calling `deletemax(A,10)`

5.9.7: Clearly, `bubbleDown` takes $O(1)$ time plus the time of the recursive call. The second argument of the recursive call is at least twice the value of the second formal parameter `i`. When `i` exceeds $n/2$, there is no recursive call made. Thus, no more than $\log_2 n$ recursive calls can result from an initial call to `bubbleDown`. Hence, the total time is $O(\log n)$.

◇ Section 5.10

5.10.1: Here is the sequence of steps made by `heapsort`:

3 1 4 1 5 9 2 6 5	initial array `A`
3 1 4 6 5 9 2 1 5	after `bubbleDown(A,4,9)`
3 1 9 6 5 4 2 1 5	after `bubbleDown(A,3,9)`
3 6 9 5 5 4 2 1 1	after `bubbleDown(A,2,9)`
9 6 4 5 5 3 2 1 1	after `bubbleDown(A,1,9)`
6 5 4 1 5 3 2 1 9	after `deletemax(A,9)`
5 5 4 1 1 3 2 6 9	after `deletemax(A,8)`
5 2 4 1 1 3 5 6 9	after `deletemax(A,7)`
4 2 3 1 1 5 5 6 9	after `deletemax(A,6)`
3 2 1 1 4 5 5 6 9	after `deletemax(A,5)`
2 1 1 3 4 5 5 6 9	after `deletemax(A,4)`
1 1 2 3 4 5 5 6 9	after `deletemax(A,3)`
1 1 2 3 4 5 5 6 9	after `deletemax(A,2)`

Chapter 6. The List Data Model

Section 6.2

6.2.1:
a) Length is 5.
b) Prefixes are ϵ, (2), (2,7), (2,7,1), (2,7,1,8), (2,7,1,8,2).
c) Suffixes are ϵ, (2), (8,2), (1,8,2), (7,1,8,2), (2,7,1,8,2).
d) Sublists are ϵ, (2), (7), (1), (8), (2), (2,7), (7,1), (1,8), (8,2), (2,7,1), (7,1,8), (1,8,2), (2,7,1,8), (7,1,8,2), (2,7,1,8,2).
e) There are 31 distinct subsequences.
f) The first 2 is the head.
g) The list (7,1,8,2) is the tail.
h) There are five positions.

6.2.3: Prefixes: There are always exactly $n+1$ prefixes, one each of the lengths 0 through n.

Sublists: First, suppose that all the positions of a string of length n how different symbols. Then there is one sublist of length 0, n different sublists of length 1, $n-1$ different sublists of length 2, $n-2$ of length 3, and so on, for a total of $n(n+1)/2+1$. This is the maximum possible number. The minimum occurs when all the positions hold the same symbol. Then, all sublists of the same length are the same, and there are only $n+1$ different sublists.

Subsequences: Suppose all symbols are distinct. Then every set of the n positions yields a distinct subsequence, so there are 2^n subsequences. That is the maximum number. If all positions hold the same symbol, then all subsequences of the same length are the same, and we have $n+1$ subsequences, the minimum possible number.

6.2.5: 1,2,3 can represent an infinite number of different kinds of lists of lists including ((1),(2),(3)), ((1,2),(3)), ((1),(2,3)), ((1,2,3)), (((1,2,3))), ((((1,2,3)))), and so on.

Section 6.3

6.3.1:
a) $delete(5, L)$ = (3,1,4,1,9)
b) $delete(1, L)$ = (3,4,1,5,9) or (3,1,4,5,9)
c) $pop(L)$ removes 3 from L leaving (1,4,1,5,9)
d) $push(2, L)$ adds 2 to the beginning of L giving (2,3,1,4,1,5,9)
e) $lookup(6, L)$ returns FALSE.
f) LM = (3,1,4,1,5,9,6,7,8)
g) $first(L) = 3$; $last(L) = 9$
h) $retrieve(3, L) = 4$, the element at position 3
i) $length(L) = 5$

j) $isEmpty(L) =$ `FALSE`

6.3.3:

a) One condition under which $delete(x, insert(x, L)) = L$ is true would be if $insert(x, L)$ always added x to the beginning of L and $delete(x, L)$ removed the first occurrence of x from L.

c) $first(L)$ is always equal to $retrieve(1, L)$.

◇ Section 6.4

6.4.1:

a) Let $T(n)$ be the running time of $delete(x, L)$ where n is the length of list L. The recurrence for $T(n)$ is

$T(0) = a$
$T(n) = b + T(n-1),\ n > 0$

The solution to this recurrence is $T(n) = a + bn$.

6.4.3: Here is a program that inserts an element x into a sorted list L.

```
void insert(ETYPE x, LIST* pL) {

LIST M;

    if((*pL) == NULL) {
        (*pL) = (LIST) malloc(sizeof(struct CELL));
        (*pL)->element = x;
        (*pL)->next = NULL;
    }
    else if(x > (*pL)->element)
        insert(x, &((*pL)->next));
    else {/* insert x between cell holding pointer pL and
            the cell pointed to by *pL */
        M = (LIST) malloc(sizeof(struct CELL));
        M->element = x;
        M->next = *pL;
        (*pL) = M;
    }
}
```

6.4.5: Let p be the pointer to the cell to be deleted.

```
void delete(LIST p) {
    if(p->next != NULL)
        p->next->previous = p->previous;
    p->previous->next = p->next;
}
```

◇ Section 6.5

6.5.1(b): The following procedure deletes element x from list L using linear search to locate x.

```
void delete(ETYPE x, LIST* pL) {

    int i, j;

    i = 1;
    while(i < pL->length && x != pL->A[i]) i++
    if(i <= pL->length && x = pL->A[i]) {
        for(j = i; j < pL->length; j++) {/* shift following
                elements forward */
            pL->A[j] = pL->A[j+1];
        (pL->length)--;
    }
}
```

6.5.3(a): The following function inserts x on L if there is room on L; otherwise, it returns `FALSE`.

```
Boolean insert(ETYPE x, LIST* pL) {

    int i;

    if(pL->length >= MAXLENGTH)
        return(FALSE);
    else {
        i = 1;
        while(i <= pL->length {
            if(x < pL->A[i]) i++;
            else break;
        }
        /* here we have found position i, where x belongs */
        (pL->length)++;
        for(j=pL->length; j>i; j--) /* shift following
                elements back one position in the array */
            pL->A[j] = pL->A[j-1];
        pL->A[i] = x;
        return(TRUE);
    }
}
```

6.5.6: We shall prove the following statement by induction on $d = high - low$.

STATEMENT $S(d)$: Let $d = high - low$. If x is in the range `A[low..high]`, then the algorithm of Fig. 6.14 finds x.

BASIS. $d = 0$. If $high = low$, and if x is in `A[low..high]`, then

$$mid = \lfloor (low + high)/2 \rfloor = low$$

In this case, line (8) of Fig. 6.14 correctly returns `TRUE` since $x = A[mid]$.

INDUCTION. Suppose that $d \geq 0$ and that $S(d)$ is true. We shall prove $S(d+1)$. Suppose that x is in `A[low..high]` where $high - low = d + 1$. Since $high > low$, the block consisting of lines (3)–(8) in Fig. 6.14 is executed. Line (3) computes $mid = \lfloor (low + high)/2 \rfloor$. There are three cases to consider.

Case 1. If $x < A[mid]$, then x is in `A[low..mid-1]`. Then, by the inductive hypothesis, the call to $binsearch(x, L, low, mid - 1)$ on line (5) finds x since

$$mid - 1 - low = d$$

Case 2. If $x > A[mid]$, then the call to $binsearch(x, L, mid + 1, high)$ on line (7) finds x.

Case 3. If $x = A[mid]$, line (8) finds x and returns `TRUE`.

We have now proven the inductive hypothesis and conclude that $S(d)$ is true for all $d \geq 0$.

◇ Section 6.6

6.6.1: The following table shows the contents of the stack after each operation. The top of the stack is on the right.

STACK	ACTION
ϵ	
a	$push(a)$
ab	$push(b)$
a	pop
ac	$push(c)$
acd	$push(d)$
ac	pop
ace	$push(e)$
ac	pop
a	pop

6.6.3: Let us assume that we have a well-formed prefix expression containing numbers and binary operators. The following algorithm evaluates the expression.

Step 1: Push the numbers and operators from the expression (from left to right) on to the stack until the top three symbols on the stack are a binary operator θ, a number a, and a number b (b is on top).

Step 2: Replace θab on top of the stack by the result of applying the operator θ to a and b.

Step 3: Repeat steps (1) and (2) until no more numbers or operators remain in the prefix expression.

The number remaining on top of the stack is the answer.

6.6.5:

a) The function for an array-based stack of integers is:

```
int top(STACK* pS) {
    return(pS->A[ps->top]);
}
```

b) For a list-based stack of integers we can write:

```
int top(STACK S) {
    return(S->element);
}
```

Both implementations take $O(1)$ time.

◇ Section 6.7

6.7.1: The first column in Fig. S6.1 shows the stack of activation records after we have pushed an activation record for `sum`. The remaining columns show the activation records just before we pop each activation record for `sum` off the stack. We use `ret` to name the return value.

◇ Section 6.8

6.8.1: Below is the contents of the queue after each command. The front of the queue is at the left.

QUEUE	ACTION
ϵ	
a	$enqueue(a)$
ab	$enqueue(b)$
b	$dequeue$
bc	$enqueue(c)$
bcd	$enqueue(d)$
cd	$dequeue$
cde	$enqueue(e)$
de	$dequeue$
e	$dequeue$

◇ Section 6.9

6.9.1:

a) 4. `aana` is one.

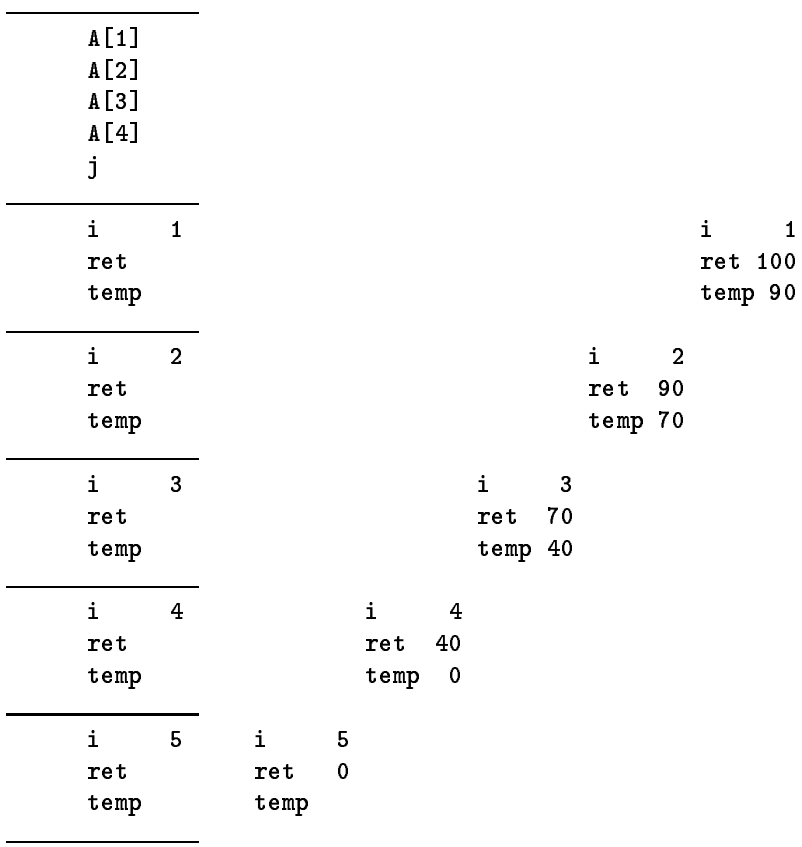


Fig. S6.1. Stack of activation records.

b) 7. `bacbcab` is one.

6.9.3: There are 20 calls to $L(1,1)$. Let $C(i,j)$ be the number of calls to $L(i,j)$ when the strings of length i and j have no symbols in common. The definition of the recursive algorithm tells us that

$C(i,j) = C(i-1,j) + C(i,j-1)$ whenever $i > 0$ and $j > 0$
$C(1,1) = 1$
$C(i,0) = 0$ for all i
$C(0,j) = 0$ for all j

From these observations it follows that $C(i,1) = 1$ for all $i \geq 1$ and $C(1,j) = 1$ for all $j \geq 1$. Thus, a simple induction on $i+j$ shows that $C(i,j) = \binom{i+j-2}{i-1}$ for all $i \geq 1$ and $j \geq 1$ (note $\binom{0}{0=1}$). Thus, $L(4,4)$ calls $L(1,1)$ $\binom{6}{3} = 20$ times.

♢ Section 6.10

6.10.3: Let n be the maximum string length and c the number of characters per cell. Assume both n and c are "large." There are two sources of waste space: the space used by pointers and the unused character space in the last cell. The average number of cells will be about $n/2c$, because the average word is about $n/2$ characters long and is packed c to a cell. There is one pointer of 4 bytes in each cell, so the number of waste bytes due to pointers is $2n/c$. The number of waste bytes in the last cell is $c/2$ on the average. We must thus find the value of c that minimizes

$$\frac{2n}{c} + \frac{c}{2}$$

If you know calculus, you know that the minimum occurs when both terms are equal; that is, $c^2 = 4n$, or $c = 2\sqrt{n}$.

6.10.5: We are in effect replacing a single byte by a 4-byte integer, which costs us 3 bytes per word stored. If integers can be stored in one byte, then it is a wash; the costs are the same.

Chapter 7. The Set Data Model

Section 7.2

7.2.1: The set $\{\{a,b\},\{a\},\{b,c\}\}$ contains three members $\{a,b\},\{a\},\{b,c\}$, each of which is also a set.

7.2.3(a): One representation is $\{b,c,a\}$. Another, using abstraction, is

$$\{x \mid x \text{ is a letter, } a \leq x, \text{ and } x \leq c$$

Section 7.3

7.3.1(a): The simplest expression for region 6 is $S \cap T \cap R$. Region 6 can also be represented as S without regions 2, 3, or 5, that is, as

$$S - (((S - T) - R) \cup ((S \cap T) - R) \cup ((S \cap R) - T))$$

7.3.1(b): One expression for regions 2 and 4 is $(S - (T \cup R)) \cup (T - (S \cup R))$. Another is $(S \cup T) - (S \cap T) - (S \cap R) - (T \cap R)$.

7.3.1(c): Two expressions for regions 2, 4, and 8 together are

$$(S - (T \cup R)) \cup (T - (R \cup S)) \cup (R - (S \cup T))$$

and

$$(S \cup T \cup R) - (S \cap T) - (S \cap R) - (T \cap R)$$

7.3.3(a): First, we show the forward containment $(S \cup (T \cap R)) \subseteq ((S \cup T) \cap (S \cup R))$. Let x be in $S \cup (T \cap R)$. By the definition of union, x is either in S or in $T \cap R$ (or in both). If $x \epsilon S$, then x is a member of the right-hand side of the forward containment. If $x \epsilon (T \cap R)$, then x is in both T and R, and again a member of the right-hand side. Thus, the forward containment holds.

Now, we show the reverse containment $(S \cup (T \cap R)) \supseteq ((S \cup T) \cap (S \cup R))$. Suppose x is in $(S \cup T) \cap (S \cup R)$. By the definition of intersection, x is a member of the left-hand side of the reverse containment. If x is in both T and R, then x is again a member of the left-hand side. We conclude the reverse containment holds.

Since both the forward and reverse containments hold, we conclude

$$(S \cup (T \cap R)) \equiv ((S \cup T) \cap (S \cup R)).$$

7.3.5: A Venn diagram with n sets divides the plane into 2^n regions, assuming no set is a subset of another. We shall prove this by induction on n.

BASIS. If $n = 1$, the Venn diagram has two regions, outside the set and inside the set.

INDUCTION. Suppose that the plane has been divided into 2^n regions with n sets. Now consider adding an $n+$ 1st set. The new set partitions each of 2^n existing regions into two, containing those points within the new set and those without. (Here is where the property that no set is a subset of another is used.) Therefore, $n+1$ sets partition the plane into $2 \times 2^n = 2^{n+1}$ regions. Thus, the inductive step is proved.

We conclude a Venn diagram with n sets divides the plane into 2^n regions for all $n \geq 1$.

If there are $n \geq 2$ sets such that exactly one set is a subset of another, then a Venn diagram for the n sets would have $2^{n-1} + 2^{n-2}$ nonempty regions. If S and T are two of the n sets and $S \subseteq T$, then every member within S must be contained with T. T can partition each of the 2^{n-2} regions formed by the other sets in two, but the 2^{n-2} new regions formed by S must all be partitions of regions contained with Y.

7.3.7: We shall represent a set of elements (integers) by a linked list of `SCELL`s defined in the usual way by our `DefCell` macro:

```
DefCell(int, SCELL, SLIST);
```

We shall represent the powerset whose elements are sets by a linked list of `PCELL`s defined by:

```
DefCell(SLIST, PCELL, PLIST);
```

To save space, we shall reuse previously constructed `SLIST`s wherever possible in the construction of a new member of the powerset.

The function *powerset*(S) takes a linked list S of type `SLIST` and products as output a `PLIST` that is a compact representation for the powerset of S. The function *powerset*(S) uses the following recursive construction:

$\mathbf{P}(\emptyset) = \{\emptyset\}$

Let $S = \{a_1, a_2, ..., a_n\}$ be a set of n elements and let a_{n+1} be a new element. Then

$\mathbf{P}(S \cup \{a_{n+1}\}) = \mathbf{P}(S) \cup (\cup_{Q \in \mathbf{P}(S)} (\{a_{n+1}\} \cup Q))$

The C code is in Fig. S7.1.

The running time of *powerset*(S) is $O(2^n)$ where n is the number of elements on the linked list S. Note that each successive element on S doubles the size of the powerset.

7.3.9: $\mathbf{P}(\mathbf{P}(\mathbf{P}(\emptyset))) = \{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\}\}$

◇ Section 7.4

7.4.1(a): The function *union*(L, M) in Fig. S7.2 is an implementation of the pseudo code in Figure 7.5. It assumes that `LIST` and `CELL` are defined as in Section 7.4. The code for the function *lookup*(x, L) is given in Figure 6.3 of the text.

```
PLIST dble(PLIST P, int e); /* applies recursion to produce
    all the sets on list P and all those sets with element
    e inserted */

PLIST powerset(SLIST S) {

    PLIST P;

    if(S == NULL) {/* create P, a list containing only the
            empty set */
        P = (PLIST) malloc(sizeof(struct PCELL));
        P->element = NULL;
        P->next = NULL;
        return P;
    }
    /* here, S is not empty. Apply recursion using its
        first element as a_{n+1}*/
    return dble(powerset(S->next), S->element);
}

PLIST dble(PLIST P, int e) {

    SLIST newS;
    PLIST newP;

    if(P==NULL) return NULL;
    /* now, P has at least one element. Double its tail */
    P->next = dble(P->next, e);
    /* Let S be the set in P->element. Create a new PCELL
        to represent S union {e}. The new SCELL holds
        e, and the rest of S union {e}is just a pointer
        to S itself. (Note the savings in the number of
        SCELLS created.) */
    newS = (SLIST) malloc(sizeof(struct SCELL));
    newP = (PLIST) malloc(sizeof(struct PCELL));
    newS->element = e;
    newS->next = P->element;
    newP->element = newS;
    newP->next = P;
    return newP;
}
```

Fig. S7.1. Computing the power set.

7.4.1(b): The following pseudocode computes the intersection sets L and M.

```
for(each x on L)
    if(lookup(x, M)
        insert(x, inter);
```

The C implementation is similar to Fig. S7.2.

7.4.1(c): The difference of sets L and M can be computed as follows:

```
LIST union(LIST L, LIST M) {

LIST union, newCell;

    /* copy L to union */
    union = NULL;
    while(L != NULL) {
        newCell = (LIST) malloc(sizeof(struct CELL));
        newCell->element = L->element;
        newCell->next = union;
        union = newCell;
        L = L->next
    }
    while(M != NULL) {
        if(!lookup(M->element, union)) {
            newCell = (LIST) malloc(sizeof(struct CELL));
            newCell->element = M->element;
            newCell->next = union;
            union = newCell
        }
    M = M->next
    }
}
```

Fig. S7.2. Taking the union of unsorted lists.

```
for(each x on L)
    if(!lookup(x, M)
        insert(x, difference)
```

Again, the C implementation is similar to Fig. S7.2.

7.4.3: If we allowed *union* to use portions of the lists L and M in the answer, we could simplify the union program in Figure 7.6 by replacing line (8) by

```
union = M
```

and line (10) by

```
union = L
```

7.4.5: We can write a program almost identical to Figure 7.6 replacing union by symmetric difference. However, when the first elements of L and M are the same we discard both from the symmetric difference. Thus, we replace lines (11) and (12) by

```
(11)    else if(L->element == M->element)
(12)        return symdiff(L.next, M.next);
```

◇ Section 7.5

7.5.1:

a) A pinochle deck contains the A, K, Q, J, 10 and 9 of each suit. The characteristic vector is thus

$$10^7 1^6 0^7 1^6 0^7 1^6 0^7 1^5$$

b) The characteristic vector for the red cards (diamonds and hearts) is

$$0^{13} 1^{26} 0^{13}$$

c) The Jack of hearts and the Jack of spades are one-eyed. The King of hearts is the suicide king. The characteristic vector for these three cards is

$$0^{36} 1 0 1 0^{10} 1 0^2$$

7.5.3: Given a small universal set U of n elements, we could use an array of integers `int S[n]` to represent a bag. `S[i]` would present the number of times the ith element appears in the bag.

a) To insert another instance of the ith element, we would increment `S[i]` by one.

b) To delete an instance of the ith element, we would decrement `S[i]` by one, provided it was not already 0.

c) To lookup the number of times element i occurs, we return `S[i]`.

◇ Section 7.6

7.6.2(a): There is a substantial unevenness in the lengths of English words, and the division into buckets is therefore not very even. The number peaks around 7 letters. For example, the 24,470 words in `/usr/dict/words`, 4045 have length 7. They divide themselves into buckets as follows.

Bucket	0	1	2	3	4	5	6	7	8	9
Count	1891	1100	639	1034	2253	3138	3813	4053	3583	2970

Should one interpret the problem as assuming that a random selection of word *occurrences* from a typical document (rather than from a list of all possible words) is to be hashed, then we would instead find the low numbers predominated, since most occurrences of words are short. Thus, the hash function would perform poorly in this situation as well.

7.6.3(a):

```
void bucketDelete(ETYPE x, LIST* pL) {
    if((*pL) != NULL) {
        if((*pL)->element == x) (*pL) = (*pL)->next;
        else delete(x, (*pL)->next);
    }
}

void delete(ETYPE x, HASHTABLE S) {
    bucketDelete(x, &(S[h(x)]));
}
```

7.6.3(b):

```
Boolean bucketLookup(ETYPE x, LIST L) {
    if(L == NULL) return FALSE;
    else if(L->element == x) return TRUE;
    else return lookup(x, L->next);
}

Boolean lookup(ETYPE x, HASHTABLE S) {
    return bucketLookup(x, S[h(x)]);
}
```

◇ Section 7.7

7.7.1: Let $A = \{a\}$ and $B = \{b, a\}$. Then $A \times B = \{(a, b)\}$ and $B \times A = \{(b, a)\}$.

7.7.3:

a) R is a partial function because every node in a tree has at most one parent.

b) R is not a total function from S to S because the root of a tree has no parent.

c) T is never a one-to-one correspondence because R is not a total function from S to S.

d) The graph for R is isomorphic to the tree.

7.7.5: F is a partial function from S to T with the following properties:

1. For every element $((a, b), c)$ in S there is an element $(a, (b, c))$ in T such that $F(((a, b), c)) = (a, (b, c))$.
2. For every element $(a, (b, c))$ in T there is an element $((a, b), c)$ in S such that $F(((a, b), c)) = (a, (b, c))$.
3. For no b in T are there two distinct elements x_1 and x_2 in S such that $F(x_1) = F(x_2) = b$.

Hence F is a one-to-one correspondence from S to T.

7.7.7:

a) The graph of the inverse of R is obtained by reversing the directions of the arcs in the graph for R.

b) The inverse need not be a function. If R is a total function from domain A to range B, there may be a b in B for which there is no a in A such that (a, b) is in R.

If R is a one-to-one correspondence from A to B, then its inverse is a total function from B to A.

◇ Section 7.8

7.8.1(a):

```
void delete(DTYPE a, LIST* F) {
    if((*F) != NULL)
        if((*F)->domain == a) (*F) = (*F)->next;
        else delete(a, (*F)->next);
}
```

7.8.1(b):

```
RTYPE lookup(DTYPE a, LIST F) {
    if(F == NULL) return UNDEFINED;
    else if(F->domain == a) return F->range;
    else return lookup(a, F->next);
}
```

Here we assume `RTYPE` includes the value `UNDEFINED`.

7.8.3(a):

```
void insertBucket(DTYPE a, RTYPE b, LIST* pL) {
    if((*pL) == NULL) {
        (*pL) = (LIST) malloc(sizeof(struct CELL));
        (*pL)->domain= a;
        (*pL)->range= b;
        (*pL)->next= NULL;
    }
    else if((*pL)->domain == a) (*pL)->range = b;
    else insertBucket(a, b, (*pL)->next);
}

void insert(DTYPE a, RTYPE b, HASHTABLE F) {
    insertBucket(a, b, &(F[h(a)]));
}
```

7.8.3(b):

```
void deleteBucket(DTYPE a, LIST* pL) {
    if((*pL) != NULL)
        if((*pL)->domain == a) (*pL) = (*pL)->next;
        else deleteBucket(a, (*pL)->next)
}

void delete(DTYPE a, HASHTABLE F) {
    deleteBucket(A, &(F[h(a)]));
}
```

◇ Section 7.9

7.9.1: This is just a slight rewrite of the function *lookup* from Fig. 7.24. Here we search the range rather than the domain for the value b and produce a list of varieties v such that (v, b) is in L.

```
PLIST lookup(PVARIETY p, RLIST L) {

    PLIST P;

    if(L == NULL) return NULL;
    else if(L->pollinizer == b) {
        P = (PLIST) malloc(sizeof(struct PCELL));
        P->variety = L->variety;
        P->next = lookup(b, L->next);
        return P;
    }
    else return lookup(b, L->next);
}
```

7.9.3(a):

```
void insertP(PVARIETY p, PLIST* pL) {
    if((*pL) == NULL) {
        (*pL) = malloc(sizeof(struct PCELL));
        (*pL)->pollinizer = p;
        (*pL)->next = NULL;
    }
    else if((*pL)->pollinizer != p)
        insert P(p, L->next);
}

void insert(PVARIETY v, PVARIETY p, PLIST Pollinizers[]) {
    insertP(p, Pollinizers[p]);
}
```

7.9.5: We shall prove by induction on n:

STATEMENT $S(n)$: On a list L of length v, $lookup(a, L)$ returns a list of all the elements b such that (a, b) is on L.

BASIS. $n = 0$. When L is empty, statements (1) and (2) return `NULL` as the value of $lookup(a, L)$.

INDUCTION. We assume $S(n)$ and prove $S(n+1)$. Suppose L is a list of length $n+1$. The initial call $lookup(a, L)$ looks at the first cell of L. There are two cases to consider:

(1) If `L->variety` matches a, then a new cell P is created, a pointer to which becomes the output of lookup. The first component of P is set to the value of `L->pollinizer` and the second component is set to point to the list of b's created by the recursive call to the tail of L. The tail of L is of length n and so by the inductive hypothesis, the recursive call correctly returns the list of all b's such that (a, b) is on the tail of L. The list returned for all of L is therefore correct and the inductive step has been proved.

(2) If `L->variety` does not match a, then the output of *lookup* is the value returned by a recursive call to the tail of L, which is of length n. By the inductive hypothesis, *lookup* returns the correct value for the tail of L. Therefore, the list returned for all of L is correct and the inductive step has been proved.

7.9.7: For dictionaries and functions: With linked lists, the operations insert, delete, and lookup, each take $O(n)$ time on average with characteristic vectors, these operations each take $O(1)$ time. With the hash table representation, each takes $O(n/B)$ time on average, but can take $O(n)$ time in the worst case. Here n is the number of pairs in the function and B is the number of buckets in the hash table.

For relations: The same observations apply to the linked-list representation of relations. With characteristic vectors, lookup takes $O(1)$ time but an insert or delete takes $O(n)$ time on average because we have to search the entire list for a given domain value to make sure that a pair (a, b) is not already present. The parameter b is the average number of b's for a given a. With the hash-table implementation each operation takes $O(max(n, n/B))$ time on average.

◇ Section 7.10

7.10.1: Let R be the relation such that aRa for element a. Then R is reflexive on the domain $\{a\}$ but not on the domain $\{a, b\}$.

7.10.3:

a) R is not reflexive because $abcdRabcd$ is false when $b \neq a$.

b) R is not symmetric because if $abcdRbcda$ is true, then $bcdaRabcd$ is false when a, b, c and d are distinct letters.

c) R is not transitive because if $abcdRbcda$ and $bcdaRcdab$ are true then $abcdRcdab$ is false when a, b, c and d are distinct.

d) R is not antisymmetric nor transitive. Hence R is not a partial order.

e) R is not an equivalence relation because it is not symmetric or transitive.

7.10.5: The problem with the "proof" is that there may be no y such that xRy. Let D be the domain $\{a\}$ and let R by the empty relation on D. Trivially, R is a symmetric and transitive relation on D.

7.10.7: To count the number of arcs in the full graph, we need to count the number of pairs of sets (S, T) such that $S \subseteq T \subseteq U$. Each of the n elements of U may be placed in one of the following three sets; S, $T - S$, and $U - T$. By the method of Section 4.2, there are 3^n ways to make this assignment. Thus, the full graph for $\subseteq_U$ has 3^n arcs.

In the reduced graph, each set has n arcs, one to each of the sets formed by inserting or deleting one of the n elements of U. Since each arc is thus counted twice, once for each end, the number of arcs is $n2^n/2 = n2^{n-1}$. Therefore, $3^n - n2^{n-1}$ arcs are saved.

7.10.9: We shall prove by induction on n

STATEMENT $S(n)$: If $a_0Ra_1, a_1Ra_2, ..., a_{n-1}Ra_n$ and R is transitive, then a_0Ra_n.

BASIS. $n = 1$. Clearly, a_0Ra_1 is true.

INDUCTION. We assume $S(n)$ and prove $S(n + 1)$. Consider the sequence of $n + 1$ pairs $a_0Ra_1, a_1Ra_2, ..., a_{n-1}Ra_n, a_nRa_{n+1}$. ¿From the inductive hypothesis, we know a_0Ra_n. By transitivity a_0Ra_n and a_nRA_{n+1} imply a_0Ra_{n+1}, proving the inductive step.

7.10.11:

a) R is not reflexive because aRa is false.

b) R is symmetric because if aRb then a and b have the common divisor in both situations.

c) R is not transitive. For example, $2R6$ and $6R9$ but $2R9$ is false since 2 and 9 do not have a common divisor other than 1.

d) R is not a partial order since R is neither transitive nor antisymmetric.

e) R is not equivalence relation since it is neither reflexive nor transitive.

◇ Section 7.11

7.11.1: Let A be a set of sets and let E be an equipotence relation on A; that is, $S\ E\ T$ if there is a 1-1 correspondence from S to T. We shall show that E is (a) reflexive, (b) symmetric, and (c) transitive.

a) $S\ E\ S$ for every set S in A because we can define the identity function $f(x) = x$ for all x in S as a 1-1 correspondence from S to itself.

b) If $S\ E\ T$, then $T\ E\ S$. Since $S\ E\ T$, there is a 1-1 correspondence from S to T. We can show that f^{-1}, the inverse of f, is a 1-1 correspondence from T to S.

c) If $S\ E\ T$ and $T\ E\ R$, then we shall show $S\ E\ R$. Let f be a 1-1 correspondence from S to T and g a 1-1 correspondence from T to R.

We need to show the composition of f and g is a 1-1 correspondence from S to R.

i) Since f and g are both total functions, for every x in S, element $g(f(s))$ is R.

ii) Let z be an element in R. Then $x = f^{-1}(g^{-1}(z))$ is an element in S such that $g(f(x)) = z$.

iii) There is no z in R for which there exist x_1 and x_2 in S such that $g(f(x_1)) = z$. If there were, then either f or g would not be a 1-1 correspondence.

Since E is reflexive, symmetric, and transitive, it is an equivalence relation.

7.11.3: (a) The 1-1 correspondence is $f(i) = i^2$.

(b) Start with the pairing function of Example 7.41. For convenience, rewrite it in terms of x and y rather than i and j. Then the pair (x, y) is associated with natural number $(x + y)(x + y + 1)/2 + x$. Now, to find a unique natural number for the triple (i, j, k), start by associating j and k with a natural number z. Using the pairing function, let $z = (j + k)(j + k + 1)/2 + j$. Then, pair i with z, giving $(i + z)(i + z + 1)/2 + i$, or in terms of i, j, and k: $\big(i + (j + k)(j + k + 1)/2 + j\big)\big(i + (j + k)(j + k + 1)/2 + j + 1\big)/2 + i$.

7.11.6: Let S_i be the set of size i that is a subset of S. Define a sequence of members of S which we call $x_1, x_2, \ldots$, as follows: x_i is the least member of S_i that is not one of $x_1, x_2, \ldots, x_{i-1}$. Since there are only $i - 1$ of the latter integers, there must be at least one integer in S_i that is not any of them. Thus, we can find x_i for any positive integer i, and all of the x_i's are distinct.

Now, consider the 1-1 correspondence $f(a)$ defined as follows:

1. If $a = x_i$ for some i, then $f(a) = x_{i+1}$.
2. If a is not one of the x_i's, then $f(a) = a$. Clearly f is a 1-1 correspondence, and its range is $S - \{x_1\}$. Thus, S has a 1-1 correspondence with one of its subsets.

Chapter 8. The Relational Data Model

◇ Section 8.2

8.2.1:

a) For the relation StudentId-Name-Address-Phone we define the record structure:

```
struct {
    int StudentId;
    char Name[30];
    char Address[50];
    char Phone[10];
}
```

c) For Course-Day-Hour:

```
struct {
    char Course[5];
    char Day[2];
    char Hour[4];
}
```

◇ Section 8.3

8.3.1:

a) {StudentId, Address} would be a key assuming that a student has only one phone at a given address.

b) We could use the relation scheme

StudentId-Name-HomeAddress-LocalAddress-HomePhone-LocalPhone

StudentId is a key for this relation.

c) We need to separate the scheme into three schemes:

StudentId-Name
StudentId-Address
StudentId-Phone

Any other decomposition either has redundancy or does not allow us to associate names, ID's, addresses, and phones properly. StudentId is a key for each scheme.

◇ Section 8.4

8.4.1: For Exercise 8.3.2, we suggest a database scheme with three relations:

1) LicenseNo, Name, Address
2) SerialNo, Manf, Model, RegNo
3) LicenseNo, RegNo

The first relation associates a name and address with each driver, identified by the license number. LicenseNo is a key for the first relation. This attribute can serve as the domain for this relation with (Name, Address) forming the range.

Let us assume there are one million drivers and two million automobiles. As a primary index structure for the first relation we could use a hash table on LicenseNo. We could use 500,000 buckets assuming each bucket would contain 2 tuples on average. This data structure would allow queries (1) and (3) to be answered in $O(1)$ time on average.

The second relation associates with each automobile its manufacturer, model number, and registration number. Each of SerialNo or RegNo could serve as a key. As a primary index structure, we can choose a hash table on registration number. We could use 500,000 buckets assuming each bucket would contain 4 SerialNo-Manf-Model-RegNo–tuples on average. This structure would allow query (5) to be answered in $O(1)$ time.

The third relation records the automobiles owned by each driver. Since an automobile can have joint owners, and a person can own several automobiles, {LicenseNo, RegNo} is the key. We suggest a hash table with 500,000 buckets and key RegNo as a primary index structure. The hash table is indexed by registration number and would allow query (6) to be answered in $O(1)$ time on average, assuming the average number of owners per automobile is a small constant.

Note that query (4) can be answered in $O(k)$ time by first consulting the third relation to obtain the k LicenseNo's for a given registration number, and then consulting the first relation to determine the name of each driver.

Only query (2) cannot be answered in $O(1)$ time with this database scheme. For that, we would need a fourth, redundant relation scheme — {Name, LicenseNo}, with primary index on Name (which is not a key). Alternatively, and preferably, we would add a secondary index on Name, as discussed in Section 8.5, to the first relation.

◇ Section 8.5

8.5.1: The following declaration of the standard kind of cells

```
DefCell(TUPLELIST, SNAME, SNAMELIST);
```

Lets us create a linked list of `SNAME` ("same name") cells, each of which has as element a pointer to a tuple with that name.

We change the declaration of `NODE` in Fig. 8.5 of the text by making the second field be the header of a linked list of `SNAME` cells:

```
typedef struct NODE *TREE;
struct NODE {
    char Name[30];
    SNAMELIST tuples;
    TREE lc, rc;
}
```

Here is an outline of a function $printTuples(x, T)$ that prints all the tuples in the binary tree T that have x for the `Name` attribute.

```
void printTuples(char x[], TREE T) {
    if(T != NULL)
    if(eq(x, T->Name)) printList(T->tuples);
    else if(lt(x, T->Name)) printTuples(x, T->lc);
    else printTuples(x, T->rc);
}
```

The functions $eq(x, y)$ and $lt(x, y)$ determine whether $x = y$ or $x < y$, respectively, where x and y are names. The function $printList(p)$ lists all of the $SNAP$ tuples pointed to by the elements of the `SNAMELIST` pointed to by p.

8.5.3: For the relations and primary index structures in Exercise 8.4.1

i) Make Name a secondary index of relation (1).

ii) The primary index for relation (1) on LicenseNo serves.

iii) Make LicenseNo a secondary index of relation (3).

iv) Make Address a secondary index of relation (1).

v) The primary index for relation (3) on RegNo serves.

◇ Section 8.6

8.6.1: Suppose the Course-StudentId-Grade relation has an index on Course alone. Then one way to find C. Brown's grade in CS101 is to proceed in two stages. In stage (1), we use step (1) of Fig. 8.9 to find the tuples in the $SNAP$ relation with C. Brown in the Name field. Suppose that there are k such tuples. Since there is an index on Name, these tuples can be found in $O(k)$ time.

In stage (2), we use the index on Course to find all tuples in the CSG relation with CS101 in the Course field. Assuming that there are c such tuples, these tuples can be found in $O(c)$ time. For each CSG tuple found, we check the StudentId field to see if it matches one of the StudentId's found in stage (1). If it does, we print the associated grade. The comparison of StudentId's for one CSG tuple can be done in $O(k)$ time. Thus, the grades of all C. Brown's in CS101 can be found in $O(ck)$ time.

Now, suppose the Course-StudentId-Grade relation has an index on StudentId alone. Then to find C. Brown's grade in CS101, we proceed in two stages. Stage (1) is the same as the first stage when the index is on Course. We find the k tuples in the $SNAP$ relation with C. Brown in the Name field. In stage (2) we use the StudentId of each tuple found in stage (1) to index the CSG relation. If the Course field contains CS101, we print the grade. If there are a total of d tuples in the CSG relation that match the StudentId of a C. Brown, then the entire query can be answered in $O(k + d)$ time.

8.6.3: We use three stages to find the prerequisites of the courses taken by C. Brown. In stage (1) we find all tuples in the StudentId-Name-Address-Phone relation that have C. Brown in the Name field. If there are n tuples in the $SNAP$ relation, then

stage (1) takes $O(n)$ time. Let us assume k tuples with C. Brown in the Name field are found.

In stage (2) we search through the Course-StudentId-Grade relation to find all tuples whose StudentId matches the StudentId field of the k tuples found in stage (1). If there are m tuples in the CSG relation, then stage (2) takes $O(km)$. Let us assume c CSG-tuples are found.

In stage (3) we search through the Course-Prerequisite relation to find all tuples whose Course component matches the Course component of one of the c CSG-tuples found in stage (2). For each CP-tuple found, we print the Prerequisite field. Assuming that there are p tuples in the CP relation, this stage takes $O(cp)$ time. This three-stage process takes $O(n + km + cp)$ time.

◇ Section 8.7

8.7.1:

a) $\sigma_{\text{Course = "CS101" AND Day = "M"}}(CDH)$

b) $\sigma_{\text{Day = "M" AND Hour = "9AM"}}(CDH)$

c) $CDH = CDH - \sigma_{\text{Course = "CS101"}}(CDH)$

8.7.3:

a) The courses taken by C. Brown can be expressed by the relational-algebra expression

$$X = \pi_{\text{Course}}(\sigma_{\text{Name = "C.Brown"}}(SNAP) \bowtie CSG)$$

The prerequisites of courses taken by C. Brown can then be expressed by

$$\pi_{\text{Prerequisite}}(X \bowtie CP)$$

Here, we must understand that X is a relation with attribute Course, so the join is on equality of the one column of X with the Course attribute of CP.

b) The students taking courses in Turing Aud. can be expressed by

$$Y = \pi_{\text{StudentId}}(CSG \bowtie \sigma_{\text{Room = "Turing Aud."}}(CR))$$

The phone number of these students is given by

$$\pi_{\text{Phone}}(Y \bowtie SNAP)$$

Here, Y is a relation with attribute StudentId.

c) The prerequisites of CS206 can be expressed by:

$$Z = \sigma_{\text{Prerequisite}}(\pi_{\text{Course = "CS206"}}(CP)))$$

The prerequisites of these courses are given by

$$\pi_{\text{Prerequisite}}(Z \bowtie CP)$$

Here, Z must be regarded as a relation with attribute Course, not Prerequisite.

◇ Section 8.8

8.8.1:

a) Use the primary index to find in $O(1)$ time each tuple in the StudentId-Name-Address-Phone relation with StudentId = 12345. Include the tuple in the answer if the address is not 45 Kumquat Blvd.

b) Use the secondary index on Phone to find in $O(1)$ time each tuple in the $SNAP$ relation with Phone = 555-1357. Include the type in the answer if the name matches C. Brown.

c) Neither index helps. The easiest (and fastest) way to proceed is to iterate through each of the tuples in the $SNAP$ relation, selecting those tuples where Name = "C. Brown" is true or Phone = 555-1357 is true (or both are true). If there are n tuples in the $SNAP$ relation, then this process takes $O(n)$ time.

8.8.3:

a) There are two nested loops, each of which iterates n times. The body of the inner loop is a comparison of tuples and so takes $O(1)$ time. The total time is thus $O(n^2)$.

b) We can sort the relations in $O(n \log n)$ time. As we compare tuples, we find $O(n^{3/2})$ matches, and this time dominates the sorting. The total time is thus $O(n^{3/2})$.

c) We must consider each of the n tuples in S. For each one, we look up the matching tuples in R through the index, taking time proportional to the number of matching tuples found. Since there are $n^{3/2}$ matches in all, the sum of the number of matches, over all tuples in S, is $n^{3/2}$, and thus the total time is $O(n^{3/2})$.

d) Same as (c).

8.8.5:

a) Suppose R and S each have the scheme $\{A, B\}$ and S has an index on attribute A. To compute $R \cap S$ consider each tuple (a, b) in R and use the index on a to find all tuples (a, x) in S. Include (a, b) in $R \cap S$ if$((a, b)$ is also in S. If A is a key for relation S, then in this way we can compute $R \cap S$ in time proportional to the sum of the sizes of R and S.

b) The answer is similar to part (a) except we include (a, b) in $R - S$ if (a, b) is not in S.

◇ Section 8.9

8.9.3: We have, in the solution to Exercise 8.7.1, described each expression with the selections and projections pushed down as far as they go.

8.9.5: First, we prove that if a tuple t is in $\sigma_C(R \bowtie S)$ it is in $\sigma_C(R) \bowtie S$. Since t is in $\sigma_C(R \bowtie S)$, t satisfies condition C and is in $R \bowtie S$. If t is in $R \bowtie S$, then there are tuples r in R and s in S that agree with t on their common attributes, and also

agree with each other on the join attribute. Since $\sigma_C(R)$ makes sense, condition C must involve only attributes of r. Since r and t agree on common attributes, and t satisfies C, r also satisfies C. Thus, in $\sigma_C(R) \bowtie S$, tuples r and s join to make t, proving that t is in $\sigma_C(R) \bowtie S$.

Conversely, suppose t is in $\sigma_C(R) \bowtie S$. Then there are r in $\sigma_C(R)$ and s in S that agree with t and with each other on common attributes. Since r is in $\sigma_C(R)$, r must by in R and must satisfy C. Therefore, t, which agrees with r on whatever attributes C mentions, must also satisfy C. When we join $R \bowtie S$, r in R and s in S join to form t. Since t satisfies C, we conclude that t is in $\sigma_C(R \bowtie S)$.

8.9.7: Let R be the relation $\{(a,b),(a,c)\}$ and S the relation $\{(a,c)\}$. Both relations have attributes A and B. Here, $\pi_A(R-S) = \{a\}$ but $\pi_A(R) - \pi_A(S) = \{a\} - \{a\} = \emptyset$. Thus, $\pi_A(R-S) \neq \pi_A(R) - \pi_A(S)$.

Chapter 9. The Graph Data Model

◇ Section 9.2

9.2.1:

a) There are 8 arcs.

b) There are 2 simple paths: ad and $abcd$.

c) Nodes a and e are predecessors of node b.

d) Nodes c and f are successors of node b.

e) There are 5 simple cycles: $abfa$, $abcdefa$, $adefa$, $bcdeb$, $adebfa$.

f) $abfabfa$ is the only nonsimple cycle of length at most 7.

9.2.3: A *complete* directed graph has an arc from each node to every other node including itself. A complete directed graph has the maximum possible number of arcs: an n-node complete graph has n^2 arcs. Thus, a 10-node graph has at most 100 arcs. The smallest number of arcs any graph can have is zero.

9.2.5: The number number of arcs an n-node acyclic directed graph can have is $\binom{n}{2}$ $= n(n-1)/2$. To see this, we can start with a complete n-node undirected graph in which there is an edge between every pair of distinct nodes. Such a graph has $\binom{n}{2}$ edges. We can then assign a direction to each edge $\{i, j\}$ so that the edge is directed from node i to node j if $i < j$. We can show that the resulting directed graph is acyclic and has the maximum possible number of edges.

9.2.7: The cycle $(0, 1, 2, 0)$ can also be written as the cycles $(1, 2, 0, 1)$ and $(2, 0, 1, 2)$.

9.2.9: Let S be the relation defined on the subset of the nodes of the graph that are involved in simple cycles. To show that S is an equivalence relation, we need to show that it is reflexive, symmetric, and transitive.

a) *Reflexivity*. If node u is involved in a simple cycle, then there is a simple cycle that begins and ends at u. Thus, uSu.

b) *Symmetry*. If uSv, then vSu because u and v are included in the same simple cycle.

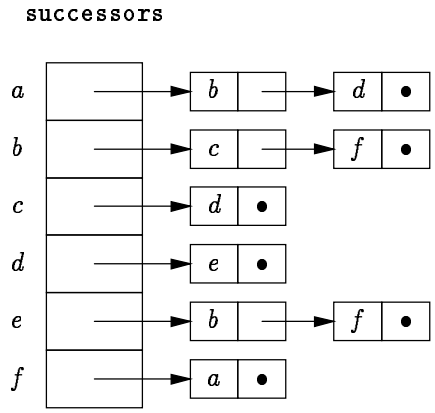
c) *Transitivity*. Suppose uSv and vSu. Then there are two intersecting simple cycles that include nodes u and w. Let a and b be the first and last nodes of intersection on the simple cycle from u to v to u. Then $u - a - w - b - u$ is a simple cycle that includes u and w. Therefore, uSw.

◇ Section 9.3

9.3.1(a): An appropriate type definition is

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    LIST next;
};

LIST successors[MAX];
```



9.3.1(b): An appropriate definition of the adjacency matrix would be

```
BOOLEAN arcs[MAX][MAX];
```

	a	*b*	*c*	*d*	*e*	*f*
a	0	1	0	1	0	0
b	0	0	1	0	0	1
c	0	0	0	1	0	0
d	0	0	0	0	1	0
e	0	1	0	0	0	1
f	1	0	0	0	0	0

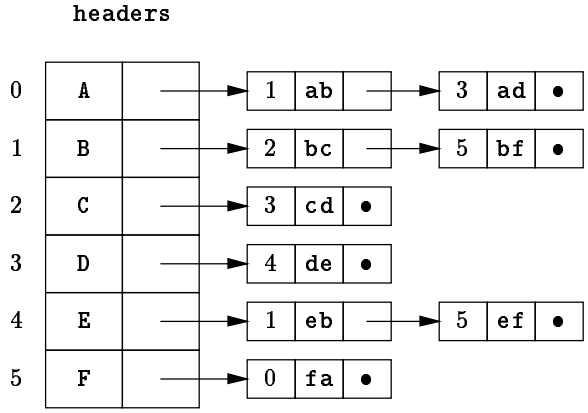
9.3.3(a): Appropriate type definitions for the lists of cells are

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    char arcLabel[3];
    LIST next;
};
```

Note that we must leave room for the null character `'\0'` at the end of the two-character arc label. (We have not shown the null character in the following figures.)

We declare the array of list headers by

```
struct {
    char nodeLabel;
    LIST successor;
} headers[MAX];
```



9.3.3(b): An appropriate definition of the adjacency matrix would be

```
typedef char ARCTYPE[3];
ARCTYPE arcs[MAX][MAX];
```

	a	b	c	d	e	f
a	-	ab	-	ad	-	-
b	-	-	bc	-	-	bf
c	-	-	-	cd	-	-
d	-	-	-	-	de	-
e	-	eb	-	-	-	ef
f	fa	-	-	-	-	-

9.3.5: We shall prove by induction on n

STATEMENT $S(n)$: In an undirected graph with n nodes and e edges, the sum of the degrees of the nodes is $2e$.

BASIS. For the basis, we choose $n = 1$. A single-node graph has 0 edges and the sum of the degrees of the nodes is 0.

INDUCTION. Assume $S(n)$ holds for all graphs of n nodes. Consider any graph G of $n+1$ nodes and pick a node x in G with m edges incident on x.

If we remove x and all edges incident on x from G, we are left with a graph G' of n nodes and e edges. By the inductive hypothesis, the sum of the degrees of the nodes of G' is $2e$. When we restore x to G' along with its incident edges, we see that G has $m + e$ edges and the sum of the degrees of its nodes is $2(m + e)$. This proves the inductive step.

9.3.7: For an undirected graph, an edge appears twice in an adjacency-matrix and an adjacency-list representation.

a) Function to insert edge (a, b) into an adjacency matrix:

```
void insert(NODE a, NODE b, BOOLEAN edges[MAX][MAX])
{
    edges[a][b] = TRUE;
    edges[b][a] = TRUE;
}
```

The delete function is similar except we make the two entries in the array `FALSE`.

b) Function to insert edge (a, b) for an adjacency-list representation:

```
void insert(NODE a, NODE b, LIST successors[]);
{
    insertList(b, &successors[a]);
    insertList(a, &successors[b]);
}
```

For `insertList` we can use the function in Fig. 6.5.
The delete function is similar.

◇ Section 9.4

9.4.1: There are two connected components:

Escanba, Marquette, Menominee, Sault Ste. Marie

and

Ann Arbor, Battle Creek, Detroit, Flint, Grand Rapids,
Kalamazoo, Lansing, Saginaw

◇ Section 9.5

9.5.6:

a) It is easy to see why a node of odd degree inhibits an Euler circuit. The circuit must visit the node some number of times, and each time it visits, it enters and leaves on two different edges (the direction in which we traverse the circuit is arbitrary, but once we pick a direction, the edges incident upon a node v can be identified as entering or leaving). It follows that there are as many entering edges as leaving edges for a node v, and therefore the number of edges incident upon v is even.

For the converse, we need to show how to construct an Euler circuit when all the degrees are even. We do so in part (b), where we are asked not only to produce an algorithm to construct Euler circuits, but to design an efficient algorithm.

b) We need to use an appropriate data structure: adjacency lists, plus a list of all the edges. We also need to generalize the notion of an Euler circuit to cover the case in which the graph has more than one connected component. In that case, we say an "Euler circuit" for the graph is an Euler circuit for each connected component. Start with any edge, say $\{v_0, v_1\}$, and arbitrarily pick one of the nodes, say v_0, as the beginning of a path. Extend the path to nodes $v_2, v_3, \ldots$, without reusing any edge, which we may do since every time we enter a node, we know it has even degree so there is an unused edge by which to leave. Eventually, we repeat a node on the path, say v_i.

Now, remove the edges of the cycle, say $v_i, v_{i+1}, \ldots, v_k, v_i$, but leave the nodes. Recursively find an "Euler circuit" for the remaining graph, but start with the portion of the path already constructed, $v_0, v_1, \ldots, v_i$. Note that we quote "Euler circuit," because the resulting graph may not be connected. Finally, we need to assemble an Euler circuit for the entire graph. We use the removed cycle $v_i, \ldots, v_k, v_i$ as a base and follow it around. Each time we visit a node, say v_j, if we have not previously visited any nodes from its connected component, we follow the Euler circuit for this connected component, starting and ending at v_j. Then we continue around the cycle to v_{j+1}. When we return around the cycle to v_i, we have an Euler circuit for the entire graph.

♢ Section 9.6

9.6.3:
a)

Tree arcs: ab, bc, cd, de, ef
Forward arcs: ad, bf
Backward arcs: eb, fa
There are no cross arcs

b)

Tree arcs: ab, bf, bc, cd, de
Forward arcs: ad
Backward arcs: eb, ef, fa
There are no cross arcs

c)

Tree arcs: de, ef, fa, ab, bc
Forward arcs: eb
Backward arcs: ad, bf, cd
There are no cross arcs

◇ Section 9.7

9.7.1: There are four topological orders:

$dcebfa$
$dcefba$
$decbfa$
$decfba$

9.7.3: The connected components are

Escanba, Marquette, Menominee, Sault Ste. Marie

and

Ann Arbor, Battle Creek, Detroit, Flint, Grand Rapids
Kalamazoo, Lansing, Saginaw

◇ Section 9.8

9.8.1:

CITY	DISTANCE
Detroit	0
Ann Arbor	28
Escanba	INFTY
Flint	58
Grand Rapids	138
Kalamazoo	138
Lansing	78
Marquette	INFTY
Menominee	INFTY
Saginaw	89
Sault Ste. Marie	INFTY

9.8.3(b):

SPECIFIES	TIME
AF	0
AA	0.8
HH	1.0
AR	1.3
HE	2.2
AB	1.2
HS	2.9

◇ Section 9.10

9.10.1:

a) The chromatic number of the graph in Fig. 9.4 is 3.

b) The clique number is 3.

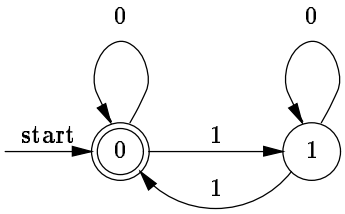
c) { Maili, Pearl City, Wahiawa } and { Hilo, Kamuela, Kona } are both cliques of size 3.

Chapter 10. Patterns, Automata, and Regular Expressions

◇ Section 10.2

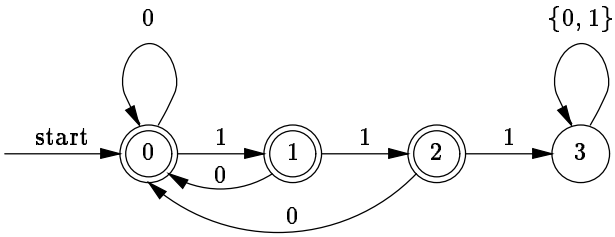
10.2.1:

a) The following automaton accepts strings of 0's and 1's having an even number of 1's:



This automaton is in state 0 if it has seen an even number of 1's and it is in state 1 if it has seen an odd number of 1's.

b) The following automaton accepts any string of 0's and 1's that does not contain 111 as a substring:



In this automaton, state 0 means the previous input symbol was not a 1, state 1 means the previous input symbol was a 1 and the one before that was not a 1, state 2 means the two previous input symbols were 1's, and state 3 means the three previous input symbols were 1's.

◇ Section 10.3

10.3.3: The nondeterministic automaton in Fig. (a) accepts all strings of letters ending in `father`, `man`, or `son`.

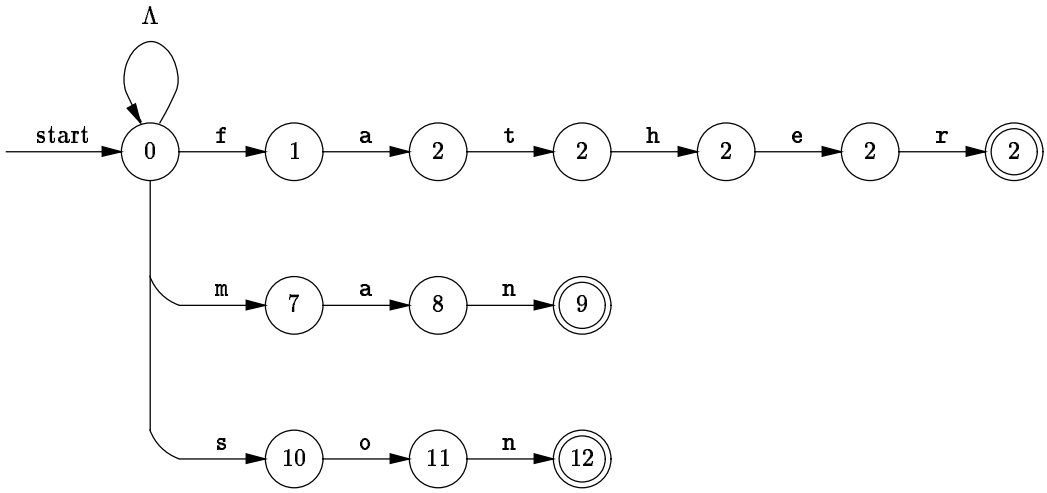


Fig. (a). Automaton accepting strings ending in `father`, `man`, or `son`.

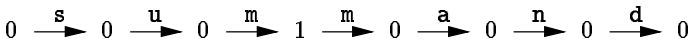


Fig. (b). Simulation of automaton of Fig. 10.10.

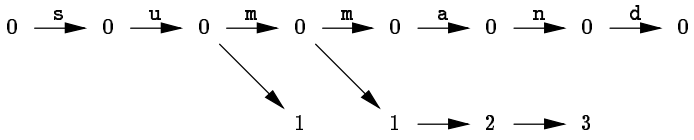


Fig. (c). Simulation of automaton of Fig. 10.11.

10.3.5: Figures (b) and (c) simulate the automata in Figs. 10.10 and 10.11, respectively.

◇ Section 10.4

10.4.3(a): A deterministic automaton for Fig. 10.24(a) is shown in Fig. (d).

◇ Section 10.5

10.5.1: The two regular expressions (**ac** | **abc** | **abbc**) and **a**(**c** | **b**(**c** | **bc**)) also define the same language.

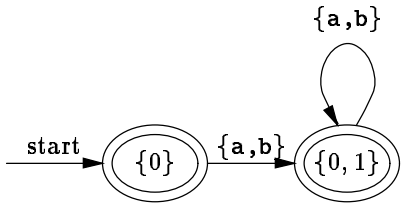


Fig. (d). Solution to Exercise 10.4.3(a).

10.5.3:

a) **b***((**aa*****b***)*

b) (**0** | **1** | ⋯ | **9**)(**0** | **1** | ⋯ | **9**)*.(**0** | **1** | ⋯ | **9**)*

c) **0***(**10*****10***)*

10.5.5:

a) (**a** | (**bc**)) | (**de**)

b) (**a** | (**b***)) | ((**a** | **b**)***a**)

10.5.7:

a) $\emptyset \mid \epsilon$ defines either the empty set or the set containing the empty string.

c) (**a** | **b**)* defines the set of all strings of a's and b's (including the empty string).

e) (**a*****ba*****b**)***a*** defines the set of all strings of a's and b's containing an even number of b's.

g) R^{**} is the same as R^*, that is, the set consisting of the concatenation of zero or more strings from the set defined by R.

◇ Section 10.6

10.6.1:

a) Single-character operators and punctuation symbols in C:

```
!"#$%&'()*+,-./:;<=>?[]^{}~
```

c) Lower-case consonants:

[**bcdfghjklmnpqrstvwxyz**]

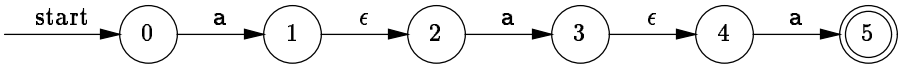
◇ Section 10.7

10.7.1: We first show the forward containment $L((S \mid T)R \subseteq L(SR \mid TR)$. If x is in $L((S \mid T)R)$, then $x = yr$ where y is in $L(S)$ or $L(T)$ and r is in $L(R)$. If y is in $L(S)$, then x is in $L(SR)$. If y is in $L(T)$, then x is in $L(TR)$. In either case, x is in $L(SR \mid TR)$. Thus, the forward containment holds.

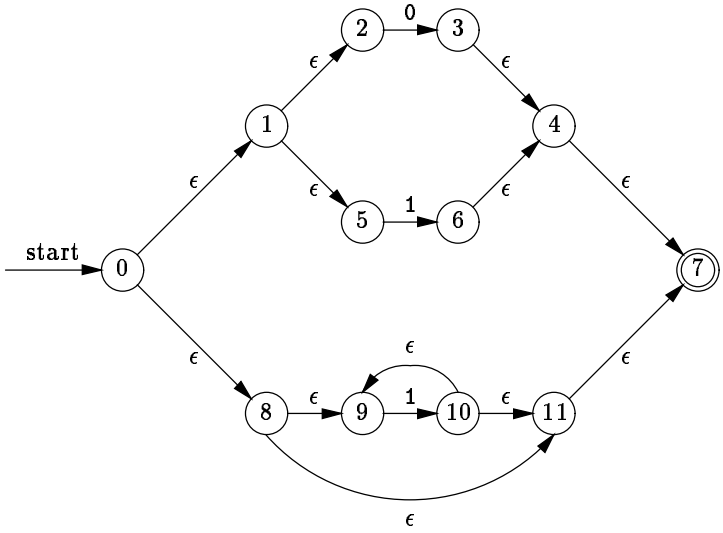
We now show the reverse containment $L((S \mid T)R) \supseteq L(SR \mid TR)$. If x is in $L(SR \mid TR)$, then x is in either $L(SR)$ or $L(TR)$. If x is in $L(SR)$, then $x = sr$ where s is in $L(S)$ and r is in $L(R)$. Thus, x is in $L((S \mid T)R$. Similarly, if x is in $L(TR)$, we can show x is in $L((S \mid T)R$. We have now shown the reverse containment holds.

♢ Section 10.8

10.8.1:
a) Automaton for **aaa**:



c) Automaton for (**0** | **1** | **1***)*:



♢ Section 10.9

10.9.1:
a) $(\Lambda$ - **a**$)^*$**a**$(\Lambda$ - **e**$)^*$**e**$(\Lambda$ - **i**$)^*$**i**$(\Lambda$ - **o**$)^*$**o**$(\Lambda$ - **u**$)^*$**u**

c) Λ^***man**

e) $(\Lambda$ - **a**$)^*$**a**$(\Lambda$ - **a**$)^*$**a**

g) Λ^***man**

Chapter 11. Recursive Description of Patterns

◇ Section 11.2

11.2.1: In Pascal, an identifier is a string of letters and digits, beginning with a letter. We can define an identifier with the following grammar.

$\langle Letter \rangle \rightarrow \mathbf{A} \mid \mathbf{B} \mid \cdots \mid \mathbf{Z} \mid \mathbf{a} \mid \mathbf{b} \mid \cdots \mid \mathbf{z}$
$\langle Digit \rangle \rightarrow \mathbf{0} \mid \mathbf{1} \mid \cdots \mid \mathbf{9}$
$\langle Identifier \langle \rightarrow \langle Identifier \rangle \langle Letter \rangle$
$\langle Identifier \langle \rightarrow \langle Identifier \rangle \langle Digit \rangle$
$\langle Identifier \langle \rightarrow \langle Letter \rangle$

Strictly speaking, a Pascal reserved word such as `program` or `begin` cannot be used as an identifier. Expressing this restriction grammatically would greatly complicate the grammar, and in Pascal compilers, this type of error is caught by other means.

11.2.3: In Pascal, a real number begins with an optional sign, followed by one or more digits, followed by a decimal point, followed by one or more digits (e.g., `0.1`, `1.0`, `3.14`). In addiition, a real number may have at its end a scale factor consisting of an upper case `E`, followed by an optional sign, followed by one or more integers (e.g., `0.1E2`, `1.0E-2`). To allow reals as operands, we can add the following productions to the grammar of Fig. 11.2 of the text:

$\langle Expression \rangle \rightarrow \langle Real \rangle$
$\langle Real \rangle \rightarrow \langle OptSign \rangle \langle Number \rangle \langle Number \rangle \langle ScaleFactor \rangle$
$\langle OptSign \rangle \rightarrow + \mid - \mid \epsilon$
$\langle ScaleFactor \rangle \rightarrow \mathbf{E} \langle OptSign \rangle \langle Number \rangle$

11.2.7: The following productions define for-statements (only):

$\langle Statement \rangle \rightarrow$ **for variable :=** $\langle Expression \rangle$ **to** $\langle Expression \rangle$
do $\langle Statement \rangle$
$\langle Statement \rangle \rightarrow$ **for variable :=** $\langle Expression \rangle$ **downto** $\langle Expression \rangle$
do $\langle Statement \rangle$

◇ Section 11.3

11.3.1: The new strings for the languages S and L are tabulated in Fig. (a).

	S	L
Round 4:	`wcdwcdwcds`	`wcdwcds`
	`wcdbse`	`bse`
	`bs;wcdse`	`s;wcds;s`
	`bs;se`	`s;wcds;wcds`
	`bwcdse`	`s;wcds;wcdwcds`
		`s;wcds;bse`
		`s;s;s`
		`s;s;wcds`
		`s;s;wcdwcds`
		`s;s;bse`
		`wcds;s`
		`wcds;wcds`
		`wcds;wcdwcds`
		`wcds;bse`
		`s;wcdwcds`
		`s;bse`

(a) Words added on round 4.

11.3.3:

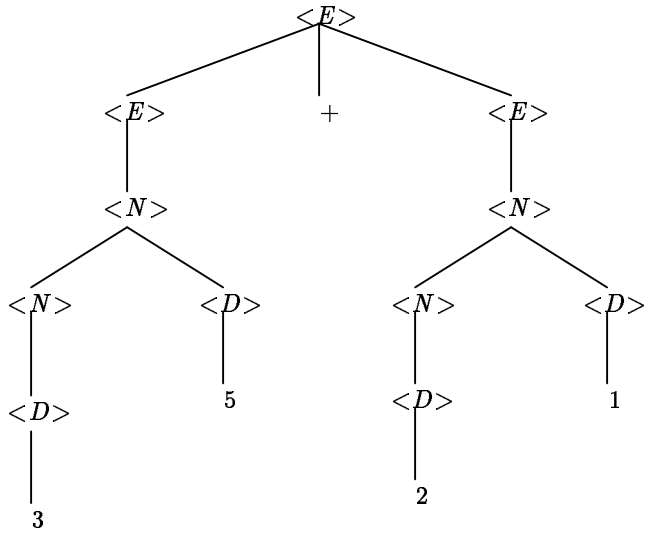
	Fig. 11.3	Fig. 11.4
Round 1:	ϵ	ϵ
Round 2:	`()`	`()`
Round 3:	`()()`	`(())`
	`(())`	`()()`
	`(())()`	

On round 3, the grammars generate different sets of strings. Thus, the answer to the question is "no." In fact, all all subsequent rounds the sets of strings generated by the two grammars are different. However, the sets of strings generated taken over all the rounds are the same; both sets are the set of all balanced parenthesis strings.

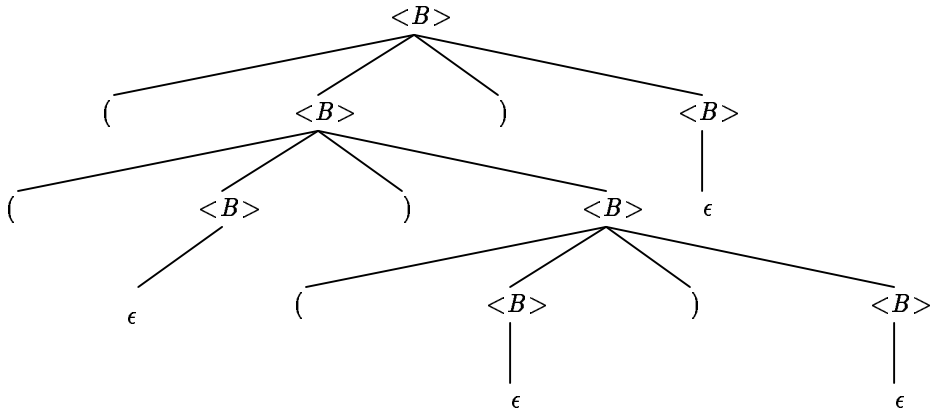
11.3.5: Suppose we are generating round r. If we make a substitution that uses only strings available on round $r-2$ or earlier, then the same substitution could have been made on round $r-1$. Thus, the string generated by this substitution must have appeared on round $r-1$ or on some round earlier than that.

◇ Section 11.4

11.4.1(a): The parse tree for `35+21` is shown in Fig. (b).



(b) Parse tree for `35+21`.



(c) Parse tree for `(()())`.

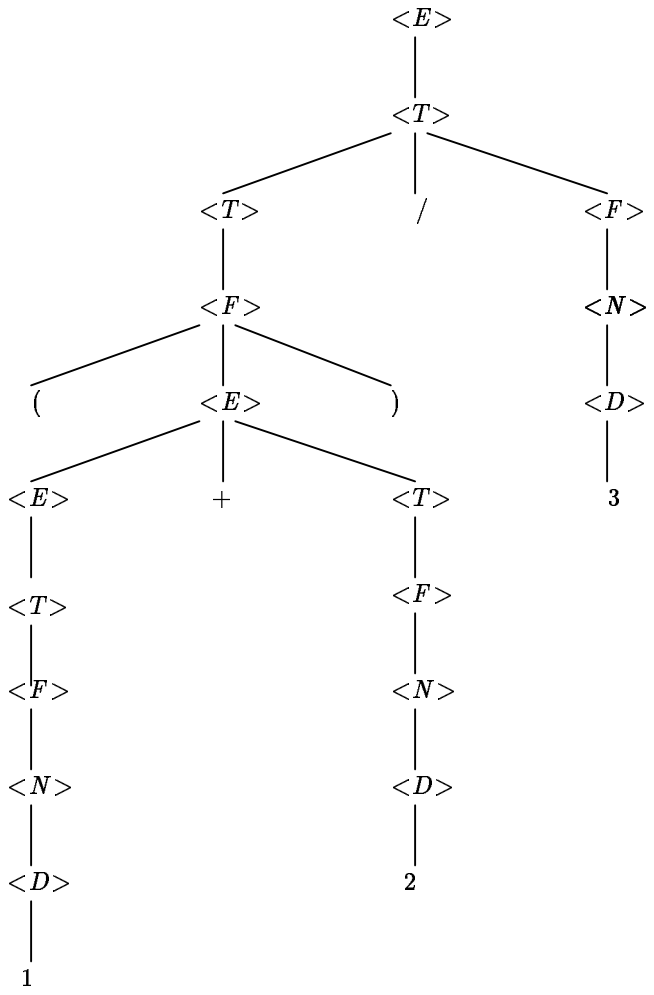
11.4.3(a): The parse tree for `(()())` is shown in Fig. (c).

♢ Section 11.5

11.5.1(a): The parse tree for `(1+2)/3` is shown in Fig. (d).

11.5.3: We introduce a new syntactic category, say <C> ("comparison"), which is either an expression or a comparison operator between two expressions. In line (3) of Fig. 11.22 of the text, we replace the productions for <F> by

$$<F> \rightarrow (<C>) \mid <N>$$



(d) Parse tree for `(1+2)/3`.

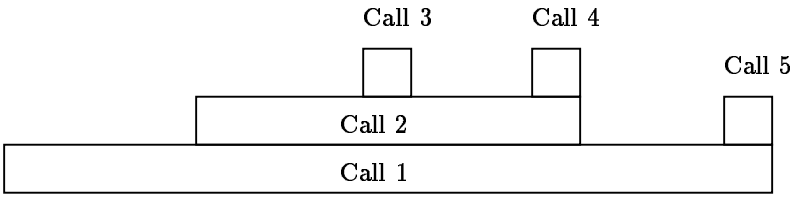
Then, we introduce the productions for $<C>$:

$<C> \rightarrow <E> <Cop> <E> \mid <E>$
$<Cop> \rightarrow = \mid <> \mid <= \mid >= \mid < \mid >$

11.5.7: The string `010` has two parse trees, one in which the first two characters are grouped first into a string, and the other in which the last two characters are grouped first.

11.5.9: There are an infinite number of parse trees for the empty string. We can replace one $<B>$ by two $<B>$'s as often as we like and then make each of the $<B>$'s be replaced by the empty string.

(( )) ENDM



(e) Sequence of calls made on input (()).

◇ Section 11.6

11.6.1(a): The structure of the calls is shown in Fig. (e).

11.6.3: In the first case, the productions

$$<Number> \rightarrow <Digit> <Number> \mid <Digit>$$

when we see a $<Digit>$ as the next input, there is no way to tell which production to use, so the grammar is not parsable by a recursive descent parser.

In the seond case, the productions

$$<Number> \rightarrow <Number> <Digit> \mid \epsilon$$

when we see a $<Digit>$ we cannot tell how many times to apply the first production, so we go into an infinite loop. That is, when the lookahead symbol is a digit, and we have to expand $<Number>$, we must pick the first production, and we are then faced with the same situation we started with: a digit as lookahead with $<Number>$ as the syntactic category to expand.

◇ Section 11.7

11.7.1(a):

	STACK	LOOKAHEAD	REMAINING INPUT
1)	$<S>$	b	seENDM
2)	b$<L>$e	b	seENDM
3)	$<L>$e	s	eENDM
4)	$<S><T>$e	s	eENDM
5)	s$<T>$e	s	eENDM
6)	$<T>$e	e	ENDM
7)	e	e	ENDM
8)	ϵ	ENDM	ϵ

	STACK	LOOKAHEAD	REMAINING INPUT
1)	$<S>$	b	bs;se;seENDM
2)	b$<L>$e	b	bs;se;seENDM
3)	$<L>$e	b	s;se;seENDM
4)	$<S><T>$e	b	s;se;seENDM
5)	b$<L>$e$<T>$e	b	s;se;seENDM
6)	$<L>$e$<T>$e	s	;se;seENDM
7)	$<S><T>$e$<T>$e	s	;se;seENDM
8)	s$<T>$e$<T>$e	s	;se;seENDM
9)	$<T>$e$<T>$e	;	se;seENDM
10)	;$<L>$e$<T>$e	;	se;seENDM
11)	$<L>$e$<T>$e	s	e;seENDM
12)	$<S><T>$e$<T>$e	s	e;seENDM
13)	s$<T>$e$<T>$e	s	e;seENDM
14)	$<T>$e$<T>$e	e	;seENDM
15)	e$<T>$e	e	;seENDM
16)	$<T>$e	;	seENDM
17)	;$<L>$e	;	seENDM
18)	$<L>$e	s	eENDM
19)	$<S><T>$e	s	eENDM
20)	s$<T>$e	s	eENDM
21)	$<T>$e	e	ENDM
22)	e	e	ENDM
23)	ϵ	ENDM	seENDM

Fig. (f)

11.7.1(c): See Fig. (f).

11.7.5: We factor the first two productions to get

$<Statement> \rightarrow$ **if** *condition* **then** $<Statement>$ $<Tail>$
$<Statement> \rightarrow$ *simpleStat*

$<Tail> \rightarrow$ **else** $<Statement> \mid \epsilon$

When $<Statement>$ is on top of the stack, the lookahead symbol, **if** or *simpleStat*, tells us which production for $<Statement>$ to use. When we need to expand a $<Tail>$, we make the first choice on lookahead **else** and the second choice (ϵ) on any other lookahead.

◇ Section 11.8

11.8.1(a):

$<A> \rightarrow \mathbf{a}$
$<B> \rightarrow \mathbf{b}$
$<C> \rightarrow <A> \mid <B>$
$<D> \rightarrow <C> <D> \mid \epsilon$
$<E> \rightarrow <D> <A>$

11.8.1(c):

$<A> \rightarrow \mathbf{a}$
$<B> \rightarrow \mathbf{b}$
$<C> \rightarrow \mathbf{c}$
$<D> \rightarrow <A> <D> \mid \epsilon$
$<E> \rightarrow <B> <E> \mid \epsilon$
$<F> \rightarrow <C> <F> \mid \epsilon$
$<G> \rightarrow <D> <E>$
$<H> \rightarrow <G> <F>$

11.8.3: If L were defined by a regular expresion, then it would also be defined by a finite automaton. Suppose the language $L = \{0^n 10^n \mid n \geq 0\}$ is the language of some finite automaton A. Let A have m states. Consider what happens when A has input $0^m 10^m$. This string is in the language L, so there is a path with label $0^m 10^m$ from the start state of A to some final state f. Consider the first $m+1$ states along this path. As A has only m different states, there will be two numbers of 0's, say i and j, with $0 \leq i < j \leq m$, such that after following i 0's and again after following a total of j 0's, A is in the same state, say s.

Now, consider what happens when the input to A is $0^{m-j+i}10^m$. The first i 0's get us to state s. The remainder of the input, $0^{m-j}10^m$ takes us to state f, because we know that when the input was $0^m 10^m$, A went from state s to state f after reading the first j 0's. Thus, A accepts $0^{m-j+i}10^m$, which is not in L, since $j > i$. We contradict our assumption that A accepts language L. Since we assumed nothing but that A did accept L, we conclude that no automaton accepts L. Hence, L cannot be accepted by a regular expression.

Chapter 12. Propositional Logic

◇ Section 12.3

12.3.1(a): In this and the next answer we use 0 for `FALSE` and 1 for `TRUE`. The function has domain consisting of pairs of truth values, for p and q respectively, and a range that is a truth value. This function can therefore be represented as the set $\{\big((0,0),0\big),\ \big((0,1),0\big),\ \big((1,0),1\big),\ \big((1,1),1\big)\}$.

12.3.1(c): $\{\big((0,0),1\big),\ \big((0,1),0\big),\ \big((1,0),0\big),\ \big((1,1),1\big)\}$.

◇ Section 12.4

12.4.1(a): A row has 1 unless both of the given columns have 1 in that row.

12.4.1(c): A row has 1 if the two given columns agree in the row, and 0 if not.

12.4.3: The logical expression p `AND NOT` q corresponds to the set expression $P-Q$.

12.4.5: Here are the 16 Boolean functions of two variables.

p	q	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Functions f_0, f_5, f_{10}, and f_{15} do not depend on the first argument. That is, these columns agree in the rows for $pq = 00$ and $pq = 10$, and they also agree in the rows for $pq = 01$ and $pq = 11$. Functions f_0, f_3, f_{12}, and f_15 do not depend on their second argument.

12.4.7:

STATEMENT $S(b)$: There are a^b ways to paint b houses using a colors.

BASIS. $b = 1$. There are a colors for one house.

INDUCTION. Assume $S(b)$ and prove $S(b+1)$. Consider the $(b+1)$st house. For each color choice for this house, there are, by the inductive hypothesis, a^b ways to paint the remaining houses. Thus there are $b \times a^b = a^{b+1}$ color choices for the $b+1$ houses.

◇ Section 12.5

12.5.1: $a = \bar{p}\bar{q}r + p\bar{q}\bar{r} + p\bar{q}r + pq\bar{r} + pqr$; $b = \bar{p}\bar{q}\bar{r} + \bar{p}\bar{q}r + \bar{p}q\bar{r}$.

12.5.3(a): Suppose we have an expression involving two variables p and q, and we try to construct other functions of p and q by applying the $\equiv$ operator to two previously constructed columns of a truth table. It turns out that the columns we can obtain are rather limited, as the following table shows.

p	q	$p \equiv q$	$p \equiv p$
0	0	1	1
0	1	0	1
1	0	0	1
1	1	1	1

We notice that if we combine any of the four rows above with the $\equiv$ operator, we get one of the same four rows. Thus, none of the other 12 functions of two variables can be expressed with only the $\equiv$ operator, and therefore this operator is not complete. For example, it cannot express p `AND` q.

12.5.3(c): Using `NOR`, we can express `NOT` p as p `NOR FALSE`. Also, p `OR` q is (p `NOR` q) `NOR FALSE`; p `AND` q is (p `NOR FALSE`) `NOR` (q `NOR FALSE`). Thus, `NOR` is complete.

12.5.5: Consider two monotone functions f and g of variables $p_1, p_2, \ldots, p_n$. Also, consider a truth table with 2^n rows and columns for f and g. Let r_1 and r_2 be rows of this table, and suppose that r_2 has 1 for any variable p_i whenever r_1 has 1 for p_i. Then the monotonicity of f says that if f is 1 in row r_1, then it must also be 1 in row r_2; a similar statement holds for g. We must show that if the column for f `AND` g has 1 in row r_1, then it also has 1 in row r_2, and similarly for the column for f `OR` g.

The only way f `AND` g can be in row r_1 1 is if both f and g to be 1 in this row. But then, both f and g have 1 in row r_2, and therefore f `AND` g has 1 there.

Now consider f `OR` g. This column can have 1 in row r_1 if either f or g or both have 1 there. But then, at least one of f and g have 1 in row r_2, and so does f `OR` g.

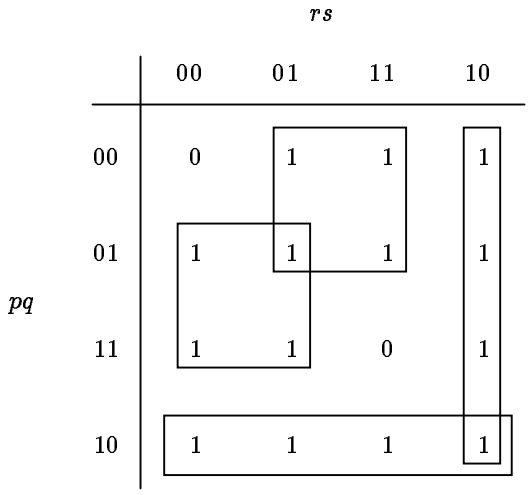
◇ Section 12.6

12.6.1(a): The Karnaugh map is shown in Fig. (a).

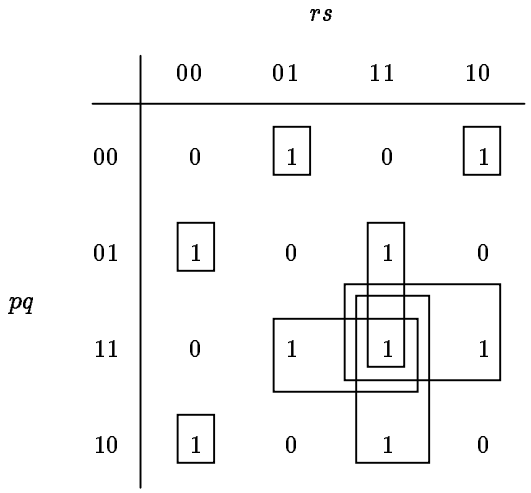
12.6.1(c): The Karnaugh map is shown in Fig. (c).

12.6.1(e): The Karnaugh map is shown in Fig. (e).

12.6.3: First, a product is of the right form to be an implicant; it is a product of literals. The function f represented by this hypothetical sum-of-products expression can be written $f = P + E$, where P is the product in question, and E is the sum of all the other products. Whenever an assignment of truth values to the variables makes P true, it surely makes $P + E$ true, by the definition of "or." Thus, f is true whenever the product P is true, which means P is an implicant of f.



(a) Karnaugh map for Exercise 12.6.1(a).



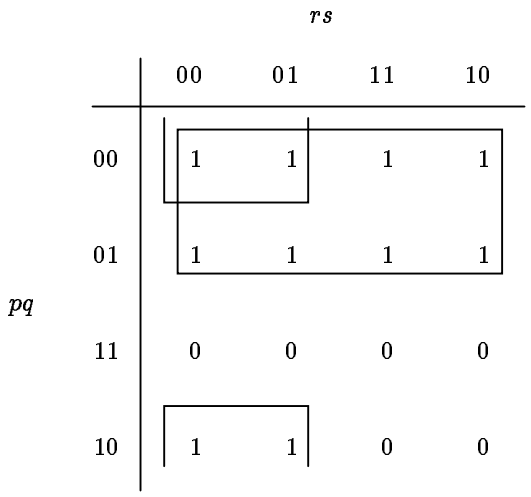
(c) Karnaugh map for Exercise 12.6.1(c).

12.6.5:

(a) $(p+q+r+s)(\bar{p}+\bar{q}+\bar{r}+\bar{s})$

(c) $(p+q+r+s)(p+q+\bar{r}+\bar{s})(p+\bar{q}+r+\bar{s})(p+\bar{q}+\bar{r}+s)(\bar{p}+\bar{q}+r+s)(\bar{p}+q+\bar{r}+s)(\bar{p}+q+r+\bar{s})$

(e) $(\bar{p}+\bar{q})(\bar{p}+\bar{r})$



(e) Karnaugh map for Exercise 12.6.1(e).

◇ Section 12.7

12.7.1: (a) $pqr \rightarrow p + q$ is a tautology; (c) $(p \rightarrow q) \rightarrow p$ is not a tautology. In particular, expression (c) is false when p is true and q is false.

◇ Section 12.8

12.8.1: As an example, here is the truth table for (12.4).

p	q	$p \equiv q$	$\bar{p} \equiv \bar{q}$	$(p \equiv q) \equiv (\bar{p} \equiv \bar{q})$
0	0	1	1	1
0	1	0	0	1
1	0	0	0	1
1	1	1	1	1

12.8.5(a):

1)	From (12.15)	$(1 + q) \equiv 1$
2)	From (12.10)	$(p1) \equiv p$
3)	Substitute $1 + q$ for 1 (line 1)	$\big(p(1 + q)\big) \equiv p$
4)	From (12.9)	$(p1 + pq) \equiv p$
5)	From (12.10)	$(p + pq) \equiv p$

12.8.7: We shall prove the following, which is (12.20c), by induction on k.

STATEMENT $S(k)$: $\big(\texttt{NOT}\ \ (p_1 p_2 \cdots p_k)\big) \equiv (\bar{p}_1 + \bar{p}_2 + \cdots + \bar{p}_k)$

BASIS. $k = 2$. $\texttt{NOT}\ (p_1 p_2) \equiv (\bar{p}_1 + \bar{p}_2)$ by (12.20a).

INDUCTION. We assume $S(k)$ and prove $S(k+1)$, which says that

$$\big(\texttt{NOT}\ (p_1 p_2 \cdots p_{k+1})\big) \equiv (\bar{p}_1 + \bar{p}_2 + \cdots + \bar{p}_{k+1})$$

To begin,

$$\big(\texttt{NOT}\ (p_1 p_2 \cdots p_{k+1})\big) \equiv \big(\texttt{NOT}\ (p_1 p_2 \cdots p_k) + \bar{p}_{k+1}\big) \tag{1}$$

by (12.20a), with $p_1 p_2 \cdots p_k$ in place of p and p_{k+1} in place of q.

By the inductive hypothesis, $\big(\texttt{NOT}\ (p_1 p_2 \cdots p_k)\big) \equiv (\bar{p}_1 + \bar{p}_2 + \cdots + \bar{p}_k)$. When we make this substitution in (1) and use the associative law of $+$, we get exactly $S(k+1)$.

We can also look at the proof from the point of view of truth tables. It is easy to observe that both sides of $S(k)$ have value 1 except when all of the p_i's are 1. The two proofs for (12.20d) have essentially the same ideas.

12.8.9: The question is ill formed in two ways. First, there is the matter of a typo; k should be n. More serious, there is a simple, noninductive proof of (12.24b), given (12.24a). By (12.24a), with $p_1 p_2 \cdots p_n$ in place of p, we have $(p_1 p_2 \cdots p_n \to q) \equiv \big(\texttt{NOT}\ (p_1 p_2 \cdots p_n) + q\big)$. By (12.20c) and the associative law of $+$, $(p_1 p_2 \cdots p_n \to q) \equiv (\bar{p}_1 + \bar{p}_2 + \cdots + \bar{p}_n + q)$.

12.8.11:

(a) $(w\bar{x} + w\bar{x}y + \bar{z}\bar{x}w) \equiv (w\bar{x} + \bar{z}\bar{x}w) \equiv (w\bar{x})$

(b) $\big((w + \bar{x})(w + y + \bar{z})(\bar{w} + \bar{x} + \bar{y})(\bar{x})\big) \equiv \big((w + \bar{x})(w + y + \bar{z})(\bar{x})\big) \equiv \big((w + y + \bar{z})(\bar{x})\big)$

♢ Section 12.9

12.9.1: It is evident that when we replace `AND` by `OR`, `OR` by `AND`, 0 by 1 and 1 by 0, we turn (12.25) into (12.27), and vice-versa. Thus, these expressions are duals of each other.

12.9.3: We use the following propositional variables with their intuitive meanings:

p: "x is a perfect square."
e: "x is even."
d: "x is divisible by 4."

We want to prove the theorem $pe \to d$. Since we are asked for a proof by contradiction, we want to show $\big(\texttt{NOT}\ (pe \to d)\big) \to 0$. The left side is equivalent to $pe\bar{d}$, so we can instead prove $pe\bar{d} \to 0$. This is as far as we can go using propositional logic alone. Now we must use some of the things we know about numbers.

We start with p, e, and $\bar{d}$ and derive a contradiction. Proposition p says that x is a perfect square, so $x = n^2$ for some integer n. If n is odd, then n^2 is odd. But we assume e, which says that $x = n^2$ is even. Thus, n is even. (This is a little proof by contradiction within the main proof.) If n is even, then $n = 2m$ for some integer m. Thus, $x = n^2 = 4m^2$. That says x is divisible by 4, or d. Since we also assumed $\bar{d}$, we have $d\bar{d}$, which is equivalent to 0. We have now proved $pe\bar{d} \to 0$.

12.9.5(a): In what follows, we use the associative and commutative laws of $+$ many times; we shall not make these uses explicit. $pq + r + \bar{q}\bar{r} + \bar{p}\bar{r}$ is equivalent to $pq + r + \bar{q} + \bar{p}\bar{r}$ by (12.19b) with r in place of p. That is equivalent to $pq + r + \bar{q} + \bar{p}$ for the same reason. Another use of (12.19b) transforms this expression to $p + r + \bar{q} + \bar{p}$. Now, (12.25) lets us replace $p + \bar{p}$ by 1. Finally, $(1 + r + \bar{q}) \equiv 1$ by (12.15).

12.9.7: Suppose 2^k cases are defined by the propositional variables $p_1, p_2, \ldots, p_k$, which may be true or false in any combination. The general case analysis law is

$$\left(\texttt{AND}_{i=0}^{2^k-1}\right) C_i \equiv q$$

where each C_i is of the form $x_1 x_2 \cdots x_k \rightarrow q$. Each x_j is either p_j or $\bar{p}_j$; it is p_j if the jth bit from the right in the binary integer i is 1, and it is $\bar{p}_j$ if that bit is 0.

For $k = 2$ we have

$$\left((p_1 p_2 \rightarrow q) \texttt{ AND } (p_1 \bar{p}_2 \rightarrow q) \texttt{ AND } (\bar{p}_1 p_2 \rightarrow q) \texttt{ AND } (\bar{p}_1 \bar{p}_2 \rightarrow q)\right) \equiv q$$

If q is false, then the left side of this equivalence is false for any truth assignment to the p's, as there must be one of the implications whose left side is true and whose right side (q) is false. Thus, the equivalence is true when q is false.

When q is true, each of the implications on the left must be true, because an implication cannot be false if its right side is true. Thus, the equivalence is again true, and we have proved it is a tautology. Note this proof applies to the general case as well as the case $k = 2$.

◇ Section 12.10

12.10.1(a):

1)	$p \rightarrow q$	Hypothesis
2)	$(p \rightarrow q) \equiv (\bar{p} + q)$	Law 12.24(a)
3)	$\bar{p} + q$	(d) with lines (1) and (2)
4)	$p \rightarrow r$	Hypothesis
5)	$(p \rightarrow r) \equiv (\bar{p} + r)$	Law 12.24(a)
6)	$\bar{p} + r$	(d) with (4) and (5)
7)	$(\bar{p} + q)$ AND $(\bar{p} + r)$	(c) with (3) and (6)
8)	$(\bar{p} + q)$ AND $(\bar{p} + r) \equiv (\bar{p} + qr)$	Law 12.14
9)	$\bar{p} + qr$	(d) with (7) and (8)
10)	$(\bar{p} + qr) \equiv (p \rightarrow qr)$	Law 12.24(a)
11)	$p \rightarrow qr$	(d) with (9) and (10)

12.10.1(b):

1)	$p \to (q + r)$	Hypothesis
2)	$\big(p \to (q + r)\big) \equiv (\bar{p} + q + r)$	Law 12.24(a)
3)	$\bar{p} + q + r$	(d) with (1) and (2)
4)	$p \to (q + \bar{r})$	Hypothesis
5)	$\big(p \to (q + \bar{r})\big) \equiv (\bar{p} + q + \bar{r})$	Law 12.24(a)
6)	$\bar{p} + q + r$	(d) with (4) and (5)
7)	$(\bar{p} + q + r)(\bar{p} + q + \bar{r})$	(c) with (3) and (6)
8)	$\big((\bar{p} + q + r)(\bar{p} + q + \bar{r})\big) \equiv (\bar{p} + q + r\bar{r})$	Law 2.14
9)	$\bar{p} + q + r\bar{r}$	(d) with (7) and (8)
10)	$(r\bar{r}) \equiv 0$	Law 12.27
11)	$\bar{p} + q + 0$	Substitution into (7), using (10)
12)	$\bar{p} + q$	Law 12.11
13)	$(\bar{p} + q) \equiv (p \to q)$	Law 12.24(a)
14)	$p \to q$	(d) with (12) and (13)

◇ Section 12.11

12.11.1:

p	q	r	$p+q$	$\bar{p}+r$	$(p+q)(\bar{p}+r)$	$q+r$	$\big((p+q)(\bar{p}+r)\big) \to (q+r)$
0	0	0	0	1	0	0	1
0	0	1	0	1	0	1	1
0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	1
1	0	0	1	0	0	0	1
1	0	1	1	1	1	1	1
1	1	0	1	0	0	1	1
1	1	1	1	1	1	1	1

12.11.3: The following clauses come from Exercise 12.11.2:

1)	$(a + c + \bar{t})$
2)	$(b + c + \bar{s})$
3)	$(\bar{a} + t)$
4)	$(\bar{b} + s)$
5)	$(\bar{c} + s)$
6)	$(\bar{c} + t)$
7)	$(a + b + c + o)$

We can then apply resolution to these clauses to derive the following. We show clauses that are needed to derive minimal clauses, even if they are not themselves minimal, but we do not show all the (nonminimal) clauses.

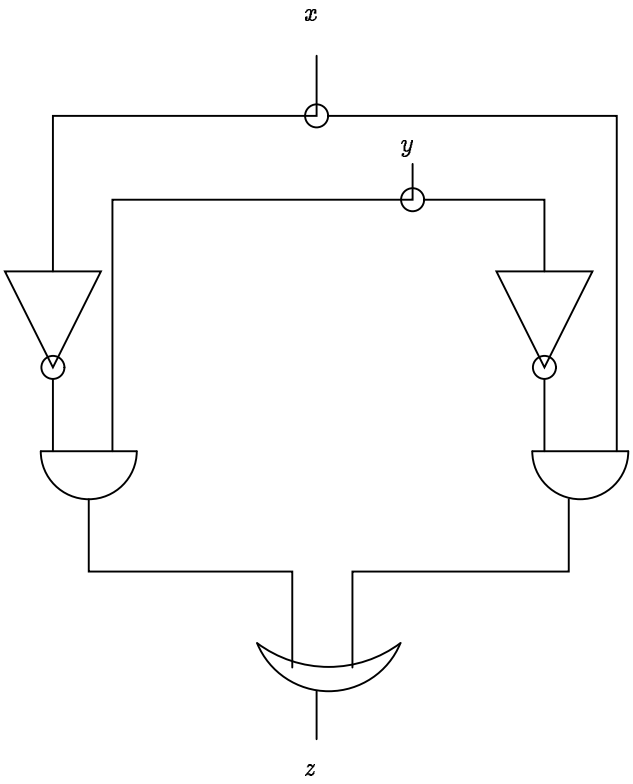
8)	$(a + s + \bar{t})$	From (1) and (5)
9)	$(b + \bar{s} + t)$	From (2) and (6)
10)	$(a + b + s + o)$	From (5) and (7)
11)	$(a + s + o)$	From (4) and (10)
12)	$(s + t + o)$	From (3) and (11)
13)	$(a + b + t + o)$	From (6) and (7)
14)	$(b + t + o)$	From (3) and (13)

Of these, only (8), (9), (11), (12), and (14) are minimal. These five clauses form the answer to the question.

Chapter 13. Using Logic to Design Components

♢ Section 13.3

13.3.1(a):



13.3.1(b): The solution is shown in Fig. (a).

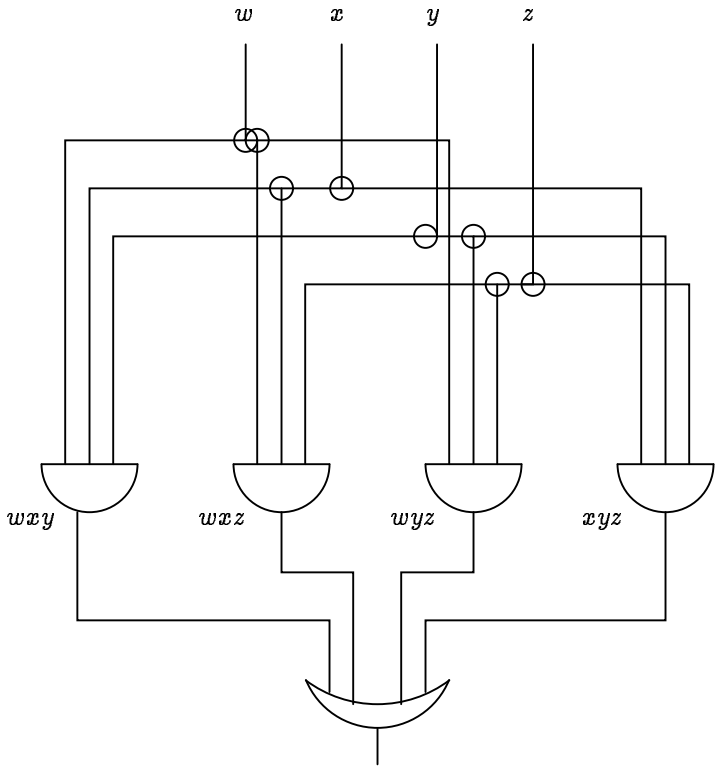
13.3.3: The output will become 1 as soon as one of x and y (or both) becomes 1. The output will then remain 1 no matter what happens to x and y.

♢ Section 13.4

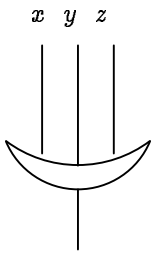
13.4.1(a): The solution appears in Fig. (b).

13.4.1(c): Note that $\big(x+\bar{y}\bar{x}(y+z)\big) \equiv (x+\bar{y}z)$. thus, the circuit in Fig. (c) serves.

13.4.3(b): Start with expression $\bar{x}yc + x\bar{y}c + xy\bar{c} + xyc$. Two uses of Law (12.17), the idempotence of `OR`, applied to xyc, gives us



(a) Solution to Exercise 13.3.1(b).



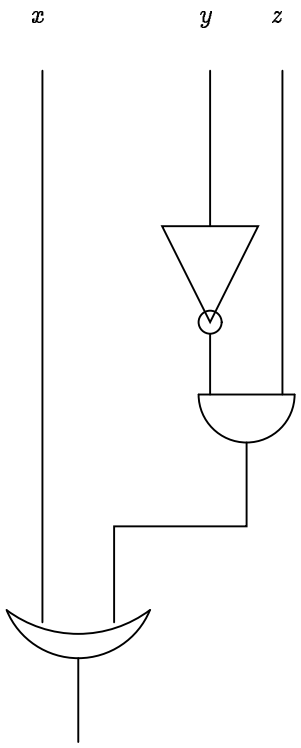
(b) Solution to Exercise 13.4.1(a).

$$\bar{x}yc + x\bar{y}c + xy\bar{c} + xyc + xyc + xyc$$

If we use the associative and commutative laws of `OR`, and then use the distributive law of `AND` over `OR`, we can rearrange these terms as

$$yc(\bar{x} + x) \;+\; xc(\bar{y} + y) \;+\; xy(\bar{c} + c)$$

Finally, three uses of the law of the excluded middle (12.25), transforms the above into $yc + xc + xy$.



(c) Solution to Exercise 13.4.1(c).

◇ Section 13.5

13.5.1: Using OR-gates with fan-in k, we can take the OR of n inputs with delay $\log_k n$, by using a complete k-ary tree of OR-gates. If we used a cascading circuit like that shown in Fig. 13.13 of the text, the delay would be $(n-k)/(k-1)+1$.

13.5.3: Let 2^k be the smallest power of 2 that is no less than n. Then we can take the OR of n inputs with k levels of 2-input OR-gates. That k levels is sufficient should be obvious. With that many levels, we can take the OR of 2^k inputs, which is at least n inputs. If n is strictly less than 2^k, we can set $2^k - n$ of the inputs to 0. That may let us eliminate some of the OR-gates. Elimination of gates cannot increase the number of levels.

Also, we cannot take the OR of n inputs in fewer than k levels. In $k-1$ levels, we can only take the OR of 2^{k-1} inputs, which is strictly less than n inputs, because we chose k so that 2^k is the smallest power of 2 equal to or greater than n.

◇ Section 13.6

13.6.5: We shall show by induction on n that

STATEMENT $S(n)$: If n is a power of 2, then $G(n) = 3n\log_2 n + 15n - 6$.

BASIS. $n = 1$. $G(1)$ is defined to be 9, and $S(1)$ says that $G(1) = 0 + 15 - 6 = 9$.

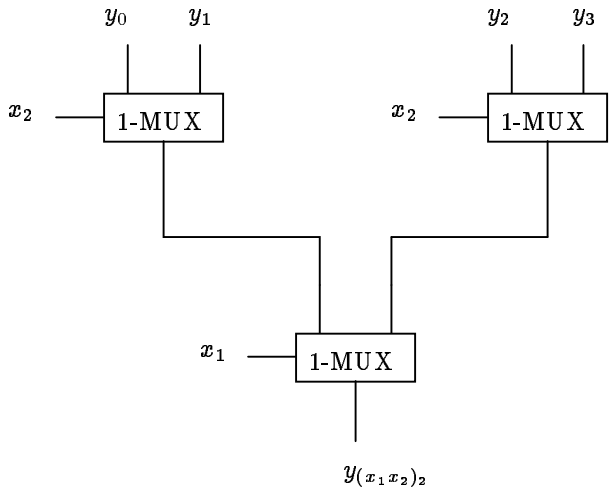
INDUCTION. We assume $S(n)$ and prove $S(2n)$. By definition, $G(2n) = 2G(n) + 6n + 6$. By the inductive hypothesis, $G(n) = 3n\log_2 n + 15n - 6$. Substituting this formula for $G(n)$ gives

$$G(2n) = 2(3n\log_2 n + 15n - 6) + 6n + 6 = 6n\log_2 n + 36n - 6$$

The above formula for $G(2n)$ is equal to $3(2n)\log_2(2n) + 15(2n) - 6$, which is the statement $S(2n)$.

◇ Section 13.7

13.7.1(a): A 2-MUX is constructed from 1-MUX's as follows.



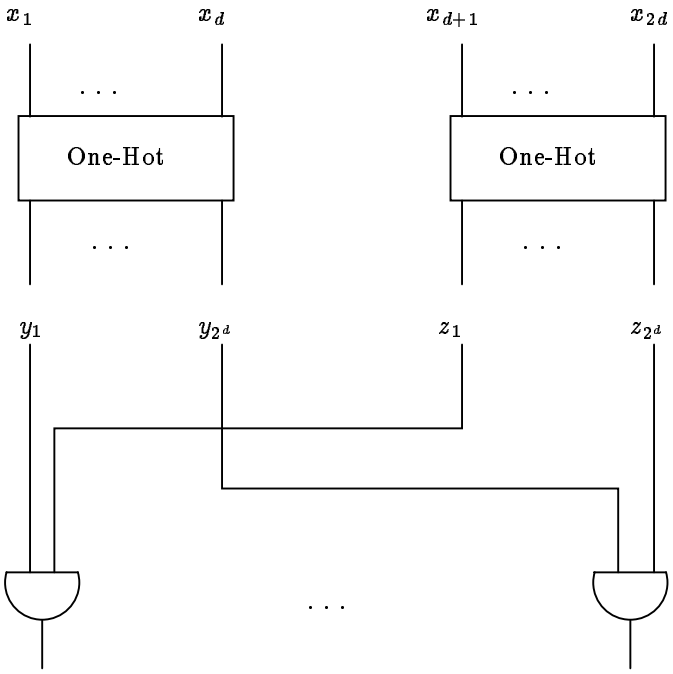
13.7.3: In Fig. (d) is the suggestion of a circuit that follows the second strategy of the hint. Two one-hot decoders for d inputs each are used. The first has inputs from the first d of $2d$ bits and the second has inputs from the last d bits. Their outputs, $y_1, \ldots, y_{2^d}$ and $z_1, \ldots, z_{2^d}$ are combined in all possible ways through AND-gates, to create 2^{2d} outputs, one for each possible setting of the $2d$ inputs.

Now, let us consider the gate count and delay for this circuit. For the case $d = 1$, there is an obvious basis circuit that uses only a single inverter; it has count 1 and delay 1. For the inductive step, the delay increases by only 1 going from d to $2d$ inputs. Thus, the delay for d inputs is easily seen to be $1 + \log_2 d$.

For the gate count, note that the circuit for $2d$ input uses twice the gates of the d-input circuit, plus 2^{2d} AND-gates at the last level. Thus, the recurrence for $G(d)$, the number of gates in the d-input circuits, is

BASIS. $G(1) = 1$.

INDUCTION. $G(2d) = 2G(d) + 2^{2d}$.



(d) Recursive construction of a one-hot decoder.

The solution to this recurrence, as we can show by repeated expansion, is

$$G(d) = d + 2^d + 2 \times 2^{d/2} + 4 \times 2^{d/4} + 8 \times 2^{d/8} + \cdots$$

We cannot find a convenient closed form for this series, but we note that 2^d is the dominant term, so $G(d)$ is slightly more than 2^d.

13.7.5: The trick is to compute the exact number of inputs that are 1. If the inputs are $x_1, \ldots, x_n$, then the outputs are $y_0, y_1, \ldots, y_n$, where y_i means that exactly i of the x's are 1.

BASIS. $n = 1$. $y_0 =$`NOT` x_1, and $y_1 = x_1$. Thus, a single gate and a delay of 1 suffice for the $n = 1$ case.

INDUCTION. Suppose we have a circuit for n inputs and we want one for $2n$ inputs. We take two copies of the n-input circuit, and feed half the inputs to each. We get outputs $u_0, u_1, \ldots, u_n$ from the first and $v_0, v_1, \ldots, v_n$ from the second. We can combine each u_i with each v_j, using $(n+1)^2$ `AND`-gates. The `AND`-gate for u_i and v_j is one way that $i + j$ of the $2n$ inputs can be 1.

If we have `OR`-gates that can take any number of inputs, we can `OR` together all the ways that k of the inputs can be 1. For example, zero 1's can only occur when both u_0 and v_0 are 1. That is, $y_0 = u_0 v_0$. One 1 can occur when either $u_0 = 1$ and $v_1 = 1$, or $u_1 = 1$ and $v_0 = 1$. That is, $y_1 = u_0 v_1 + u_1 v_0$. Similarly, $y_2 = u_0 v_2 + u_1 v_1 + u_2 v_0$. Then, $2n - 1$ `OR`-gates are needed, one for each but the y_0 and y_{2n} outputs.

If we restrict ourselves to 2-input `OR`-gates, then we need to combine the terms

in trees of OR-gates. As the middle output, y_n, has $n+1$ terms, we need $O(\log n)$ levels. We can calculate the exact number of 2-input OR-gates needed by the following trick. We note that there are $(n+1)^2$ inputs to the trees of OR-gates. Each 2-input OR-gate reduces the number of lines by 1, and at the output there are $2n+1$ lines left. Thus, there are exactly 2-input n^2 OR-gates.

Now, let us count the levels and gates, both for the case of 2-input OR-gates, and multi-input OR-gates. In the latter case, the recurrence for delay is

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

with a solution $D(n) = 2\log_2 n + 1$.

For 2-input gates, we need $O(\log n)$ levels when we double the number of inputs, so we get a recurrence of the form

$$D(1) = 1$$
$$D(2n) = D(n) + O(\log n)$$

lp The solution to this recurrence is $D(n) = O\big((\log n)^2\big)$.

For the gate count, assuming multi-input OR gates, we need $(n+1)^2$ AND-gates and $2n-1$ OR-gates, or n^2+4n gates in all. These are in addition to the gates in two subcircuits for n inputs each. The recurrence is

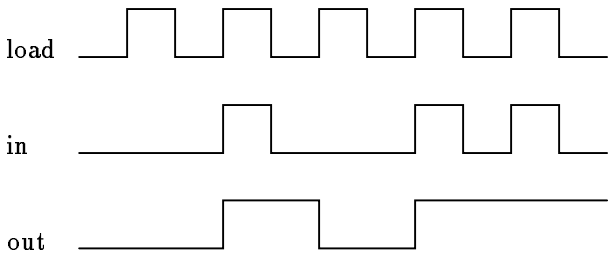
$$G(1) = 1$$
$$G(2n) = 2G(n) + n^2 + 4n$$

The solution is $G(n) = 2n^2 + n(4\log_2 n - 1)$.

Finally, if we use only 2-input OR-gates, then the gates used outside the two subcircuits is $2n^2 + 2n + 1$, but the solution to the recurrence for gate count is still $O(n^2)$.

◇ Section 13.8

13.8.1:



◇◇◇ Chapter 14. Predicate Logic

◇ Section 14.2

14.2.1:
a) CS205 is a variable
b) cs205 is a constant
c) 205 is a constant
d) "cs205" is a constant
e) $p(X, x)$ is a nonground atomic formula
f) $p(3, 4, 5)$ is a ground atomic formula
g) "$p(3, 4, 5)$" is a constant

◇ Section 14.3

14.3.1: A suitable logical expression is:

$$\big(csg(\text{"PH100"}, S, G) \texttt{ AND } snap(S, \text{"L. Van Pelt"}, A, P\big) \rightarrow answer(G)$$

answer is true when $G =$ C+. For this we use

$S = 67890$
$G =$ "C+"
$A =$ "34 Pear Ave."
$P =$ "555-5678"

◇ Section 14.4

14.4.1:

a) $(\forall X)(\exists Y)$ `NOT` $(p(X)$ `OR` $p(Y)$ `AND` $q(X))$

b) $(\exists X)(\texttt{NOT}\ p(X)$ `AND` $((\exists Y)p(Y)$ `OR` $(\exists X)q(X, Z)))$

14.4.3: $(\exists X)(\texttt{NOT}\ p(X)$ `AND` $((\exists Y)p(Y)$ `OR` $(\exists W)q(W, Z)))$

14.4.5:

a) $(\forall C)csg(C,$ "C. Brown", "A")

b) $(\exists C)$ `NOT` $csg(C,$ "C. Brown", "A")

◇ Section 14.5

14.5.1(a): Consider the interpretation I_1:

1. $D = \{a, b\}$
2. $loves(X, Y)$ is true if XY is one of aa, ab, bc, bb

Under this interpretation ("everyone loves everyone"), expression (a) is true.

Now consider the interpretation I_2:

1. $D = \{a, b\}$
2. $loves(X, Y)$ is true if XY is one of ba, bb

Under this interpretation ("a is a misanthrope), expression (a) is false.

14.5.1(b): Interpretation I_1:

1. $D = \{a\}$
2. $p(a)$ is true

Under I_1, expression (b) is true.

Interpretation I_2:

1. $D = \{a\}$
2. $p(a)$ is false

Under I_2, expression (b) is false.

14.5.1(c): Interpretation I_1:

1. $D = \{a\}$
2. $p(a)$ is true

Under I_1, expression (c) is true.

Interpretation I_2:

1. $D = \{a, b\}$
2. $p(a)$ is true, $p(b)$ is false

Under I_2, expression (b) is false because $p(a) \rightarrow (\forall X)p(X)$ is false.

14.5.1(d): Interpretation I_1:

1. $D = \{a, b, c\}$
2. $p(X, Y)$ is true if XY is one of ab, bc, ac

Under I_1, expression (d) is true.

Interpretation I_2:

1. $D = \{a, b, c\}$
2. $p(X, Y)$ is true if XY is one of ab, bc

Under I_2, expression (d) is false.

◇ Section 14.6

14.6.1:

a) $(r$ `OR` $s) \equiv (s$ `OR` $r)$ is a tautology in propositional logic (law 12.7). The predicate logic expression $(p(X)$ `OR` $q(Y)) \equiv (q(Y)$ `OR` $p(X))$ is derived by substituting $p(X)$ for r and $q(Y)$ for s.

b) $(r \texttt{ AND } s) \equiv r$ is a tautology in propositional logic (law 12.16). The expression $(p(X,Y) \texttt{ AND } p(X,Y)) \equiv p(X,Y)$ results by substituting $p(X,Y)$ for r.

c) $(r \rightarrow \texttt{FALSE}) \equiv \texttt{NOT } r$ is a tautology in propositional logic (law 12.24(a) with FALSE in place of q). The expression $(p(X) \rightarrow \texttt{FALSE}) \equiv \texttt{NOT } p(X)$ follows by substituting $p(X)$ for r.

◇ Section 14.7

14.7.1:

a) $(\exists X)\Big(\big(\texttt{NOT } p(X)\big) \texttt{ AND } \Big((\exists Y)(p(Y)) \texttt{ OR } (\exists W)\big(q(W,Z)\big)\Big)\Big)$

b) $(\exists X)((\exists Y)p(Y) \texttt{ OR } (\exists Z)q(Z) \texttt{ OR } r(X))$

14.7.3: Technically, law (14.12) does not allow us to change the binding of any variable occurrence. Thus, law (14.12) does not allow us to conclude

$$\big(p(X,Y) \texttt{ AND } (\forall X)q(X)\big) \equiv (\forall X)\big(p(X,Y) \texttt{ AND } q(X)\big)$$

However, the two expressions are equivalent for other reasons.