Programming Languages - Principles and Practice
2nd Edition

by Kenneth C. Louden
Thomson Brooks/Cole 2002

Answers to Selected Exercises

Chapter 1

1.2.

(a)

```
function numdigits(x: integer): integer;
var t,n: integer;
begin
  n := 1; t := x;
  while t >= 10 do begin
    n := n + 1;
    t := t div 10;
  end;
  numdigits := n;
end;
```

(c)

```
numdigits x = if x < 10 then 1 else numdigits(x / 10) + 1
```

(e)

```
function numdigits(x: Integer) return Integer is
t: Integer := x;
n: Integer := 1;
begin
  while t >= 10 loop
    n := n + 1;
    t := t / 10;
  end loop;
  return n;
end numdigits;
```

(g)

```
class NumDigits
{ public static int numdigits(int x)
  { int t = x, n = 1;
    while (t >= 10)
    { n++;
      t = t / 10;
    }
    return n;
  }
}
```

1.5. (a) The remainder function, which we shall write here as **%** (some languages use **rem** or **remainder**), is defined by the integer equation **a = (a / b) * b + a % b**. The **modulo** function, which we shall write here as **mod** (some languages use **modulo**) is defined by the properties

1. **a mod b** is an integer between **b** and **0** not equal to **b**, and
2. there exists an integer **n** such that **a = n * b + a mod b**.

When both **a** and **b** are non-negative, it can be shown that **a % b** = **a mod b**. But when either (or both) of **a** and **b** are negative these definitions can produce different results. For example, **–10 % 3** = **–1**, since **–10 / 3** = **–3** and **–3 * 3 – 1** = **–10**. However, **-1** is not between **0** and **3**, and indeed **–10 mod 3** = **2**, since **–10 = –4 * 3 + 2**. Some languages (C, Java) have only a remainder operation, some languages (Pascal, Modula-2) have only a modulo operation, and some languages (Ada, Scheme, Haskell) have both.

(b) Indeed, the above differences can cause the results of the gcd to differ by a sign. For example, the Ada implementation produces 5 as the gcd of –15 and 10, while the C implementation produces –5. For this reason (and because the sign of the gcd makes little sense), most languages with a built in gcd operation (like Scheme and Haskell) apply the absolute value to the operands before computing the gcd. Then it doesn't matter whether remainder or modulo is used.

1.10. Two possible things can happen when overflow occurs: either an interrupt or exception occurs, halting execution (if not handled), or the value "wraps", usually to a negative number (using two's complement format), and the factorial function appears to work but produces an erroneous value. The ANSI C Standard (see Kernighan and Ritchie [1988], p. 200) does not specify any particular behavior on overflow, but in C, the constant **INT_MAX** defined in the **limits.h** standard library header file can be used to perform a check:

```
int fact (int n)
{ if (n <= 1) return 1;
  else
  { int temp = fact(n – 1);
    if (temp < 0) return temp;
    else if (INT_MAX / n < temp) return –1;
    else return n * fact (n – 1);
  }
}
```

This code uses –1 as a return value indicating that overflow has occurred, but program execution is not halted. (It may appear that the test **INT_MAX / n < temp** is not a precise one, since **(INT_MAX / n) * n** is less than **INT_MAX** if **n** does not divide **INT_MAX**. Properties of integers guarantee, however, that we cannot have both **INT_MAX / n < temp** and **temp * n <= INT_MAX**.)

In languages other than C, different approaches may be necessary. For example, in Ada, overflow generates an exception which must be handled (see exception handling in Chapter 7). In Scheme, overflow cannot occur, since integer values can be arbitrarily large; other functional languages are similar. In Java, code similar to the C code above will work (with **INT_MAX** replaced by **Integer.MAX_VALUE**); note that Java *requires* that no interrupt or exception occur on integer overflow.

1.14. A string data type is predefined in Ada, C++, Java, Scheme, Haskell, and BASIC, but not in C, Pascal, or FORTRAN. That does not mean these latter languages do not have strings. For instance, C uses the data type **char *** as its string type. In Standard Pascal all types **packed array [1..n] of char** are considered string types. In FORTRAN77 string types are declared as **CHARACTER*N**, where **N** is a positive integer constant, as in

```
CHARACTER*80 LINE
```

which declares the variable **LINE** to be a string of 80 characters.

The predefined operations that come with the string data types are as follows, for selected languages in the above list.

Pascal: None, except for the usual operations of assignment and comparison. Lexicographic ordering is used for the comparison operators "<," "<=," ">," ">=." String constants, or literals, are given using single quotes, as in **`'This is a string'.`**

Ada: In addition to assignment and comparison, as in Pascal, Ada provides concatenation with notation "&", as in

```
"This is a " & "string"
```

C: C provides no operations that give the desired behavior for strings (note that usual comparison "= =" of two strings tests for identity of pointers, not equality of strings). However, C provides a standard library of string functions, including **`strcat`** (concatenation), **`strcmp`** (comparison), **`strlen`** (length, or number of characters), and other memory-oriented utilities (C does not automatically allocate memory for strings as do Pascal, Modula-2, and Ada).

FORTRAN: The FORTRAN77 standard provides for the usual comparisons and assignment (with truncation and blank padding for different-sized strings) and also for concatenation and substring operations. Concatenation is given by two forward slashes "//." Substrings are indicated by parenthesized constants separated by a colon, as in **`LINE (10 : 20`**). The predefined **`LEN`** function returns the length of a string.

Scheme: Scheme has a range of predefined string functions, including **`string-length, string-append`** (concatenation), **`string-substring, string=?`** (string comparison for equality), and **`string<? (string`** less than comparison).

1.17. The errors are as follows:

Line 2: The "pound" sign in **`u#`** is a lexical error, since it is not a legal character in C (unless it occurs in the first column of a line, in which case it indicates a preprocessor command that is replaced before compilation begins). (This is usually reported as a syntactic or parse error by a compiler.)

Line 2: The declaration of parameter **`v`** as a **`double`** is a static semantic error; it will be reported on line 4 as an invalid operand when applying the **`%`** operator.

Line 2: The semicolon at the end of the line is a syntactic error.

Line 3: The use of assignment instead of equality in the test of the if-statement is a logical error in C; it will, however, result in a division by zero runtime error at line 4, so it could also be reasonably classified as a dynamic semantic error.

Line 4: The use of **`#`** is a lexical error, as in line 2.

Line 10: The idenifier Gcd is mis-capitalized; this is a static semantic error that, however, will not be caught by the compiler but by the linker.

Line 11: there is a missing return value after **`return`** (usually 0), since **`main`** is implicitly declared to return an **`int`** in C. This is a static semantic error, which is usually reported as a warning only by C compilers, or even ignored completely.

1.20. The question really is one of whether a goto-statement exists and can be used to override the structuring of the if-statement, as in

```
main()
{ goto a;
  if (2 < 1)
    a: printf("ack!\n");
  else printf("ok.\n");
  return 0;
}
```

In Java no goto is available, so this is impossible. In Standard Pascal and Ada, it is also impossible, since a goto cannot jump inside a structured construct. In C and FORTRAN (permissive languages both), this is permitted.

1.22. The following answers are for C. Similar answers hold for Java and Ada.

(1) The value of a variable is dynamic since it can change during execution.
(2) The data type of a variable cannot change during execution; it is fixed during translation by its declaration.
(3) The name of a variable is also fixed by its declaration and cannot change during execution, except in the case of a dynamically allocated memory location without a name of its own, which can be accessed using different pointer variable names at different times, as for example in

```
int* p;
int* q;
p = (int*) malloc(sizeof(int));
*p = 2;
q = p;
*q = 1;
```

After the assignment `q = p`, the same (dynamically allocated) variable can be accessed using the names `*p` and `*q`. (An alternative view would say that the variable *accessed* using the *operations* `*p` and `*q` has in fact no name of its own, so we cannot say that its name has changed.)

1.27 It is very easy to write imperative-style code in C++, since C is essentially embedded in C++. But Java, too, allows imperative-style code to be written: an imperative program can be turned into a Java program simply by surrounding all the functions with a class declaration and declaring all the functions to be static (thus, they do not participate in object-oriented mechanism – see Chapter 10). For example, the C program of Figure 1.7 can be turned directly into the following Java program:

```
import javax.swing.JOptionPane;

class Gcd
{
  static int gcd(int u, int v)
  { if (v == 0) return u;
    else return gcd (v, u % v);
  }

  public static void main(String[] args)
  { int x, y;
    x = Integer.parseInt(
          JOptionPane.showInputDialog("Please input an integer"));
    y = Integer.parseInt(
          JOptionPane.showInputDialog("Please input an integer"));
```

```
        System.out.println("The gcd of "+x+" and "+y+" is "+gcd(x,y));
        System.exit(0);
      }
    }
```

Here the only problem is the input – Java is not set up to do console input as readily as C or C++. In this example we have elected to use a small window utility – **`JOptionPane`** – rather than define the necessary console input objects. (The **`System.exit(0)`** call at the end is a consequence of using **`JOptionPane`**: when windows are involved, Java will not end execution normally, since a window may still be executing.) Note that this program contains no objects (in the object-oriented sense) and is not object-oriented.

The question of to what extent functional-style code is possible in C++ and Java is a little more involved. As a general rule, when only built-in data types are required in a program, both Java and C++ can imitate functional programs quite well, since unrestricted recursion is built-in in both languages. Difficulties arise, however, with the use of user-defined data. See more on this issue in Section 11.2.

Chapter 2

2.1. (a) The cistern is assumed to be a rectangular solid with volume length × width × height. Let L = length and W = width. Since height = length, the volume V is given by $V = L^2W$. The cross-sectional area $A = LW$, and $A + V = 120$. Substituting gives

$$LW + L^2W = 120$$

and solving for W gives

$$W = 120 / (L + L^2)$$

or

$$W = 120 / L\,(1 + L)$$

The algorithm performs the division of 120 by $1 + L$ first, and then the division by L, so the precise steps of the algorithm are specified by a left-to-right evaluation of the formula

$$W = 120/(1 + L)/L$$

(b) A C function that returns the width given the length and area plus volume is as follows:

```
double bab(double length, double sum)
{ double temp = sum / (length + 1);
  return temp / length;
}
```

Whether this is really easier to understand than the Babylonian description depends on the reader. Those knowledgeable about computers and C may find the code easier to read, while others may find the Babylonian version preferable. Note that the steps described are the same, however.

2.5. JOVIAL—Jules' Own Version of the International Algorithmic Language: Developed by Jules Schwartz at SDC Corporation, based on Algol58, a predecessor to Algol60. Used by the U.S. Air Force as its standard language for many years, only to be replaced by Ada. See Shaw [1963].

Euler: A language designed by Niklaus Wirth in the 1960s in which he developed some of his ideas on simplicity in language design; a predecessor of both Algol-W and Pascal. See Wirth and Weber [1966a,b].

BCPL: A systems programming language developed in England in the late 1960s, it was a strong influence on the C language. See Richards and Whitby-Stevens [1979].

Alphard: A language developed at Carnegie Mellon University in the late 1970s incorporating ideas on abstract data types similar to Euclid and CLU. See Wulf, London, and Shaw [1976] and Shaw [1981].

HOPE: An experimental functional language developed at Edinburgh University in the late 1970s. Many of its ideas were incorporated into ML. See Burstall, MacQueen, and Sanella [1980].

2.7. Here are a few ways of determining dates of origin:

1. Date language development began
2. Date of first publication about the language
3. Date first translator became available outside the development team
4. Date of publication of first language definition
5. Date of first use of the language outside the development team
6. Date of first commercially available translator

The criterion for date of origin is important for the existence question posed in Exercise 2.6 in at least two ways. First, a language can exist only after its date of origin, so the criterion for date of origin must be satisfied by the criterion for existence. Second, the disappearance of a language can be judged by the same conditions. For example, if there are no more commercially available translators, or the language is no longer in use outside the development team, the language can be said to no longer exist.

2.14.(a) The difference is that the mathematical definition is not an algorithm, in the sense that it provides no direct construction of the gcd, but is just a property that the gcd must satisfy, which may require the checking of a possibly infinite set of numbers.
(b) Since the given definition is not an algorithm, it cannot be used directly to program a computation of the gcd. Certain additional properties of numbers can, however, be used to reduce the computation involved in the definition to a manageable size. For example, if we use the property that any divisor of both *u* and *v* must be between 1 and the min of *u* and *v*, we can check the given property for every number between 1 and min(*u,v*), until success is reached, as for example, in the following C function:

```
int gcd (int u, int v)
{ int min, i, j, done, found;
  if (u <= v) min = u; else min = v;
  i = min; found = 0;
  while (!found && i > 1)
    if (u % i == 0 && v % i == 0)
    { j = min; done = 0;
      while (j > 1 && !done)
      { if (u % j == 0 && v % j == 0)
          if (i % j != 0) done = 1;
          else j--;
        else j--;
      }
      if (j == 1) found = 1;
      else i--;
    }
    else i--;
  return i;
}
```

Of course, this algorithm searches for the gcd in a particular order, namely, from min (*u*, *v*) down to 1 (indeed, by using the further property that the gcd is in fact the largest number that divides both *u* and *v*, we could eliminate the inner while-loop altogether). Thus it cannot be said to "implement" the definition, even though it is closer to the spirit of the definition than Euclid's algorithm (it is also enormously less efficient than Euclid's algorithm).

(c) Mathematics is not always concerned with the *construction* of things, but sometimes only with their existence. And even when a possible construction is given (as with constructions of the real numbers), the processes used take potentially infinitely many steps (as with limits). Programs can express only those aspects of mathematics that are concerned with finite constructive quantities and properties. (See the mention of constructive methods in the text.)

Chapter 3

3.3. In C and Java, any loop can be exited at any time by using a **`break`** statement (Java also has a labeled **`break`**, which we do not discuss here); the **`exit`** command is used for program termination. Unfortunately, the **`break`** statement is also used in the **`switch`** statement to prevent automatic fall-through (see Chapter 7), and this often leads to the erroneous use of **`break`** to exit if-statements at arbitrary points, often with unpleasant consequences (if the if-statement is enclosed in a loop, it will exit the loop, rather than generate a compile-time error as it should). Thus, the **`break`** statement has non-uniform semantics in C and Java.

3.4. (a) This is a nonorthogonality, since it is an interaction between assignment and the datatypes of the subjects of the assignment. It definitely cannot be viewed as a nongenerality, since it makes no sense for assignment to be so general as to apply to all cases (assignment should only apply when data types are comparable in some sense). It also can't be labeled a nonuniformity, since we are not comparing two different constructs.

(b) This is a security issue, since assignment of a real (double or float) to an integer results in automatic truncation, which could result in incorrect execution.

3.8. **Readability**. Requiring the declaration of variables forces the programmer to document his/her expectations regarding variable names, data types, and scope (the region of the program where the variable will be applicable). Thus, the program becomes much more readable to the programmer and to others.

Writability. Requiring the declaration of variables may actually decrease writability in its most direct sense, since a programmer cannot simply use variables as needed, but must write declarations in their appropriate places to avoid error messages. This increased burden on the programmer can increase programming time. On the other hand, without declarations there can be no local variables, and the use of local variables can increase writability by allowing the programmer to reuse names without worrying about non-local references. Forcing the programmer to plan the use of variables may also improve writability over the long run.

Efficiency. As we saw, readability and writability can be viewed as efficiency issues from the point of view of maintenance and software engineering, so the comments about those issues also apply here in that sense. The use of declarations may also permit more efficient implementation of the program. Without declarations, if no assumptions are made about the size of variables, less efficient access mechanisms using pointers must be used. Also, the programmer can use declarations to specify the exact size of variable needed (such as **short int** or **long int**). Restricting scope by using local variables can also save memory space by allowing the automatic deallocation of variables. Note, however, that Fortran is a very efficient language in terms of execution speed, so it is not always true that requiring declarations must improve execution speed. Also, speed of translation may actually be decreased by the use of declarations, since more information must be kept in tables to keep track of the declarations. (It is not true, as Fortran and BASIC attest, that without declarations a translator must be multi-pass.)

Security. Requiring declarations enhances the translator's ability to track the use of variables and report errors. A clear example of this appears in the difference between ANSI C and old-style Unix C. Early C did not require that parameters to functions be declared with function prototypes. (While not exactly variable declarations, parameter declarations are closely related and can be viewed as essentially the same concept.) This meant that a C compiler could not guarantee that a function was called with the appropriate number or types of parameters. Such errors only appeared as crashes or garbage values during program execution. The use of parameter declarations in ANSI C greatly improved the security of the C language.

Expressiveness. Expressiveness may be reduced by requiring the declaration of variables, since they cannot then be used in arbitrary ways. Scheme, for example, while requiring declarations, does not require that data types be given, so that a single variable can be used to store data of any data type. This increases expressiveness at the cost of efficiency and security.

3.10. C has the same problems with semicolons as C++ — indeed, C++ inherited them from C. Thus, in C, we must always write a semicolon after a `struct` declaration:

```
struct X { int a; double b; } ; /* semicolon required here */
```

but never after a function declaration:

```
int f( int x) { return x + 1; } /* no semicolon */
```

The reason is C's original definition allowed variables to be declared in the same declaration as types (something we would be very unlikely to do nowadays):

```
struct X { int a; double b; } x;
          /* x is a variable of type struct X */
```

In addition to this nonuniformity of semicolon usage, C (and C++) have at least one additional such nonuniformity: semicolons are used as *separators* inside a for-loop specifier, rather than as terminators:

```
for (i = 0; i < n; i++ /* no semicolon here! */ )
```

3.14. Readability: Ada's comment notation is difficult to confuse with other constructs, and the comment indicators are always present on each comment line. By contrast, a C comment may have widely separated comment symbols, so it may not be easy to determine what is a comment and what is not (especially noticeable if a comment extends over more than one video screen). Embedded C comments may also be confusing, since `/` and `*` are arithmetic operators:

```
2 / 3 /* this is a comment */
2 / 3 / * this is an error */
```

Nested comments can also present readability problems in C:

```
/* A comment
      /* a nested comment
  ...
  but only one comment closer */
```

Thus Ada comments may be judged more readable than C's.

Writability: Ada's comments require extra characters for each new line of comments. This makes it more difficult to write an Ada comment, if only from a count of the number of extra characters required. C's comments, on the other hand, can be written more easily with a single opening and closing character sequence.

Reliability: A more readable comment convention is likely to be more reliable, since the reader can more easily determine errors, so Ada is likely to be more reliable in its comment convention. The main feature of Ada comments that perhaps increases their reliability is their *locality of reference:* all comments are clearly indicated locally, without the need for a proper matching symbol farther on. The nested comment issue in C, mentioned above, is also a source of errors, since more than one comment closer will result in compiler errors that are difficult to track down. Thus, C's comment convention is less reliable than Ada's.

C++ Comment Convention: C++ cannot use the Ada convention of a double-dash, since it is already in use as a decrement operator, and a translator would have no way of guessing which use was meant.

3.21. An obvious advantage of arbitrary-precision integers is that it frees the behavior of integers from any dependence on the (implementation-dependent) representation of the integers, including elimination of the need for considering overflow in the language definition (see Exercise 1.10). The disadvantage is that the size of memory needed for an integer is not static (fixed prior to execution), and therefore memory for an integer must be dynamically allocated. This has serious consequences for a language like C. For example, in the following code,

```
struct X { int i; char c; } x;
   ...
x.i = 100;
x.c = 'C';
  ...
x.i = 100000000000000000;
...
```

the allocation of new storage for **x** on the second assignment to **x.i** means **x.b** must also be reallocated and copied, unless indirection is used. Indeed, a reasonable approach would be to make integer variables into pointers and automatically allocate and deallocate them on assignment. This means that the runtime system must become "fully dynamic" (with a garbage collector), substantially complicating the implementation of the language. The arithmetic operators, such as addition and multiplication, also become much less efficient, since a software algorithm must be used in place of hardware operations.

In principle, a real number with arbitrary precision can be represented in the same way as an arbitrary-precision integer, with the addition of a distinguished position (the position of the decimal point). For example, 33.256 could be represented as (33256,2), the 2 expressing the fact that the decimal point is after the second digit. (Note that this is like scientific notation, with the 2 representing a power of 10: $33.256 = .33256 * 10^2$.) The same comments hold for such reals as for arbitrary-precision integers. However, there is a further complication: while integer operations *always* result in a finite number of digits, real operations can result in infinitely many digits (consider the result of 1.0/3.0 or sqrt(2.0)). How many digits should these results get? Any answer is going to have to be arbitrary. For this reason, even systems with arbitrary-precision integers often place restrictions on the precision of real numbers. (Scheme calls any number with a decimal point *inexact*, and any time an integer—which is exact—is converted to a real, it becomes inexact, and some of its digits may be lost.)

Chapter 4

4.2. A sample test in C (or Java) would insert a comment in the middle of a reserved word, such as in

```
in/*comment*/t x;
```

This will in fact produce an error, transforming `int` into two tokens `in` and `t` (usually interpreted as identifiers by a compiler). In Ada and FORTRAN, such a comment cannot be written. In Ada, all comments begin with two adjacent hyphens and extend up to the next newline. Thus, Ada comments by design can only occur as part of white space (a newline). In FORTRAN comments can only be entire lines, so a similar remark holds.

4.6. (a) The problem is that allowing signed integer constants creates an ambiguity in recognizing tokens that a scanner cannot resolve. For example, the string **`2-3`** should have three tokens: a **`NUMBER`**, a **`MINUS`**, and a **`NUMBER`**, and the string **`2--3`** should *also* have the same three tokens (assuming signed constants are legal). This ambiguity can only be resolved by the parser, thus creating a serious problem in writing an independent scanner.
(b) There are several possible solutions. Perhaps the most common is to have the scanner always recognize a minus sign as a separate token, and then extend the grammar to allow the parser to recognize leading minus signs whenever a constant is expected. This means that, while in theory constants can include leading minus signs, in practice they are constructed by the parser by applying a unary minus operation at compile-time to an unsigned constant. An alternative to this strategy is to simply build the above solution into the language definition itself: disallow signed constants and extend the grammar to allow leading unary minuses whereever constants can appear.

4.11. We use `-` for subtraction and `/` for division. We add these operations to the BNF and EBNF, leaving the modification of the syntax diagrams of Figure 4.11 to the reader.

BNF:

expr → *expr* **`+`** *term* | *expr* **`-`** *term* | *term*
term → *term* **`*`** *factor* | *term* **`/`** *factor* | *factor*
factor→ **`(`** *expr* **`)`** | *number*
number → *number digit* | *digit*
digit → **`0`** | **`1`** | **`2`** | **`3`** | **`4`** | **`5`** | **`6`** | **`7`** | **`8`** | **`9`**

EBNF:

expr → *term* { (**`+`** | **`-`**) *term* }
term → *factor* { (**`*`** | **`/`**) *factor* }
factor→ **`(`** *expr* **`)`** | *number*
number → *digit* { *digit* }
digit → **`0`** | **`1`** | **`2`** | **`3`** | **`4`** | **`5`** | **`6`** | **`7`** | **`8`** | **`9`**

Note: In the EBNF we have used parentheses to group operations within pairs of brackets in the first two rules. This makes parentheses into new metasymbols. An alternative is to write

expr → *term* { *addop term* }
term → *factor* { *mulop factor* }
addop → **`+`** | **`-`**
mulop → **`*`** | **`/`**

Note that writing the first rule in the following form is incorrect (why?):

expr → *term* { **+** *term* } | *term* { **-** *term* }

4.13. (a)

BNF:

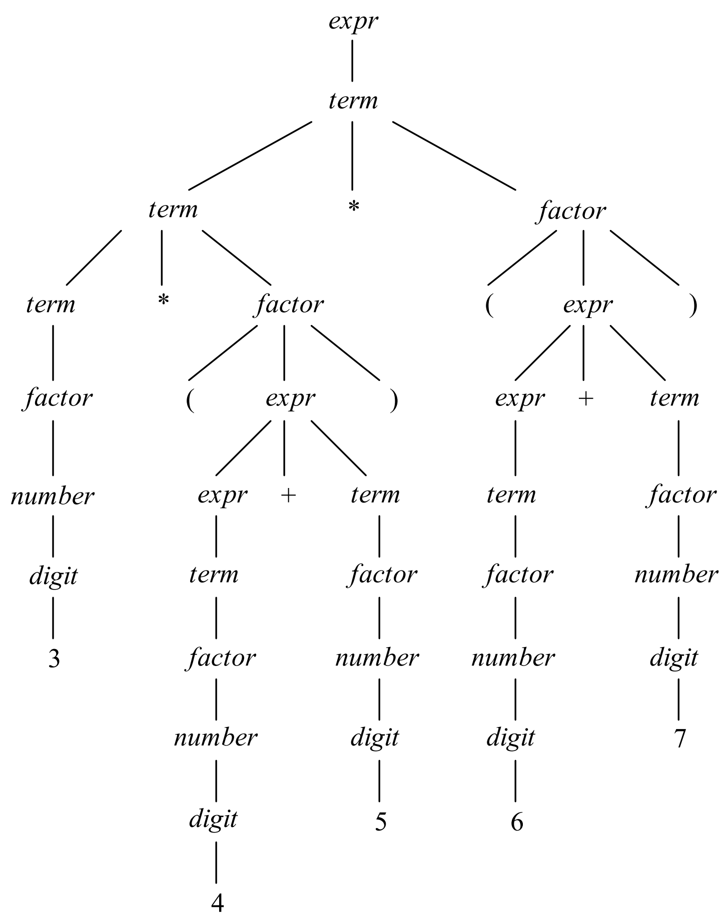
expr → *expr* **+** *term* | **-** *term* | *term*
term → *term* ***** *factor* | *term* **/** *factor* | *factor*
factor→ **(** *expr* **)** | *number*
number → *number digit* | *digit*
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

EBNF:

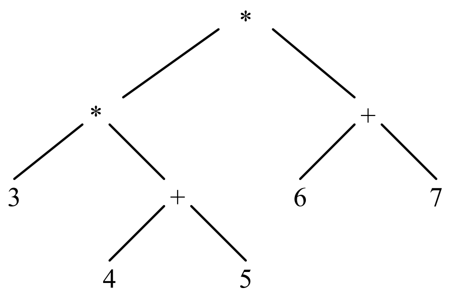
expr → [**-**] *term* { **+** *term* }
term → *factor* { ***** *factor* }
factor→ **(** *expr* **)** | *number*
number → *digit* { *digit* }
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

4.14. (d)

The parse tree is



The abstract syntax tree is



4.19. (a) The changes are in the **`expr`** and **`term`** functions only:

```
int expr(void)
/* expr -> term { ('+'|'-') term } */
{ int result = term();
  while (token == '+' || token == '-')
  { switch (token)
    { case '+': match('+'); result += term(); break;
      case '-': match('-'); result -= term(); break;
    }
  }
  return result;
}

int term(void)
/* term -> factor { ('*'|'/') factor } */
{ int result = factor();
  while (token == '*' || token == '/')
  { switch (token)
    { case '*': match('*'); result *= factor(); break;
      case '/': match('/'); result /= factor(); break;
    }
  }
  return result;
}
```

4.22. Writing the rule as *expr* → *term* + *term* allows only one **+** operation per expression, so that, for example, 3 + 4 + 5 would become illegal. This is not fatal to writing more than one **+** operation in an expression, since parentheses can be used: the expression (3 + 4) + 5 remains legal. But it *does* remove the ambiguity by changing the language recognized rather than by changing the grammar but not the language.

4.24. (a) The changes are to the rules for **`expr`** and **`term`** only:

```
expr    : expr '+' term { $$ = $1 + $3; }
        | expr '-' term { $$ = $1 - $3; }
        | term { $$ = $1; }
        ;

term    : term '*' factor { $$ = $1 * $3; }
        | term '/' factor { $$ = $1 / $3; }
        | factor { $$ = $1; }
        ;
```

4.30. Suppose a declaration has the form **`var _;`**, where **_** stands for any string usable as a variable identifier. Suppose further that only two letters, say, **`a`** and **`b`**, are usable as variable identifiers. Then the possible declarations without redeclaration are {**`var a;`**, **`var b;`**, **`var a; var b;`**, **`var b; var a`**}. These could be generated by EBNF rules

declaration → ε | **`var a;`** [**`var b;`**] | **`var b;`** [**`var a;`**]

Now suppose that **`c`** is also a legal identifier. Then instead of six legal declaration sequences there are fifteen, and EBNF rules look as follows:

declaration → ε | **`var a;`** *no-a* | **`var b;`** *no-b* | **`var c;`** *no-c*
no-a → ε | **`var b;`** [**`var c;`**] | **`var c;`** [**`var b;`**]
no-b → ε | **`var a;`** [**`var c;`**] | **`var c;`** [**`var a;`**]
no-c → ε | **`var a;`** [**`var b;`**] | **`var b;`** [**`var a;`**]

There are now four grammar rules instead of one. The grammar is growing exponentially with the number of variables. Thus, even in a language where variables can be only two characters long, there are nearly a thousand possible variables, and perhaps millions of grammar rules. Writing a parser for such a grammar would be a waste of time.

4.32. We give here the BNF and EBNF rules. Translating the EBNF into syntax diagrams is left to the reader. If a statement sequence must have at least one statement, the grammar rule is easily stated as the following.

BNF: *statement-sequence* → *statement-sequence* **`;`** *statement* | *statement*
EBNF: *statement-sequence* → *statement* {**`;`** *statement* }

However, statement sequences usually are allowed to be empty, which complicates the problem. One answer is to use the previous solution and a helper rule, as follows.

BNF:
statement-sequence → ε | *statement-sequence1*
statement-sequence1 → *statement-sequence1* **`;`** *statement* | *statement*

(Similarly for EBNF.)

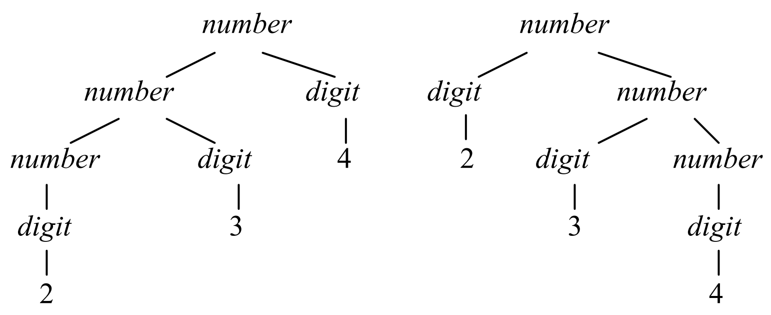
4.36. (a) Here are the number of reserved words in each language:

C:	32
C++:	74
Pascal:	35
Ada:	69
Java:	47

(b) In Pascal predefined identifiers not only include the standard data types, such as **`integer`**, that correspond to reserved words in C, but also include functions such as **`sin`**, **`cos`**, **`abs`**, **`chr`**, **`ord`**, and so on. These correspond to standard library functions in C, C++, Ada, and Java. The problem with Pascal is compounded by the fact that Pascal has *no* separate compilation or standard libraries, while C has a small set of standard libraries, and Ada, C++ and Java (particularly Java) have very large standard libraries. Of course, to be at all useful even C must have libraries comparable to C++ or Java or Ada – it's just that these libraries are all non-standard (such as the Windows C API, for example). Thus just adding library identifiers alone is still misleading. Perhaps one should add all predefined identifiers *and* standard library identifiers to have a fair comparison, but in the modern world of large standard libraries these numbers are very large (in the thousands). If one wishes to get an idea only of the complexity of a language *parser*, rather than the language as a whole, then counting reserved words does do that.

4.38. Indeed, FORTRAN is a language without reserved words, so it is certainly possible for a language to have no reserved words. The problem is that all language constructs become essentially context dependent, and a parser cannot decide which construct is applicable without help from the symbol table and semantic analyzer. This enormously complicates the parsing process, and context-free grammar techniques cannot be used. For this reason, the trend has been toward greater use of reserved words and less use of predefined identifiers in the definition of the basic syntax of a language.

4.40. The difference is immaterial in terms of recognizing numbers. The parse tree becomes significant only if the tree itself is used for further translation or computation. In the case of a number, the main "further translation" that is needed is to compute its value. If we try to compute its value recursively, based on the structure of the tree, we notice a difference in complexity between the two tree structures. Consider, for example, the number 234. It has the two possible parse trees



In the left-hand tree, at a number node with two children, its value can be computed by multiplying the value of its left child by 10 (in base 10) and adding the value of its right child. In the right-hand tree, a more complex computation is required, namely, the number of digits in each value must also be computed. For example, to compute the value of the root of the right-hand tree, the value of its left child (2) must be multiplied by $100 = 10^2$ (the exponent 2 = the number of digits of its right child), and then added to the value of its right child (34). Thus the left-hand tree, which resulted from the left-recursive rule, is to be preferred. This is an example of the principle of syntax-directed semantics.

4.43. (a) There are no precedences or associativities in abstract syntax because the structure of the syntax expression (or syntax tree) itself will provide the order in which the operations to be applied.
(b) There are no parentheses because parentheses are useful only to change associativity and precedence, and by (a) these are non-issues in abstract syntax.
(c) There is no inherent tree structure in numbers: they are just sequences of digits. Thus, a sequential representation as in EBNF is appropriate. This is, of course, not true for *expr*, since there are two subexpressions possible, and this can lead to many distinct tree structures (corresponding to associativity and precedence rules).

4.49. (a) The first condition of predictive parsing is satisfied by the grammar, since the first grammar rule is the only one with an alternative, and

First(**(** *list* **)**) ⋂ First(**a**) = { **(** } ⋂ { **a** } = ϕ

To show that the second condition for predictive parsing is satisfied, we must show that First(*list*) ⋂ Follow(*list*) = ϕ, since *list* is optional in the second grammar rule. We first compute First(*list*). By the second grammar rule, First(*list*) contains First(*expr*). Since this is the only contribution to First(*list*), we have First(*list*) = First(*expr*), and from the first grammar rule we have First(*expr*) = { **(** , **a** }, so First(*list*) = { **(** , **a** }. To compute Follow(*list*), we note that the first grammar rule shows that **)** can follow a list. The second grammar rule gives us no additional information (it tells us only that Follow(*list*) contains Follow(*list*)), so Follow(list) = { **)** }. Then

First(*list*) $\bigcap$ Follow(*list*) = { **(** , **a** } $\bigcap$ { **)** } = ϕ

so the second condition for predictive parsing is satisfied.

(b) C code for **`expr`** and **`list`** recognizers are as follows (using the same conventions as Figure 4.12):

```
void expr(void)
{ if (token = '(')
  { match('(');
    list();
    if (token == ')') match(')');
    else error();
  }
  else if (token == 'a') match('a');
  else error();
}

void list(void)
{ expr();
  if (token == '(' || token == 'a') list();
}
```

Note that in the code for **`list`** we used First(*list*) to decide whether to make the optional recursive call to **`list`**. We could just as well have used the Follow set (which will, however, give a slightly different behavior in the presence of errors):

```
void list(void)
{ expr();
  if (token != ')') list();
}
```

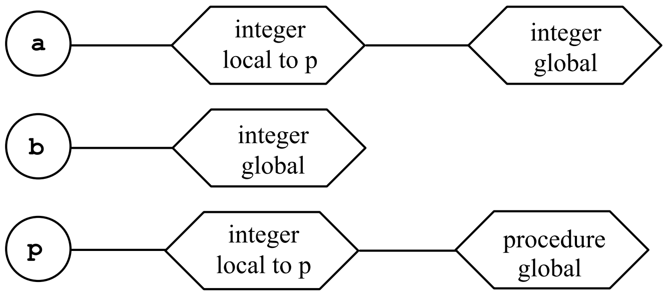
Chapter 5

5.3. (a) In C, a global variable is declared external to any function (including the main program function) and need not be declared at the beginning of the program. It is visible to all functions that follow its declaration. The FORTRAN **`COMMON`** declaration has the advantage of appearing in every procedure that has access to global variables; that is, procedures do not automatically gain access to globals (so **`COMMON`** is a little like a **`with`** declaration in Ada). It has the added flexibility of allowing a name change or alias, but aliasing can be confusing. Additionally, if the position of the variable in the **`COMMON`** block is changed in the main program, it needs to be changed everywhere else as well. Thus the declaration of a global variable is essentially spread over all **`COMMON`** declarations, violating the principle of locality and creating the opportunity for serious errors, since variables are identified by position only. In addition, **`COMMON`** declarations can subvert the type system, since no check for type equivalence is performed on **`COMMON`** variables. Thus a **`CHAR`** variable can be declared **`COMMON`** with a **`REAL`** variable.
(b) The **`EQUIVALENCE`** statement is used to identify variables within the same procedure or function rather than across procedures. Its primary use was to reuse memory allocated to different variables when the uses of the variables did not overlap. This reuse of memory was necessary because of the restricted size of memory in early computers. In modern systems it is much less important, and translators are also able to determine such "overlay" possibilities automatically, which is preferable, because it avoids the problems with aliasing and type subversion that **`EQUIVALENCE`** shares with **`COMMON`**.

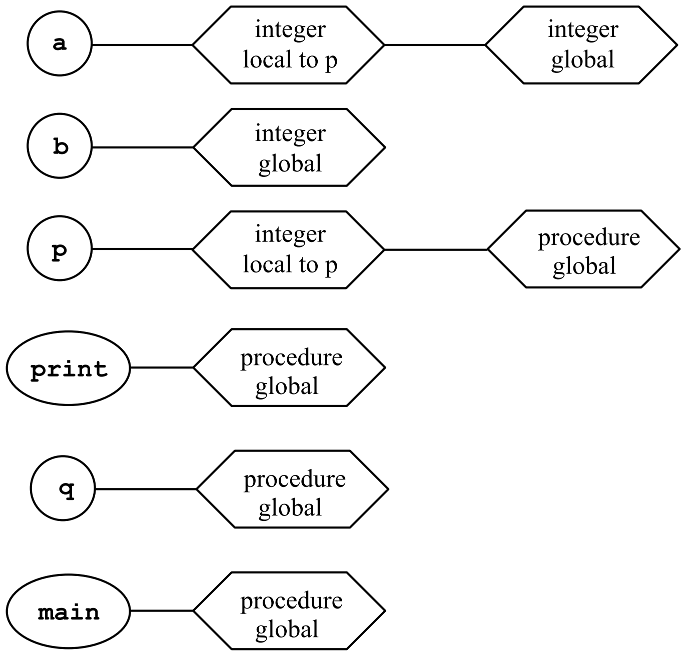
5.6. A C **`extern`** declaration generally binds only data type and scope. If used inside a function it indicates that the declared variable or function is defined elsewhere. If used outside a function, it indicates both that the variable or function has external linkage (i.e. becomes a symbol known to the linker and thus accessible to other files), and that a definition of the variable or function occurs elsewhere (perhaps in the same file). There is one unusual exception to these rules, and that is that, if no definition is found for a group of external declarations for a variable, then as a group they function as a single definition. The use of the **`extern`** declaration for functions is made unnecessary by the fact that it is implicit in a function declaration. However, in order to access a variable from another file, it must be declared **`extern`**, or it would represent a redefinition of the variable and cause a link error.

5.8. We show the symbol tables for Point 1 only:

(a) Using Static Scope:



(b) Using Dynamic Scope: (Note that Point 1 can only be reached from **`main`** through the call to **`p`** in the statement **`a = p()`**.)



Using static scope, the program would print:

```
3
1
```

Using dynamic scope, the program would print:

```
3
```

```
4
```

5.10. The problem is that, if dynamic scoping is used, type checking cannot be performed until execution since a particular name can mean different variables at different times during execution. Thus a static type is of no use, since references cannot be resolved statically. A simple example of this problem is given by the following program (in C syntax):

```
#include <stdio.h>

char x;

void p(void)
{ if (x == "a") printf("ok\n");
  else printf("oops\n");
}

void q(void)
{ char* x = "a";
  p();
}

main()
{ q();
  return 0;
}
```

Statically, there is a type error in procedure **p**, in that the only declaration of **x** known when **p** is processed is the global one of **x** as a character. However, **p** is called only from **q**, so the global declaration of **x** does not extend over **p** during execution. Instead, the local declaration of **x** within **q**, which declares **x** to be a **char*** (a string), extends to **p** during execution, and the code of **p** turns out to be type correct after all. Thus the use of dynamic scope requires that dynamic typing also be used.

There is no similar problem with static scoping and dynamic typing, since type checking can use the static scope structure, and every variable must have its type computed prior to any references to it whether it is statically or dynamically typed. For example, the Scheme language uses static scoping and dynamic typing.

5.12. No. For instance the scope hole problem exists for dynamic scope as it does for static scope; see the answer to Exercise 5.8. In that exercise, global **b** remains allocated through the execution of **q** and **print**, and so has extent equal to the duration of program execution (it is statically allocated), even though the (dynamic) scope of global **b** does not extend to **q** or **print**.

5.15. It is possible that a translator will allocate the same location to **x** and **y,** since they are the first variables declared in successive blocks. If this is the case, then the program will print garbage for **x** (since **x** has not been assigned prior to the first **printf**) and 2 for **y**. It is also possible that garbage is printed for **y** also, which means that the compiler assigns new space to **y** instead of reusing the space of **x**.

5.18. The problem is that 0 can be implicitly cast to any pointer type in C++, and the compiler can't tell whether to cast it to **int*** and use the first definition or cast it to **char*** and use the second. The solution is to provide an explicit cast:

```
void f( int* x) { ... }

void f( char* x) { ... }
```

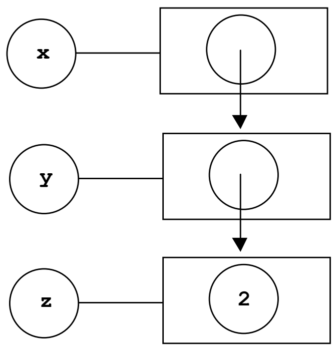
```
int main()
{ ...
  f((int*) 0); // calls the first f
  f((char*) 0); // calls the second f
  ...
}
```

(Note that this problem can also occur in Java and Ada.)

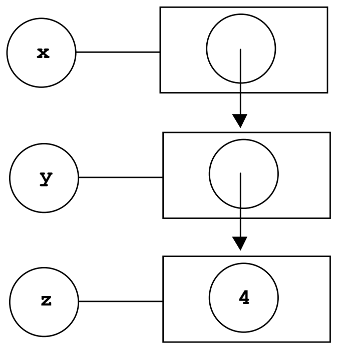
5.21. (a) There are two basic choices: either the insert and lookup operations can continue to use simply the name string as the key, in which case the result of a lookup operation must return a set of 0 or more results for the name in question; or, the lookup operation must be provided with an extra parameter representing the data type of the object being sought. In either case, the symbol table must contain a representation of the data type for each object it contains. Typically this is done by keeping a pointer to each declaration in the symbol table.
(b) Having the symbol table perform overload resolution itself corresponds to the second of the two choices mentioned in part (a). For this to work, the data type of the desired object must be known to the translator in advance; in a very strongly typed language with no implicit type conversions (such as Ada) this is possible. However, in a language such as C++, where type conversions make it impossible for a specific type to be known in advance, the exact data may not be known and this solution cannot be used exactly. Instead, the lookup operation should return a set of possible declarations, and the translator can then determine if there is a data type among those declarations that, after possibly a series of implicit conversions are applied, can be made compatible with the code in which it is to be inserted, based on the type checking rules, and if there is exactly one such that can be distinguished as the preferred choice. Alternatively, the translator could make a "best" guess as to the desired data type, and the symbol table itself could then perform the matching as required by the type rules. In C++ the rules for this process are complex and are best left to a separate overload resolution step that is not part of the symbol table itself.

5.24. (a)
(1) Not an l-value: **+** computes the integer sum of (the rvalue of) **x** and 2.
(2) Not an l-value: **&** returns a pointer (r-value) indicating the address of **x**.
(3) An l-value equivalent to **x** itself.
(4) Since **&x** is a pointer, the **+** computes the pointer (r-value) that points two locations past the location of **x** (scaled by the size of **x**).
(5) An l-value: the dereference unary operator ***** in C, when applied to a pointer, always produces an l-value.
(6) Not an l-value, but the (pointer) r-value contained in **y**. (Thus, ***&x** is equivalent to **x**, but **&*y** is *not* equivalent to **y**.)
(b) No. An l-value *must* always have a corresponding r-value (this value might, of course, be uninitialized garbage).

5.28. After the first assignment to ****x** we have the following picture:



After the second we have essentially the same picture (since there have been no pointer assignments inbetween):



Thus, the same variables are aliases of each other after both the first and second assignments: **`*x`** and **`y`** are aliases, as are **`**x`**, **`*y`**, and **`z`**. The program prints:

```
2
3
4
```

5.31. This is illegal because **`x`** is already allocated on the stack, and cannot also be allocated on the heap.

Chapter 6

6.3. (a) We compare Java and C++. C++ **`bool`** values are implicitly convertible from and to integers, and C pointers are also convertible to **`bool`** values (using the convention that nonzero converts to **`true`** and zero converts to **`false`**). Since in C++ **`bool`** values automatically convert to **`int`**s, all operations on **`int`**s including the comparison operator **`<`** can be applied to **`bool`** values, and indeed **`false`** < **`true`** under this comparison, since **`false`** converts to 0 and **`true`** to 1. In Java this is not true, however: **`int`**s and pointers are convertible to **`boolean`**, but not vice versa. Indeed, there is no conversion in Java of **`boolean`** values to values of any other type. Thus, the comparison operators cannot be applied to **`boolean`** values in Java.
(b) One might try to make a case that order relations are useful for Boolean expressions. For example, the legal C++ (even C) code

```
if ((x <= y) < (x <= z)) x = y;
else x = z;
```

is equivalent to the following:

```
if (x <= z && x > y) x = y;
else x = z;
```

Indeed, **`a < b`** is equivalent to **`b and not a`**, and **`a <= b`** is equivalent to **`b or not a`**. This is probably more confusing than helpful, however, so there doesn't seem to be a real gain in making Booleans into an ordinal type. On the other hand, the convertibility of C++ **`bool`** values into integers is essential to compatibility with C, since C doesn't have a **`bool`** type.

6.6. The reason is that **`enum`** declarations are not true type constructors in C, but merely are shorthand definitions for subranges of the integers (and associated symbolic names for values). In C++, however, **`enum`** is a true type constructor, so different **`enum`** definitions create different types, just as different **`struct`** definitions create different types.

6.9. (a) Suppose *X* is finite, with *n* elements. Then the set *X* × char is also finite and has more elements than *X* does (if char has 128 elements, typical for ASCII character sets, then *X* × CHAR has *n* times 128 elements). But this contradicts the equation *X* × CHAR = *X*. So *X* must be infinite.

(b) Consider an element *x* in the set *X*, and suppose *X* satisfies the equation *X* = *X* × char. Then *x* = (*x*′,*c*) for some *x*′ in *X* and character *c*. Now the same can be said of *x*′: *x*′ = (*x*″,*c*′), where *x*″ is an element of *X*. Continuing in this way, we get an infinite number of characters *c*, *c*′, *c*″, . . . , which must be part of *x*.

(c) Consider the infinite Cartesian product

$$P = \text{char} \times \text{char} \times \text{char} \times ...$$

The set *P* consists of all infinite tuples (*c*1,*c*2,*c*3,...) where each *ci* is in char. This certainly satisfies the equation, since *P* × char consists of the set ((*c*1,*c*2,*c*3,...), *c*0) which is the same as (*c*0,*c*1,*c*2,*c*3,...), which is the same as *P* itself (just add one to all the indices). There is also a sense in which the set *P* is the smallest such set. We omit the details.

6.12.(a) A subtype in Ada is not a new type, but is always considered to be part of its base type. Instead, a subtype is an indication to a compiler to perform range checking on its value during execution. By contrast, a derived type is a completely new type that is incompatible with its base type and other types without explicit type conversion.

(b) The Ada declaration simply establishes the name **`New_Int`** as an alias for **`integer`**, since there is no range specified. A C **`typedef`** does the same thing, so the C declaration

```
typedef int New_Int;
```

is equivalent to the Ada declaration.

(c) The Ada declaration creates a new type **`New_Int`**, which cannot be achieved in C through simple renaming. Thus there is no declaration that is completely equivalent to the Ada declaration. It is possible to imitate it, however, using a type constructor, such as

```
typedef struct { int i; } New_Int;
```

This is not equivalent to the Ada declaration since one must write **`x.i`** to access the value of a variable **`x`** of this type in C.

6.14.A reasonable type correctness rule for the C conditional expression would require that **`e1`** have **`int`** type and that **`e2`** and **`e3`** be type equivalent; the inferred type of the whole expression would then be the (common) type of **`e2`** and **`e3`**. (A similar rule holds, for example, in ML.) However, automatic conversions among pointers and numeric types in C seriously complicates the situation. Here, for example, is the C rule as explained in Kernighan and Ritchie [1988]:

> If the second and third operands are arithmetic, the usual arithmetic conversions are performed to bring them to a common type, and that is the type of the result. If both are **`void`**, or structures or unions of the same type, or pointers to objects of the same type, the result has the common type. If one is a pointer and the other is the constant 0, the 0 is converted to the ponter type, and the result has that type. If one is a pointer to **`void`** and the other is another pointer, the other pointer is converted to a pointer to **`void`**, and that is the type of the result.

6.18.(a) **`x`**, **`y`**, **`z`**, and **`w`** are all equivalent under structural equivalence.

(b) **x** and **y** are possibly equivalent under name equivalence (this is an ambiguity; in Ada they would not be). **z** and **w** are also equivalent (unambiguously). Otherwise, none of the variables are equivalent.

(c) Since C uses structural equivalence for pointers, they are all equivalent under C.

6.23. An array in C is implicitly a pointer that points to the first allocated location; in other words, if a is an array variable, then a is the same as &a[0]. Since arrays are allocated by the system (on the stack), an array is also a constant, and cannot be reassigned. Thus, the equality test in C tests pointer equality and not equality of (all) values. Further, assignment cannot be used, since an array is a constant address.

6.26.Typically there are alignment problems between **int**s and **double**s, since they are usually of different sizes. But even without alignment problems a union cannot be used for such conversions, since the underlying bitwise representations are unrelated, and unions perform no conversion on the bit patterns when switching from one representation to another. For example, consider the following C program:

```
#include <stdio.h>

union
{ long int x;
  float y;
} u;

int main()
{ printf("%d\n%d\n",sizeof(u.x),sizeof(u.y));
  u.x = 1;
  printf("%g\n",u.y);
  return 0;
}
```

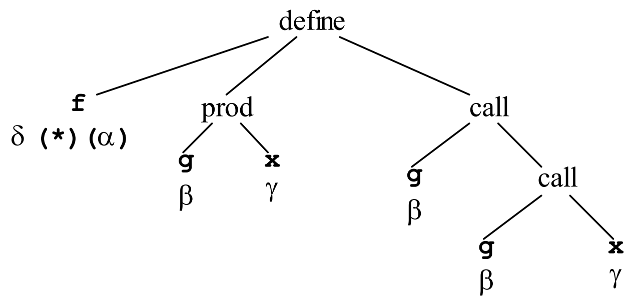
Using either Gnu C or Microsoft C on a PC, this program produces the following output:

```
4
4
1.4013e-045
```

Note that there is no alignment problem for this architecture, since both **long** and **float** are four bytes. However, the floating point result is incorrect.

6.29. (a) Using the notation of ML, this function has type **('a -> 'a) * 'a -> 'a** (a function of the Cartesian product of a function of type **a** to itself with a value of type **a**, producing a value of type **a**), or in the notation of Haskell, **(a -> a, a) -> a**.

(b) Using the notation of Sectin 6.8, here is a possible tree representation for the definition:



By pattern matching, the type checker immediately finds that $\alpha = (\beta,\gamma)$ (using tuple notation for Cartesian product), and that $\delta = (\beta,\gamma) \rightarrow \varepsilon$. Then, from the rightmost call, $\beta = \gamma \rightarrow \eta$, and from the call applied to the result η that $\eta = \gamma$ and, finally that $\varepsilon = \gamma$. Thus, the type of the function is $\delta = (\gamma \rightarrow \gamma, \gamma) \rightarrow \gamma$, as promised (with $\gamma =$ **'a** in the ML notation).

(c) Here is an equivalent C++ template function:

```
template <typename T>
T f (T (*g)(T y), T x)
{ return g(g(x)); }
```

6.34.

```
template <typename First, typename Second>
Pair<First,Second> makePair(First f, Second s)
{ Pair<First,Second> p;
  p.first = f; p.second = s;
  return p;
}
```

6.37. The occur-check problem (see page 243) cannot occur in C++ because C++ uses explicit parametric polymorphism, and self-referential recursive types cannot be written explicitly.

6.39. Polymorphic recursion is resolved more generally in C++ than in ML, and so there is no "problem" with polymorphic recursion in C++. The reason is that C++ does not apply type constraints until templates are instantiated, and a function is instantiated for all types for which the function is used. For example, the following C++ code defines the polymorphic recursive function **f** of Exercise 6.38, instantiates it for types **A** and **bool**, and executes as expected (printing **ok!**):

```
#include <iostream>

using namespace std;

template <typename T>
bool f (T x)
{ if (x == x) return true;
  else return f(false);
}

struct A { int i; };

bool operator==(A x, A y)
{ return x.i == y.i; }

int main()
{ A x;
  if (f(x)) cout << "ok!\n";
  return 0;
}
```

6.42. It is consistent for C to treat array and pointer types similarly in its type equivalence algorithm, since arrays are viewed as being essentially pointers to the first element. Indeed, given the declaration

```
int x[10];
```

the first element of **x** can be accessed as either **x[0]** or ***x**. Similarly, in a parameter declaration **x[]** and ***x** have the same meaning. Now the reason that structural equivalence is used for pointers is related to the fact that all pointers are viewed as essentially equivalent. Moreover, in a recursive declaration such as

```
struct charrec
{ char data;
  charrec * next;
} * x;
```

using declaration equivalence it would be impossible to write

```
x = x -> next;
```

since the declaration of **x** and the declaration of **next** would create two distinct and incompatible types. Thus, list operations (among others) could not be written without **typedef**s:

```
typedef struct charrec * charlist;
typedef struct charrec
{ char data;
  charlist next;
};
charlist x;
```

However, **typedef**s are not actually part of the type equivalence algorithm—they were an addition to the original language, and do not create new types, only synonyms for existing types. By contrast, structures and unions have names created by declarations independent of **typedef**s (for example, **struct charrec** in the foregoing declarations). Thus, declaration equivalence can apply to these declarations, but not to pointers or arrays.

6.46. Typical operations for strings include length, assignment (or copy), comparison (both equality and "<" using lexicographic order), concatenation, substring extraction, the position of a character, the character at a particular position, and replacement of characters within a string. Arrays of characters in a language like Modula-2 (or packed arrays of characters in Pascal) support assignment, comparison, character extraction, and character replacement well (at least within the size constraints provided for in the language). The other operations are not supported directly. Concatenation and substring extraction are particularly problematic because of the restriction that arrays have fixed sizes. In a language with dynamically sized arrays, such as C or Algol60 there are fewer problems with size restrictions, but most of the given operations are not supported (including assignment and comparison in C). C has a standard string function library that removes many of these problems (but not allocation problems). Java has the class String defined as part of the language, which also removes many problems (except for the use of **==** and a few other issues). Ada has a predefined string type that is equivalent to an unconstrained (or open-sized) array of characters. Ada directly supports assignment, comparison, concatenation, and substring extraction (through slicing).

6.49. Equality for floating point types is problematic because it is almost always implemented in hardware as a straight bit test of memory, and floating point values suffer from rounding, truncation, and other errors introduced by floating point arithmetic. For example, the following C++ program may or may not print **ok!** for certain values of **y**:

```
#include <iostream>

using namespace std;

int main()
{ double x,y;
```

```
    cin >> y;
    x = 1/y;
    if (x*y == 1)  cout << "ok!\n";
    return 0;
  }
```

For this reason, one can make a good argument that equality testing should not even be allowed for floating point types, but that programmers should provide their own functions that test approximate equality up to the desired level of precision. ML, for example, forbids the use of equality testing for floating point types (but ML does provide comparison functions in a standard **`Real`** module). Of course, equality testing of floating point values is available in C. Perhaps surprisingly, it is also available in Ada. (To compensate, Ada provides an option to specify the precise number of significant digits of any floating point type; however, programmers must still beware of errors introduced by floating point operations.) Java also permits equality testing on floating point types. To compensate, Java also specifies the precision of all floating point types, and goes farther than Ada in allowing the programmer to specify any method, class, or interface as **`strictfp`**, meaning that all implementations must provide exactly the same results for all floating point operations (and these results are precisely specified). Thus, programmers can be sure that a floating point test will provide the same results across all Java implementations. (See Arnold, Gosling, and Holmes [2000], page 158, for details.) Of course, using **`strictfp`** in Java can have significant performance penalties.

Chapter 7

7.2. (a) *exp* → *exp exp op* | *number*
op → **+** | *****
number → *number digit* | *digit*
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

7.4. The problem is that the arithmetic operations can take more than two operands, so two interpretations of the expression are

```
(+ 3 (* 4 5 6))
```

and

```
(+ 3 (* 4 5) 6)
```

with values 123 and 29, respectively.

7.8. Changing the order of evaluation can improve the execution efficiency of a program, while at the same time changing the semantics of the program, if the program behavior depends on this order. C and C++ are designed for efficient execution; thus, the language opts for maximum efficiency over predicable behavior. Java has the opposite goal: predictable behavior is more important than execution efficiency.

7.10. (a) This doesn't work because in C the parameters **`a`** and **`b`** are evaluated when the **`and`** function is called, so both **`a`** and **`b`** are always evaluated.
(b) Yes. Normal order evaluation delays the evaluation of **`a`** and **`b`** until those values are required in the computation. Thus **`b`** is evaluated only if **`a`** evaluates to a non-zero value (that is, if **`a`** is true)

7.13. The operations **`&`** and **`|`** in C are *bitwise*, not logical; for example, **`2 & 1`** is 0, since 2 is 10 binary and 1 is 01 binary. This is very different from the logical operations (**`2 && 1`** is 1, for example). Since the bitwise operations are generally used to mask bit values, short circuit evaluation doesn't make sense for these operations, since all the subexpressions will represent actual bit values.

7.15. The problem is that one cannot carry out normal-order evaluation by replacement in the presence of recursion without getting into an infinite loop, unless one performs some intermediate simplification first. For example,

```
factorial(3) = if (3 == 0) return 1; else return 3 * factorial (3 - 1);
             = if (3 == 0) return 1;
               else return 3 *
                    (if ((3-1) == 0) return 1;
                     else return (3-1)* factorial ((3-1) - 1));
             = ...
```

What must happen here is that the if-statement must always be evaluated first, before expanding the recursive calls (i.e. applicative-order evaluation must apply to the first argument of an if-statement). Then we get the appropriate normal-order evaluation for the recursive calls:

```
factorial(3) = if (3 == 0) return 1; else return 3 * factorial (3 - 1);
             = if (0) return 1; else return 3 * factorial (3 - 1);
             = 3 * factorial (3 - 1);
             = 3 * (if ((3-1) == 0) return 1;
                    else return (3-1)* factorial ((3-1) - 1)));
             = 3 * (if (2 == 0) return 1;
                    else return (3-1)* factorial ((3-1) - 1)));
             = 3 * (if (0) return 1;
                    else return (3-1)* factorial ((3-1) - 1)));
             = 3 * ((3-1)* factorial ((3-1) - 1)));
             = ...
```

7.19. The issue is whether a new reserved word **`default`** should be added to the language, or whether the existing keyword **`else`** should be reused. Adding a reserved word compromises the simplicity of the language a little, while reusing another reserved word brings up uniformity issues. In this case the use of **`else`** for a "catch-all" case is similar but not identical to its use in an if-statement. Which principle wins is a matter of judgment by the language designer. For example, in C, which also strives for simplicity, a new reserved word was added. Apparently, the designers considered this situation to be enough different to deserve a new reserved word.

7.22. **`while (`*`e`*`)`** ***`s`*** is equivalent to **`for(; `*`e`*`;)`** ***`s`*** in C (that is, the first and third expressions in the for-statement are empty.

7.24.(a) We might try to view **`while`** ***`e1`*** **`do`** ***`e2`*** as an expression by giving it the data type of ***`e2`*** and having it return the last computed value of ***`e2`*** before ***`e1`*** becomes false. The problem is that if ***`e1`*** evaluates to false the first time, ***`e2`*** is never evaluated, and so no value can be returned. We could avoid this problem in one of two ways. The first possibility is for the expression to return an undefined result if ***`e1`*** evaluates to false the first time. The second possibility is to give the expression the data type and last value of ***`e1`*** (that is, Boolean), since ***`e1`*** is always evaluated. The problem with this second possibility is that every while-expression (or at least every one that terminates) will evaluate to false, so its returned value is of little use.

(b) A case-expression is essentially the same as a (nested) if-expression and so the problems mentioned in part (a) do not apply, as long as every case-expression has a required else-part (or default case), or if fall-through would result in a runtime error. If a fall-through is legal, then a case-expression can still be defined, except that fall-through would result in an undefined value for the expression.

(c) The construct **`do`** ***`e1`*** **`while`** ***`e2`*** does not have the same problem as the while-expression, since the body is always evaluated. Thus the data type and value can be that of (the last evaluation of) ***`e1`***.

7.28. (a) The standard example of an if-statement with an ambiguous parse is one with two **`if`**s and one **`else`**:

```
if c1 then if c2 then s1 else s2
```

This statement does have a unique parse using this new grammar, since if we try to match the **`else`** to the outermost **`if`**, we must match **`if`** ***`c2`*** **`then`** ***`s1`*** to a *matched-statement*, which it isn't. However, a slightly more complex example still results in an ambiguity:

```
if c1 then if c2 then s1 else if c3 then s2 else s3
```

It is possible to parse this either as

```
if c1 then (if c2 then s1 else (if c3 then s2)) else s3
```

or

```
if c1 then (if c2 then s1 else (if c3 then s2 else s3))
```

(We have used parentheses to indicate association of the **`else`**s. The translation into parse trees should be immediate.) The first parse is, of course, incorrect.

(b) Here is an unambiguous grammar:

stmt → *matched-stmt* | *unmatched-stmt*
matched-stmt → **`if`** *cond* **`then`** *matched-stmt* **`else`** *matched-stmt*
  | *other-stmt*
unmatched-stmt → **`if`** *cond* **`then`** *matched-stmt* **`else`** *unmatched-stmt*
  | **`if`** *cond* **`then`** *stmt*

7.30 (a) The first **`switch`** statement is legal in C/C++, but the code to increment **`x`** is unreachable (and may produce a warning message). Thus, the value of **`x`** remains 2. (For code to be executed inside a **`switch`** statement, it must be inside braces with a **`case`** label.)
The second **`switch`** statement is also legal, and produces the same result as the first **`switch`** statement, since the code is not associated with any label.
The third **`switch`** statement is also legal in C/C++ and increments the value of **`x`**, so **`x`** becomes 3.

7.34 (a) (with **`FALSE`** defined as 0 and **`TRUE`** defined as 1):

```
int found = FALSE;
for (i = 0; i < n && !found; i++)
{ int zero = TRUE;
  for (j = 0; j < n && zero; j++)
    if (x[i][j] != 0) zero = FALSE;
  if (zero)
  { printf("First all-zero row is: %d\n",i);
    found = TRUE;
  }
}
```

or the slightly more efficient (but also less readable):

```
int found = FALSE;
for (i = 0; i < n && !found; i++)
{ found = TRUE;
  for (j = 0; j < n && found; j++)
    if (x[i][j] != 0) found = FALSE;
  if (found)
    printf("First all-zero row is: %d\n",i);
}
```

7.37 (a) The two loops are not equivalent. The issue is when the increment operation **`i++`** takes place. In the case of the first (**`while`**) loop, the increment always occurs, even if the test fails. In the case of the second (**`for`**) loop, the increment is only performed if the test succeeds. Thus, the value of **`i`** always differs by one at the end of each of these loops.

7.41. After printing

```
()
Number expected in factor
found: )
```

the program will abort, as explained on page 293, since **`factor`** will throw **`Unwind`**, which is no longer in its specification (compare this behavior to the previous behavior as shown on page 299). Note that some compilers get this wrong (try the Gnu compiler if your compiler does not have this behavior).

7.43. One might think that skipping extraneous characters should be done while catching the exceptions generated by these characters. However, that is not a good way to approach the problem, since that spreads the recognition of extraneous characters across the program. One could also try to do the skipping in some central place after unwinding the stack, but then resuming the parse is impossible. Perhaps the best approach is to simply rewrite the **`getToken`** procedure (lines 16-19, page 294) as follows (but see Exercise 7.44):

```
bool valid(int ch)
{ if (isdigit(ch)) return true;
  switch (ch)
  { case '(':
    case ')':
    case '+':
    case '*':
    case '\n': /* not mentioned in exercise, but
                  needed for proper behavior */
             return true;
    default: return false;
  }
}

void getToken() throw (InputError)
{ token = cin.get();
  if (cin.fail()) throw inputError;
  if (!valid(token)) getToken();
}
```

7.47. Typically, code that must be executed regardless of whether an exception has occurred or not (such as closing a file) must be placed at the end of the **`try`** block and at the end of each **`catch`** block, prior to exiting or returning. Since such situations are relatively common, Java has invented a **`finally`** clause that allows such code to be written once (see e.g., Arnold, Gosling, and Holmes [2000], Section 8.4). Of course, if the code is only to be executed if an exception doesn't occur, it must only be written once at the end of the try block – or it can be written after the **`try`** block itself in the surrounding code.

Chapter 8

8.2. The **`inc`** function as written is legal, but wrong. The problem with the code as written is that postfix unary operators have higher precedence than prefix unary operators in C. Thus **`++`** is applied to the pointer **`x`**, not to **`*x`**, with the result that memory is not changed.. Correct code for **`inc`** would be:

```
void inc(int* x)
{ (*x)++;
}
```

8.4(a). The **`intswap`** procedure is in closed form, because there are no references to externally defined entities: assignment in C++ cannot be overloaded on **`int`**s.

8.8. The output of the program using each parameter passing mechanism is as follows:

pass by value:	1 1
pass by reference:	3 1
pass by value-result:	2 1
pass by name:	2 2

8.10. (a) In the case of an expression like **`X+Y`** or **`2`**, a temporary memory location can be allocated during load time. Then code is generated to compute the value of the expression into this location, and the address of the temporary location is passed to the subroutine. Thus **`CALL P(X,X+Y,2)`** has the same effect as

```
TEMP1 = X+Y
TEMP2 = 2
CALL P(X, TEMP1, TEMP2)
```

(b) If, for example, we wished to pass **`X`** by value in the preceding call, we could write **`CALL(X+0,X+Y,2)`**.

(c) Since 1 is a constant in the call **`P(1)`**, there are two possibilities. The first is to use the method just described, with a temporary location being allocated and 1 stored into it. Then on the second call, 1 is not stored again (since it is a constant), with the result that 2 gets printed. The other possibility is that 1 is stored with the program code, and its address is passed to the subroutine. Now an attempt to change a location in the code segment will result in a runtime error, since the code segment is usually read only.

8.13. An example is the following program (in C syntax):

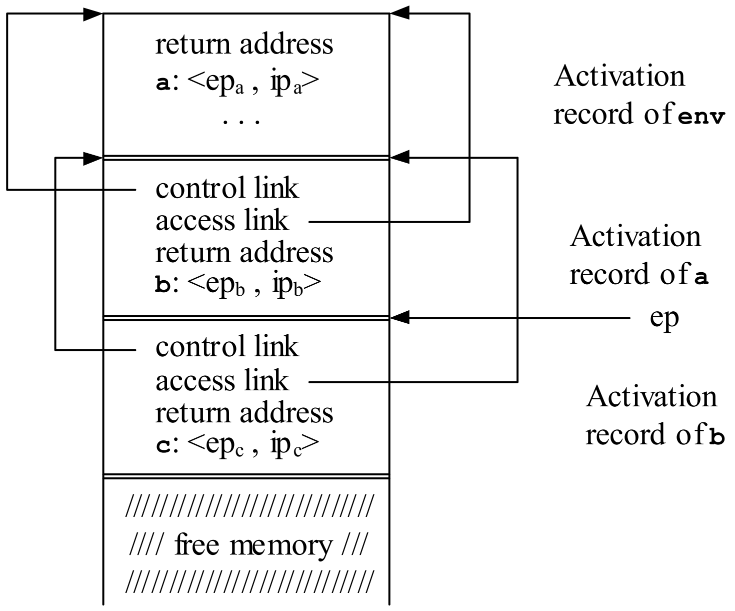
```
#include <stdio.h>

int i;
int p(int y)
{ int j = 1;
  return y;
}

void q(void)
{ int j;
  i = 2;
  j = 2;
  printf("%d\n",p(i+j));
}

main()
{ q();
  return 0;
}
```

This program prints 3 using pass by text (it would print 4 using pass by name). Note that programming languages using pass by text must also use dynamic scoping, since otherwise it is possible to pass an expression outside the scope of one of its component variables (consider the case where **`i`** is also local to **`q`** in the preceding example).

8.15. (a) A complete solution in C syntax that would print the scalar product of two integer vectors is as follows:

```
#include <stdio.h>

int product(int a, int b, int index, int size)
{ int temp = 0;
   for (index = 0; index < size; index++)
      temp += a * b;
   return temp;
}

main()
{ int a[10]; /* initialize in some way */
  int b[10]; /* initialize in some way */
  int i;
  printf("%d\n",product(a[i],b[i],i,10));
  return 0;
}
```

8.18. (a)



8.20. Figure 8.5 on page 336 demonstrates the non-static nature of the access link: **show** has two different access links, resulting from the two different activations of **p** from which it comes. Thus, the access link is dynamically dependent on the location of the active call to the enclosing procedure.

8.22. Variable-length arrays in FORTRAN cause no problems for the static environment, since all parameters in FORTRAN are passed by reference. Thus, an array parameter is, as in C, simply a base address for the appropriate array (indeed, as in C, the size of the array is not part of the representation of the array itself).

8.26. If procedures are never parameters, they must always be called from an environment that is enclosed in the environment where they are defined. That means that when a procedure is called, its location can be found by the usual access chaining, and the environment pointer to the activation where the procedure is found is now the environment pointer of the procedure closure, since the procedure must be defined in the environment in which it is found.

8.32. (a) Below is some code that implements new versions of the C malloc and free library functions, taken from Louden [1997], Figure 7.19, p. 379. This code uses a circular singly-linked list structure, with some extra information in each node that allows for automatic coalescing of adjacent blocks. (In a comparison with a version of the standard C library functions, this code ran slightly faster.)

```
#define NULL 0
#define MEMSIZE 8096 /* change for different sizes */

typedef double Align;
typedef union header
  { struct { union header *next;
             unsigned usedsize;
             unsigned freesize;
           } s;
    Align a;
  } Header;

static Header mem[MEMSIZE];
static Header *memptr = NULL;
```

```
void *malloc(unsigned nbytes)
{ Header *p, *newp;
  unsigned nunits;
  nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
  if (memptr == NULL)
  { memptr->s.next = memptr = mem;
    memptr->s.usedsize = 1;
    memptr->s.freesize = MEMSIZE-1;
  }
  for(p=memptr;
      (p->s.next!=memptr) && (p->s.freesize<nunits);
      p=p->s.next);
  if (p->s.freesize < nunits) return NULL;
  /* no block big enough */
  newp = p+p->s.usedsize;
  newp->s.usedsize = nunits;
  newp->s.freesize = p->s.freesize - nunits;
  newp->s.next = p->s.next;
  p->s.freesize = 0;
  p->s.next = newp;
  memptr = newp;
  return (void *) (newp+1);
}

void free(void *ap)
{ Header *bp, *p, *prev;
  bp = (Header *) ap - 1;
  for (prev=memptr,p=memptr->s.next;
       (p!=bp) && (p!=memptr); prev=p,p=p->s.next);
  if (p!=bp) return;
  /* corrupted list, do nothing */
  prev->s.freesize += p->s.usedsize + p->s.freesize;
  prev->s.next = p->s.next;
  memptr = prev;
}
```

8.34. Since the unused half of available memory is immediately available for allocation, blocks in the used half of memory that are reached during garbage collection can be immediately moved to the unused half of memory on a first-come basis (we must maintain a pointer to the first unused location of the unused half as this process proceeds). The pointer that was used to reach the block currently being moved can then be updated to the new location. The remaining problem is when a previously moved block is again encountered during the further search for accessible blocks. The garbage collector must discover that the block has already been moved, and where its new location is. This can be done by storing a marker and reference pointer right in the memory of the block after it has been copied. This takes up no extra room, since the memory has already been copied. (Of course, the marker must be distinguishable from any actual data that may occur at its location in the block.)

Chapter 9

9.2. (a) realpart($x * y$) = realpart(x) $*$ realpart(y) – imaginarypart(x) $*$ imaginarypart(y)
imaginarypart($x * y$) = realpart(x) $*$ imaginarypart(y) + imaginarypart(x)$*$realpart(y)

(b) realpart(x / y) = (realpart(x) $*$ realpart(y) + imaginarypart(x) $*$ imaginarypart(y)) / L
imaginarypart(x / y) = (imaginarypart(x) $*$ realpart(y) – realpart(x)$*$imaginarypart(y)) / L

where L = realpart(y)^2 + imaginarypart(y)^2.

9.6. (Note that the following solution does nothing about freeing memory.)

complex.h:

```
#ifndef COMPLEX_H
#define COMPLEX_H

struct ComplexRep;
typedef struct ComplexRep * Complex;

Complex makecomplex(double,double);
double realpart(Complex);
double imaginarypart(Complex);
Complex add(Complex,Complex);
Complex subtract(Complex,Complex);
Complex multiply(Complex,Complex);
Complex divide(Complex,Complex);
Complex negate(Complex);

#endif
```

complex.c:

```
#include <stdlib.h>
#include "complex.h"

struct ComplexRep
{ double re;
  double im;
};

Complex makecomplex(double re,double im)
{ Complex temp = malloc(sizeof(struct ComplexRep));
  temp->re = re;
  temp->im = im;
  return temp;
}

double realpart(Complex c)
{ return c->re; }

double imaginarypart(Complex c)
{ return c->im; }

Complex add(Complex x,Complex y)
{ Complex temp = malloc(sizeof(struct ComplexRep));
  temp->re = x->re + y->re;
  temp->im = x->im + y->im;
  return temp;
}

Complex subtract(Complex x,Complex y)
{ Complex temp = malloc(sizeof(struct ComplexRep));
  temp->re = x->re - y->re;
  temp->im = x->im - y->im;
  return temp;
}

Complex multiply(Complex x,Complex y)
```

```
{ Complex temp = malloc(sizeof(struct ComplexRep));
  temp->re = x->re * y->re - x->im * y->im;
  temp->im = x->re * y->im + x->im * y->re;
  return temp;
}

Complex divide(Complex x,Complex y)
{ Complex temp = malloc(sizeof(struct ComplexRep));
  double L = y->re * y->re + y->im * y->im;
  temp->re = (x->re * y->re + x->im * y->im) / L;
  temp->im = (x->im * y->re - x->re * y->im) / L;
  return temp;
}

Complex negate(Complex x)
{ Complex temp = malloc(sizeof(struct ComplexRep));
  temp->re = - x->re;
  temp->im = - x->im;
  return temp;
}
```

9.12. **type** deque(element) **imports** boolean

operations:

create: deque
empty: deque → boolean
front: deque → element
rear: deque → element
addfront: deque × element → deque
addrear: deque × element → deque
deletefront: deque → deque
deleterear: deque → deque

variables: *q*: deque; *x*: element

axioms:

empty(create) = true
empty(addfront(*q*,*x*)) = false
empty(addrear(*q*,*x*)) = false
front(create) = error

rear(create) = error
front(addfront(*q*,*x*)) = *x*
front(addrear(*q*,*x*)) = if empty(*q*) then *x* else front(*q*)
rear(addrear(*q*,*x*)) = *x*
rear(addfront(*q*,*x*)) = if empty(*q*) then *x* else rear(*q*)
deletefront (create) = error
deleterear(create) = error
deletefront(addfront(*q*,*x*)) = *q*
deletefront(addrear(*q*,*x*)) = if empty(*q*) then *q* else addrear(deletefront(*q*),*x*)
deleterear(addrear(*q*,*x*)) = *q*
deleterear(addfront(*q*,*x*)) = if empty(*q*) then *q* else addfront(deleterear(*q*),*x*)

9.13. All the arithmetic operations are constructors, as well as the makecomplex operation. The operations realpart and imaginarypart are inspectors, both of which are selectors. There are no predicates. Thus there are two inspectors and six constructors, and so the total number of axioms should be 12.

9.20. (a) We refer to the six axioms by position 1 through 6 and write the numbers of the axioms used after each equality:

front(enqueue(enqueue(create,*x*),*y*)) = front (enqueue (create, *x*)) [by (2) and (4)]
= *x* [by (1) and (4)]

9.22. One possibility is efficiency, particularly with regard to object code reuse. Although, in principle, it would be nice if all instantiations of a generic package could reuse the same code for the package operations, in practice it is difficult to generate code that can handle data with different sizes and behavior, and such code is liable to execute less efficiently than code that is tailored for a particular data type (the reason for developing static data typing in the first place). Thus it is likely that a compiler will generate new code for each instantiation of a generic type. If the instantiation is delayed to the actual declaration of a variable from the package, then the compiler may be forced to generate a new instantiation for each declaration, instead of being able to share code among all instantiations that use the same data type. Another problem is primarily syntactic, but also presents translation difficulties: a `STACK` is not itself a package in Ada, but is a data type exported by the `STACKS` package. Thus making the `ELEMENT` type a parameter to the `STACK` type makes it unavailable for use by the operations of the package, unless it is also made a parameter to them. Ada's solution is to make `ELEMENT` a parameter to the whole package, which implies that supplying it cannot be delayed to an actual declaration using the exported `STACK` type. Alternatively, the designers could have made packages into types (the way tasks are types; see Chapter 13), but this would introduce the problem of dynamically allocating packages (indeed, tasks, which may be types, are not permitted to have parameters).

9.23. Two possibilities are that dequeue(create) and pop(create) could simply return create instead of error:

dequeue(create) = create
pop(create) = create

Problems arise, however, with the return values of front(create) and top(create). To return an actual value, the imported type element must have a distinguished value (perhaps called nil or undefined) that can become the returned value. This places an added restriction on what data types can be used as elements, something that is difficult to represent in the specification method described in this chapter (Ada does have a way of specifying such restrictions as generic formal subprograms; see the example on page 395). An alternative would be simply to specify these rules with the keyword "undefined" and leave it to the implementor whether this should be an actual error or whether an undefined or distinguished value is returned.

9.28. Using the same notatation as Figure 9.6:

queue.h:
```
#ifndef QUEUE_H
#define QUEUE_H

namespace MyQueue
{ void createq();
  void enqueue(void* elem);
  void* frontq();
  void dequeue();
  bool emptyq();
}
```

```
          #endif
```

<u>queue.cpp:</u>

```
          #include "queue.h"

          namespace MyQueue
          {
            struct Queuerep
            { void* data;
              Queuerep* next;
            }* myq;

            void createq()
            { myq = 0; }

            void enqueue(void* elem)
            { Queuerep* temp = new Queuerep;
              temp->data = elem;
              if (myq)
              { temp->next = myq->next;
                myq->next = temp;
                myq = temp;
              }
              else
              { myq = temp;
                myq->next = temp;
              }
            }

            void* frontq()
            { return myq->next->data;
            }

            void dequeue()
            { Queuerep* temp;
              if (myq == myq->next)
              { temp = myq;
                myq = 0;
              }
              else
              { temp = myq->next;
                myq->next = myq->next->next;
              }
              delete temp;
            }

            bool emptyq()
            { return myq == 0;
            }
          }
```

We leave the comarison with Ada to the reader.

9.33. Since the initial algebra is the image of members of the free algebra, every member of the initial algebra can be represented by a series of constructor operations, that is, enqueue, dequeue, and create. Thus every initial algebra member q has the form enqueue(q',x), dequeue(q'), or create, where q' is another initial algebra member. Now consider the number of operations in the representation of the algebra member q, and pick a representation that has the smallest number of operations. We show that this representation must be in canonical form. Suppose not, so that one of the operations is a dequeue operation:

q = . . . (dequeue(. . .)

Pick the innermost dequeue operation, so that

q = . . . (dequeue(enqueue(. . . (enqueue. . . .)

and let q' be the subpart that begins with the dequeue:

q' = dequeue(enqueue(enqueue(. . .)))

Then by the last axiom of the algebraic specification of the queue ADT, the dequeue can be interchanged with all the enqueue operations except the last:

q' = enqueue(enqueue(. . . (deque(enqueue(create,x)), . . .)))

Now by the same rule dequeue(enqueue(create,x)) = create, and

q' = enqueue(enqueue(. . . (create), . . .))

Thus the representation of q' has had the number of operations reduced by two, and so q itself has a shorter representation, which is a contradiction to the choice of representation. Thus the shortest representation can contain no dequeue operations and so must be in canonical form.

It remains now to show that the initial and final semantics of a queue are the same. Suppose not, so that two different elements q and q' of the initial algebra are equal in the final algebra. Write q and q' in canonical form. These canonical forms must be different. Suppose they differ for the first time at some interior point:

q = enqueue(enqueue(...enqueue($q1$,....)))
q' = enqueue(enqueue(...enqueue($q2$,...)))

with $q1$ = enqueue($q3$,x) and $q2$ = either create or enqueue($q4$,y) with $y \neq x$. In either case we have $q1 \neq q2$ in the final algebra, since front($q1$) = x and front($q2$) = error or y. But then $q \neq q'$ in the final algebra, since each enqueue with the same element must preserve equality (and inequality) in the final algebra. This finishes the proof.

Chapter 10

10.2. In Chapter 5 an object was defined as an area of storage that is allocated in the environment as a result of the processing of a declaration. In Chapter 10 an object was defined to be an instance of a class that can respond to messages. The primary difference in concept is the passive nature of an object according to the first definition and the active nature of the object according to the second. Thus, in object-oriented languages, objects are viewed as controlling their own internal state, while in ordinary imperative languages, objects are acted upon by operations from "outside." In this view even named constants and values can be viewed as objects in an object-oriented language (as indeed true and false are objects of class True and False in Smalltalk), while these are not objects in the sense of Chapter 5, since they are not allocated their own space in the environment.

10.5. An example is the extension of a queue to a deque, given in the chapter. If operations are to be added, access is needed to the internal implementation of a queue, but this makes derived class deque dependent on internal details. If the internal implementation of queue is really hidden from all other classes, then deque must be implemented from scratch.

10.8. Since `x` is an object of class `A`, the fact that `x` might actually be an object of class `B` can only be checked during execution. Thus the validity of the call `x.m` cannot be checked statically. The subtype principle is in fact designed to facilitate static validity checks, and it states the opposite of the situation here, namely, that all the methods of class `A` are callable for derived object `y`.

10.11. (a),(b) Both questions involve increasing the protection of inherited features. As mentioned in the previous exercise, this creates a problem with the subtype principle, so normally this should be prohibited (as it is, in fact, in Java). However, in C++ such protection increase is possible using private inheritance:

```
class A {
private:
  int a;
protected:
  int b;
public:
  int c;
};

class B : private A {
protected:
  A::b;
...
};
```

In class `B` above, `c` has become private, whereas in `A` it was public. Now if in a client `x` is declared to be an `A` and `y` to be a `B`, after the assignment `x = y`, should it be possible to access `x.c`? Indeed, it makes little sense to allow inherited protected features to be made hidden from descendants, since the whole nature of inheritance is to allow for reuse and extension of existing code, and similarly for inherited public features to be made protected or private.

(c) Making a hidden feature protected or public is clearly impossible by the nature of the mechanism. Making a protected feature public is also questionable, and is also unnecessary, at least for methods, since a new public method can be defined that simply calls the protected method.

10.14. The keyword `virtual` is used in C++ in two senses: (1) a method (member function) must be declared `virtual` for it to participate in dynamic binding (pages 432 and 441); (2) inheritance itself must be declared `virtual` in order to achieve shared multiple inheritance (page 444).

10.17. If the order of initializations is not made explicit by the creation routine itself, an order must be inferred from the declarations. In the case of single inheritance, this does not present a problem, since there is one path from the current class to the root of the class hierarchy. Usually, initializations are scheduled in reverse order on this path. In the case of multiple inheritance, there may be multiple paths forming a DAG, and any schedule would have to represent a topological sort of the DAG. One way of scheduling (used by C+ +) is in left-to-right order of the declared base classes, so that given the DAG

$$\begin{tikzcd}
\mathbf{A} & & \mathbf{A} \\
\mathbf{B} \arrow[u, dash] \arrow[dr, dash] & & \mathbf{C} \arrow[u, dash] \arrow[dl, dash] \\
& \mathbf{D} &
\end{tikzcd}$$

the order of initialization would be **A**, **B**, **A**, **C**, **D**, and in the case

$$\begin{tikzcd}
& \mathbf{A} \arrow[dl, dash] \arrow[dr, dash] & \\
\mathbf{B} \arrow[dr, dash] & & \mathbf{C} \arrow[dl, dash] \\
& \mathbf{D} &
\end{tikzcd}$$

the order would be **A**, **B**, **C**, **D**. (Note that in this case **A** should not be initialized twice.) An alternative solution (Eiffel) would be to require that all ancestor initializers be explicitly called.

10.19. The output is

```
B.p
A.q
C.q
B.p
```

We leave the explanation to the reader.

10.24. A class method is a method of class `Class`, and so responds to a message sent to a class viewed as an object of class `Class`, Thus the message `new` is sent to a class name, as in

```
x <- LinkableObject new
```

10.27. [Answer for whileTrue only]
(a) The message `whileTrue` should be sent to a block object (the condition to be evaluated). There are two possible forms, one without parameters and one with: `(B1 whileTrue)` simply evaluates block `B1` until it returns false; `(B1 whileTrue: B2)` evaluates `B1` and goes on to evaluate `B2` if the result of `B1` is true.

(b) These two methods could be defined as follows in the definition of a `Block` class:

```
whileTrue
  self value
    ifTrue: (self whileTrue)
    ifFalse: nil
whileTrue: aBlock
  self value
    ifTrue: [aBlock value.self whileTrue: aBlock]
    ifFalse: nil
```

10.32. The problems with multiple inheritance include choosing a method implementation for invocation among multiple implementations in an inheritance hierarchy, and choosing the order of initializations (i.e. constructor calls) when instantiating an object. Both of these problems do not occur for a Java interface hierarchy, since there are no implementations and no constructors. All Java interfaces do is guarantee the existence of certain methods; they have no other implications for implementation issues.

10.36. (a)

```
class IntWithGcd
{
public:
  IntWithGcd( int val ) { value = val; }
  int intValue() { return value; }
  int gcd ( int v )
  { int z = value;
    int y = v;
    while ( y != 0 )
    { int t = y;
      y = z % y;
      z = t;
    }
    return z;
  }
private :
  int value;
};
```

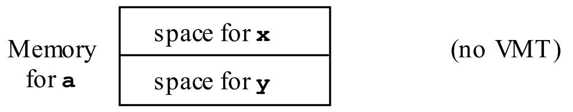
10.38. (a) (in Java)

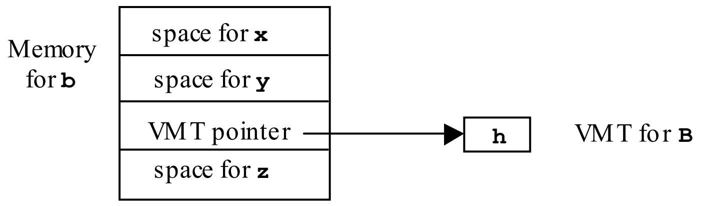
```
class OperatorTree extends SyntaxTree
{ public OperatorTree(char op, SyntaxTree left, SyntaxTree right)
  { if (op == '+' || op == '-' || op == '*' || op == '/')
    { leftSubTree = left;
      rightSubTree = right;
      operator = op;
    }
    else throw new IllegalArgumentException(
      "Bad operator in OperatorTree Constructor:" + op);
  }

  public String toString()
  { return "(" + leftSubTree.toString() + operator
               + rightSubTree.toString() + ")";
  }

  public int value()
  { switch (operator)
    { case '+':
        return leftSubTree.value() + rightSubTree.value();
      case '-':
        return leftSubTree.value() - rightSubTree.value();
      case '*':
        return leftSubTree.value() * rightSubTree.value();
      case '/':
        return leftSubTree.value() / rightSubTree.value();
      default:
        throw new IllegalStateException(
          "Bad operator in OperatorTree: " + operator);
    }
  }

  private SyntaxTree leftSubTree;
```

```
    private SyntaxTree rightSubTree;
    private char operator;
  }

  class NumberTree extends SyntaxTree
  { public NumberTree( int value )
    { val = value;
    }

    public String toString()
    { return (new Integer(val)).toString();
    }

    public int value()
    { return val;
    }

    private int val;
  }
```

Note that **`OperatorTree`** depends in two separate places on the list of allowed operators (and also allows each operator to be only a single character). Much better would be to have an **`Operator`** class that would encapsulate this information, and **`OperatorTree`** would become a client of this class. As a further exercise, try to carry out this program.

10.43. The diagrams in Figure 10.20 would change to the following:





10.47. The problem is that indirect references to objects of the unspecified parameter must be used, since local allocation cannot occur. Thus every use of an object of the parameter class must be replaced by an indirect reference. In Java objects of classes are always references anyway, so this requires no extra work. In C++, however, arbitrary classes are not pointers, so the compiler must provide the indirection.

Chapter 11

11.2.

```
int readMax1(int currentMax)
{ int x;
  scanf("%d",&x);
  if (x == 0)
    return currentMax;
  else if (x > currentMax)
```

```
      return readMax1(x);
    else
      return readMax1(currentMax);
}

int readMax(void)
{ int x;
  scanf("%d",&x);
  if (x != 0) return readMax1(x);
}
```

Note that the **readMax** function returns no value if the (first) **x** is 0; this is because an empty list has no maximum. In practice, this C code will likely return an arbitrary erroneous value; better would be to generate an exception, or at least print an error message and abort, as for example the following:

```
int readMax(void)
{ int x;
  scanf("%d",&x);
  assert(x != 0);
  if (x != 0) return readMax1(x);
}
```

11.6. No. A function that is referentially transparent has a value that depends only on the value of its arguments (parameters). A function that has no side effects makes no change to memory or program state. It is possible for a function to make no changes to memory and still not be referentially transparent. For example, any function whose value depends on the value of a nonlocal variable is not referentially transparent, as in the following C code:

```
#include <stdio.h>

int x;

int p(int y)
{ return x + y;
}

main()
{ x = 1;
  printf("%d\n",p(2));
  x = 3;
  printf("%d\n",p(2));
}
```

If **p** were referentially transparent in the foregoing code, the same value should be printed twice. In fact, the program prints 3 and 5. Similarly, a function can make changes to memory and still be referentially transparent, as in the following code:

```
int x;

int q()
{ x++;
  return 0;
}
...
```

The function **q** changes **x**, but its value is always 0, regardless of the context from which it is called. (A referentially transparent function with no parameters is a constant function; that is, it always returns the same value.)

11.7. (In C)

(a)

```
int B(int n, int k)
{ int t, i;
  if ((n >= 0) && (k >= 0) && (n >= k))
  { t = 1;
    for (i = 1; i <= k; i++)
      t = t * (n - i + 1) / i; /* integer division */
    return t;
  }
  else return 0; /* actually undefined here */
}
```

(b)

```
int B(int n, int k)
{ if ((n >= 0) && (k >= 0) && (n >= k))
    if ((k == 0) || (k == n)) return 1;
    else return B(n-1, k-1) + B(n-1, k);
  else return 0; /* actually undefined here */
}
```

(c) If we use a global array to keep precomputed values of the function, we can reduce the number of calls significantly, since the recursive equation repeats certain calls (**B(10, 5)** saves 16 calls):

```
int memo[20][20];   /* only up to 20 */

int B(int n, int k)
{ if ((n >= 0) && (k >= 0) && (n >= k))
    if ((k == 0) || (k == n))
    { memo[n][k] = 1;
      return 1;
    }
    else if (memo[n][k] > 0)
      return memo[n][k];
    else
    { memo[n][k] = B(n-1, k-1) + B(n-1, k);
      return memo[n][k];
    }
  else return 0; /* actually undefined here */
}
```

11.8. (c)

Scheme:

```
(define (collect)
  (let ((n (read)))
       (if (= n 0) '()
         (cons n (collect)))))
```

ML:

```
open TextIO;
fun collect () =
  let val n = Int.fromString(inputLine(stdIn))
  in
    case n of
    SOME x =>
      if x = 0 then nil
      else x :: collect () |
    _ => raise Fail "Bad Input"
  end;
```

(h)

Scheme:

```
(define (twice f) (lambda (x) (f (f x))))
```

ML:

```
fun twice f n = f (f n);
```

Haskell:

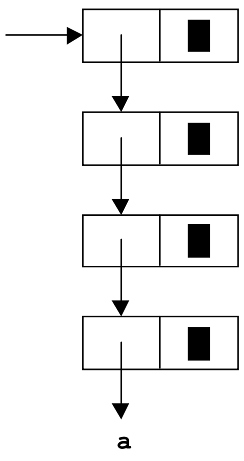
```
twice f n = f (f n)
```

The function `(twice (twice square))` is the function that computes the sixteenth power of its argument.

11.11.

```
instance Show a => Show (BST a) where
  show Nil = "Nil"
  show (Node x Nil Nil) =
    "Node " ++ (show x) ++ " Nil Nil"
  show (Node x Nil right) =
    "Node " ++ (show x) ++ " Nil (" ++ (show right) ++ ")"
  show (Node x left Nil) =
    "Node " ++ (show x) ++ " (" ++ (show left) ++ ") Nil"
  show (Node x left right) =
    "Node " ++ (show x) ++ " ("
            ++ (show left) ++ ") (" ++ (show right) ++ ")"
```

11.13. A box diagram for the list `((((a))))` is as follows:



11.17.

```
(define (deep-reverse L)
  (cond ((null? L) L)
        ((list? L) (append (deep-reverse (cdr L))
                           (list (deep-reverse (car L)))))
        (else L)))
```

11.22. (a) `((lambda x y) E) 2 3)`

(b) No. The **`let`** evaluates all its bindings in the scope surrounding the **`let`**, as the foregoing interpretation of the **`let`** as a lambda shows. To allow **`letrec`** to handle recursive references within its bindings, these bindings must be evaluated within the scope of the **`letrec`** itself; that is, the names established in the binding list are assumed to already have meanings. This is impossible to imitate in a function call unless delayed evaluation is used and the arguments are evaluated using dynamic scope (this was called pass by text in Exercise 8.13). Since Scheme does not use this evaluation rule (on either count), **`letrec`** cannot be imitated by a **`lambda`**.

(c) The value of **`b`** uses the value of **`a`**, which has not yet been bound. (It would be legal if **`a`** had a value in the surrounding scope, but this might not be what the user intended.)

(d) If a **`let`** could be interpreted as equivalent to a cascade of **`let`**s, then this would be legal. For example, if the given **`let`** were equivalent to

```
(let ((a 2))
  (let ((b (+ 1 a)))
   ...))
```

then all would be well. Given the equivalence in part (a), this would be equivalent to saying that **`lambda`** expressions are fully curried. But this is not true (see Exercise 11.26).

11.26. (a) The following attempt will produce an error in Scheme:

```
(define (add x y) (+ x y))
(add 2)
```

(b)

```
(define (curry f)
  (lambda (n) (lambda (m) (f n m))))
```

11.30. (a)

```
allsquares = [n | n <- [0..] , issquare n]
  where
    issquare n = n == square (floor (sqrt (fromInt n) + 0.5))
    square x = x * x
```

In this solution, neither `floor` nor `fromInt` was discussed in the text. The `floor` function in Haskell is the usual truncation function from reals to integers. The `fromInt` function is necessary to resolve overloading (showing that Hindley-Milner type inference is still problematic in the presence of overloading).

11.33. The problem is the ML `fact` function is written using an else-clause, and the else-clause causes an infinite recursive regress. Clearly, it is beyond the scope of a translator to know when a recursive call will never return. On the other hand, given the Scheme definition (using a `cond` expression),

```
(define (fact n)
  (cond
    ((= n 0) 1)
    ((> n 0) (* n (fact (- n 1))))
    ))
```

or the Haskell (using *guards*, which were not discussed in the text),

```
fact n
  | n == 0 = 1
  | n > 0 = n * fact (n - 1)
```

it would be possible for a translator to realize that this is only a partial function, undefined for **n** < 0. (In fact, neither Scheme nor Haskell will in general recognize this, however.) In the case of ML, we can use an exception to indicate the partialness of `fact`:

```
fun fact n = if n < 0 then
                raise Fail "illegal input to fact"
             else if n = 0 then 1
             else n * fact (n - 1);
```

11.37.

```
(define (intlist n) (cons n (delay (intlist (+ 1 n)))))

(define (evens L)
  (let ((a (car (force L))))
    (if (even? a)
      (delay (cons a (evens (cdr (force L)))))
      (evens (cdr (force L))))))
```

These two procedures can be put together as follows (with the delayed version of `take` from page 509):

```
(take 10 (evens (delay (intlist 2))))
```

11.43. A lambda expression for the twice function is $\lambda f .\ \lambda x .\ f\,(f\,x)$.

11.44. The expression (twice (twice square)) has the following form in lambda calculus:

$$(\lambda f .\ \lambda x .\ f\,(f\,x))\ ((\lambda f .\ \lambda x .\ f\,(f\,x))\ (\lambda x .\ *\ x\ x))$$

We reduce this stepwise in the following, using square to represent the square function:

Normal Order:

$(\lambda f\,.\,\lambda\,x\,.\,f\,(f\,x))\;((\lambda f\,.\,\lambda x\,.\,f\,(f\,x))\text{ square}) =$
$\lambda x\,.\,[(\lambda f\,.\,\lambda x\,.\,f\,(f\,x))\text{ square}]([(\lambda f\,.\,\lambda x\,.\,f\,(f\,x))\text{ square}]\;x) =$
$\lambda x\,.\,(\lambda x\,.\text{ square (square }x))([(\lambda f\lambda x\,.\,f\,(f\,x))\text{ square}]\;x) =$
$\lambda x\,.\,(\lambda x\,.\text{ square (square }x))((\lambda x\,.\text{ square (square }x))\;x) =$
$(\lambda x\,.\text{ (square (square }((\lambda x\,.\text{ square (square }x))\;x)))) =$
$(\lambda x\,.\text{ (square (square (square (square }x)))))$

Applicative Order:

$(\lambda f\,.\,\lambda x\,.\,f\,(f\,x))\;((\lambda f\,.\,\lambda x\,.\,f\,(f\,x))\text{ square}) =$
$(\lambda f\,.\,\lambda x\,.\,f\,(f\,x))\;(\lambda x\,.\text{ square (square }x)) =$
$(\lambda x\,.\,(\lambda x\,.\text{ square (square }x))((\lambda x\,.\text{ square (square }x))\;x)) =$
$(\lambda x\,.\,(\lambda x\,.\text{ square (square }x))\text{ (square (square }x))) =$
$(\lambda x\,.\text{ (square (square (square (square }x)))))$

11.48. Since $H = (\lambda F\,.\,\lambda n\,.\text{ (if }(n = 0)\;1\;(*\;n\;(F\;n\;1)))))$ and H fact = fact, we have

fact 1 = (H fact) 1 =
$[(\lambda F\,.\,\lambda n\,.\text{ (if }(n = 0)\;1\;(*\;n\;(F\;(-\;n\;1)))))\text{ fact}]\;1 =$
$[\lambda n\,.\text{ (if }(n = 0)\;1\;(*\;n\text{ (fact }(-\;n\;1))))]\;1 =$
$\text{(if }(1 = 0)\;1\;(*\;1\text{ (fact }(-\;1\;1))))\;(*\;1\text{ (fact }0)) =$
$(*\;1\;((H\text{ fact})\;0)) =$
$(*\;1\;([\lambda F\,.\,\lambda n\,.\text{ (if }(n = 0)\;1\;(*\;n\;(\text{F}\;(-\;n\;1)))))\text{ fact}]\;0)) =$
$(*\;1\;([\lambda n\,.\text{ (if }(n = 0)\;1\;(*\;n\text{ (fact }(-\;n\;1))))]\;0)) =$
$(*\;1\text{ (if }(0 = 0)\;1\;(*\;n\text{ (fact }(-\;0\;1))))) =$
$(*\;1\;1) = 1$

Chapter 12

12.3. We show $a \rightarrow$ false is logically equivalent to not(a) using a truth table. By Exercise 1, $a \rightarrow$ false is false if a is true, and true if a is false, so we have

a	not a	$a \rightarrow$ false
T	F	F
F	T	T

The equivalence follows.

12.7. As we mentioned in the answer to Exercise 2.14, the definition of the gcd is not an algorithm, so we must supply an order for the tests to be performed. We do this by working backward through the integers $1 \le n \le \min(u,v)$. For simplicity, we also use the property that the largest number in that range that divides both u and v is in fact the gcd:

```
gcd(U,V,X) :- min(U,V,M),
              greatestdivisor(U,V,M,X).
greatestdivisor (U,V,M,M) :- divides(U,M),
                             divides(V,M),
                                     !.

greatestdivisor(U,V,M,X) :- N is M - 1,
                            greatestdivisor(U,V,N,X).
min(U,V,U) :- U < V.
min(U,V,V) :- U >= V.
divides(U,M) :- N is U mod M, N = 0.
```

This implementation has linear time in the size of min(*u*,*v*), but it is still much less efficient than Euclid's algorithm (which has logarithmic time complexity).

12.11.

```
last([X],X).
last([X|Y],Z) :- not(Y = []), last(Y,Z).
```

12.14. The following solution uses **append** as defined in Figure 12.1 (page 558):

```
merge([],X,X).
merge(X,[],X).
merge([X|Y],[Z|W],[X|V]) :-
        X < Z,
        merge(Y,[Z|W],V).
merge([X|Y],[Z|W],[Z|V]) :-
        X >= Z,
        merge([X|Y], W,V).
mergesort([],[]) :- !.
mergesort([X],[X]) :- !.
mergesort(X,Y) :-
        split(X,U,V),
        mergesort(U,W),
        mergesort(V,Z),
        merge(W,Z,Y).
split(X,U,V) :-
        size(X,N),
        M is N // 2, /* // = div */
        take(M,X,U),
        append(U,V,X).
size([],0).
size([X|Y] N) :-
        size(Y,M),
        N is M+1.
take(0,X,[]).
take(N,[X|Y],[X|Z]) :-
        N > 0,
        M is N-1,
        take(M,Y,Z).
```

12.17. Since control is expressed by backtracking in Prolog, without the cut it is possible that further alternatives remain to be computed and that the attempt to satisfy the last goal will not be the last computation performed. With the cut, however, the runtime system can throw away any alternatives and maintain a strictly linear (or stacklike) structure in its search to satisfy future goals. The attempt to satisfy the last goal can indeed be viewed as a tail-recursive call, since after it returns no further computation is made.

12.20.

```
fact(N,M) :- fact1(N,M,1).
fact1(0,SoFar,SoFar).
```

```
fact1(N,M,SoFar) :-
         N > 0,
         X is N-1,
         Y is N * SoFar,
         !,
         fact1(X,M,Y).
```

12.23. Consider the search tree as pictured in Figure 12.2, page 559. The cut applies to rule 2, which is only used to produce the rightmost branch from the root. Thus the cut is encountered only at the end of the search and therefore eliminates none of the search paths. Indeed, since the cut is in the last defining clause for `ancestor`, there will never be a case in which this cut will make the search for the first solution more efficient (although it will prevent a further search for more solutions). To improve the efficiency, the second rule should be written first rather than trying to use a cut.

12.26. The difference is caused by Prolog's treatment of `not (X = Y)`. If `X` and `Y` are not both already instantiated when this goal is reached, the goal will succeed by unifying `X` and `Y`, and so the goal `not (X = Y)` will fail, thus causing the `sibling1` goal to fail. This means that `sibling1` is only useful for testing the sibling relationship for two already instantiated variables. On the other hand, `sibling2` may also be used to find sibling pairs, since `X` and `Y` can be instantiated by the `parent` subgoals before reaching the `not (X = Y)`. Thus `sibling2` has more general application and therefore is better.

12.30. The goal `not(human(X))` is interpreted as the failure of the goal `human(X)`. But since `X` is not instantiated, `human(X)` will succeed if there are any humans at all, and so `not(human(X))` will fail.

12.33. In the lambda calculus, variables are bound by being attached to a lambda symbol, and in logic, variables are bound by being attached to existential ("there exists") and universal ("for all") quantifiers. In a purely syntactic sense, then, they are equivalent notions, although lambda calculus is "simpler" in that it has only one symbol that binds variables, while "standard" logic (i.e., first-order predicate calculus) has two. In a slightly more semantic sense, the universal quantifier of logic is more suggestive of the role of the lambda in lambda calculus, in that, given a universal quantifier, a particular substitution for the bound variable can occur. For example, given the statement "for all x in S $p(x)$ is true," one can substitute a particular s from S for x to get the true statement $p(s)$, which is suggestive of beta-reduction (i.e., function application) in lambda calculus. The existential quantifier is less similar in behavior to any operation in the lambda calculus.

12.36. Picking up the state of the program at the end of page 569, the subgoal `fact(N1,R1)` is matched to the second clause, producing the further subgoal `fact(N2,R2)` and the additional constraints

```
R1 = N1 * R2
N1 > 0
N2 = N1 - 1
```

Now the subgoal `fact(N2,R2)` is matched against the first clause, instantiating `N2` to 0 and `R2` to 1, and producing the constraints

```
R1 = N1 * 1
N1 > 0
0 = N1 - 1
```

This gives `N1` the value 1 and `R1` the value 1. Then the previous constrains from page 569 give `X` the value 2, and result in the constraint `6 = 2 * 1`, which fails. Thus, `fact(N2,R2)` is matched with the second clause, producing a third subgoal `fact(N3,R3)` and a third set of constraints

```
R2 = N2 * R3
```

```
N2 > 0
N3 = N2 - 1
```

Now the goal `fact(N3,R3)` is matched against `fact(0,1)`, producing in turn the values **N3** = 0, **R3** = 1, **N2** = 1, **R2** = 1, **N1** = 2, **R1** =2, **X** = 3, and the equation **6 = 3 * 2**, which is true. Thus, the system reports all constraints satisfied with the value **X = 3**.

12.40. The point being made is that Prolog provides the control through the implementation algorithms, particularly the versions of resolution and unification algorithms that are implemented by the system. The programmer has no way of affecting the way the system responds to particular programs, so that the only way to change behavior is to change the logic of the program. However, Prolog does in fact provide some limited control mechanisms in two ways. First, the cut allows the user to change the control the system has over backtracking. Second, the deterministic nature of the search of the database in Prolog—usually sequential—allows the programmer to affect the control by rearranging the order of the program clauses. Neither of these is really a change in the logic of the program.

Chapter 13

13.1. The keywords fi and od are necessary tokens for the parser to determine what statements are contained within the statement block and to disambiguate the dangling else problem. Thus they represent "concrete" syntax used to construct the syntax tree. If we are interested only in abstract syntax, that is, we assume that the syntax tree has already been constructed, then these tokens are unnecessary, and the semantics of a language can be specified without regard to the precise concrete syntax used to express the structure of the syntax tree.

13.4. This question is about the difference between strings as they are constructed by the syntax rules for identifiers and the identifiers themselves. This is similar to the distinction between '0' and 0, and we will write '*a*' for the character and *a* for the identifier. We also write *id* + *a* for the identifier *id* with the letter *a* appended. The reduction rules for identifiers are as follows:

'*a*' => *a*
'*b*' => *b*
. . .
'*z*' => *z*

id '*a*' => *id* + *a*
id '*b*' => *id* + *b*
. . .
id '*z*' => *id* + *z*

Now our use of *I* in the operational semantics should be changed to *id* to refer to identifiers constructed by the given reduction rules rather than strings representing the syntactic representation of identifiers. (Perhaps even better would be to use *IV*, for identifier value, instead of *id*, just as reduced numbers were indicated by *V*.) There is one more inference rule needed that allows for the reduction of a string to an identifier on the left of an assignment:

$$\frac{<I := E \mid Env>,\ I => id}{<id := E \mid Env>}$$

13.8. Even if we make numbers and identifiers into tokens, if a scanner just produces the token and does not save the actual string representing an identifier (or the value of a number), then the necessary information to determine the meaning of the identifier or number will be lost. Thus the scanner cannot escape making these semantic constructions unless we decide to make each character of an identifier or number into a token, and this would be hopelessly inefficient. Indeed, the actual semantic content of the construction of the name of an identifier, or the value of a number, is relatively trivial, and, once we have understood the distinction between syntax and semantics, there is a practical advantage to ignoring these scanner constructions in the formal semantics as well (the complexity of the specification is reduced). See Exercises 13.4 and 13.5.

13.12. We sketch the reductions, listing the number(s) of the rules used in each reduction on the right:

a:=2; b:=a+l; a:=b∗b
=> <a:=2; b:=a+l; a:=b∗b | Env_0> (19)
=> <b:=a+l; a:=b∗b | {a=2}> (16, 18)
=> <a:=b∗b | {a=2,b=3}> (15, 7, 3, 17, 16, 18)
=> {a = 9,b = 3} (15, 9, 5, 17, 16)

13.15. Label the statements of the program as follows:

S1: **`a := 0-11;`**
S2: **`if a then a := a else a := 0 – a fi`**

Then we have

P[[P]](a) = ***L***[[L]](Env_0)(a) = ***L***[[Sl;$S2$]](Env_0)(a) =
(***L***[[$S2$]]°***L***[[$S1$]])(Env_0)(a) = (***L***[[$S2$]](***L***[[$S1$]](Env_0))(a) =
S[[$S2$]](***S***[[$S1$]](Env_0))(a) = (***S***[[$S2$]](***S***[[a:=0–11]](Env_0))(a) =
(***S***[[$S2$]](Env_0&{a = ***E***[[0–11]](Env_0)})) (a) =
(***S***[[$S2$]](Env_0&{a = (***E***[[0]](Env_0)–***E***[[11]](Env_0))}))(a) =
(***S***[[$S2$]](Env_0&{a = (***N***[[0]]–***N***[[11]])}))(a) =
(***S***[[$S2$]](Env_0&{a = (***D***[[0]]–(10∗***N***[[1]]+***N***[[1]])}))(a) =
(***S***[[$S2$]](Env_0&{a = (0–(10∗***D***[[l]]+***D***[[1]])}))(a) =
(***S***[[$S2$]](Env_0&{a = (0–(10∗1+1))}))(a) =
(***S***[[$S2$]](Env_0&{a = (0–11)}))(a) =
(***S***[[$S2$]](Env_0&{a = –11}))(a) =
(***S***[[$S2$]]({a= –11}))(a) =
(***S***[[if a then a:= a else a:= 0–a fi]]({a= –11}))(a) =
(***L***[[a:=0–a]]({a = –11}))(a) =
(***S***[[a:=0 –a]]({a = –11}))(a) =
({a= –11}&{a=***E***[[0–a]]({a= –11})})(a) =
({a= –11}&{a=(***E***[[0]]({a= –11}) –***E***[[a]]({a= –11})})(a) =
({a= –11}&{a=0 –(–11)})(a) = ({a= –11}&{a= 11})(a) =
({a= 11})(a) = 11

13.18. (a) < **`skip`**, *Env*> => *Env*. The importance of skip is that it permits us to rewrite an if-statement with a missing else-part as an if-statement with an else-part, thus reducing the complexity of the semantics of if-statements. It also allows us to simplify and regularize the semantics of the while-statement (see Exercise 13.39 and the answer to Exercise 13.38, below).

13.21. *wp*(**`a:=2;b:=a+1;a:=b*b`**, **`a`** = 9) =
wp(**`a:=2;b:=a+1`**, **`b*b`**=9) =
wp(**`a:=2;b:=a+1`**, **`b`**=3 or **`b`**=–3) =
wp(**`a:=2`**, **`a+1`**=3 or **`a+1`**=–3) =

wp(`a:=2`, `a`=2 or `a`=–4) = (2=2) or (2=–4) = true.

13.24. (a) and (b) are not invariant; (c) is invariant.

13.27. sorted(`a`) = for all `i`, 1 ≤ `i` ≤ `n–1`, `a[i]` ≤ `a[i+1]`
permutation(`a` , `A`) =
there exists a function *f*:{1,...,`n`} → {1,...,`n`}
such that for all `i`, `j`, 1 ≤ `i` < `j` ≤ `n`, *f*(`i`) ≠ *f*(`j`)
and such that for all `i`, 1 ≤ `i` ≤ `n`, `a[i]` = `A[`*f*(`i`)`]`.

13.30. By the distributivity of conjunction and the law of the excluded miracle, *wp*(*S*,*Q*) and *wp*(*S*,not *Q*) = *wp*(*S*,*Q* and not *Q*) = *wp*(*S*,*F*) = *F*. Negating, we get

not(*wp*(*S*,*Q*) and *wp*(*S*,not *Q*)) = not *wp*(*S*,*Q*) or not *wp*(*S*,not *Q*) = T

By the equivalence of *P* → *Q* with *Q* or not *P*), this says that *wp*(*S*,not *Q*) → not *wp*(*S*,*Q*).

13.32. We could define 0 ∗ anything = 0, meaning that if the first operand is 0, the second may be undefined, and the result is still 0. Similarly, we could also define anything ∗ 0 = 0. (Notice that for both these definitions to hold, the computation must be nondeterministic.)

13.34. In a typical semantic definition, identifiers have undefined values until they are assigned a value. If we wanted to eliminate `undef` as a value, we could try to initialize all variables to 0. This will work as long as the result of all operations in the language are defined for all values. In the sample small language of this chapter, there is no division, so this approach would work in this case. In a more realistic language, however, division by zero will result in an undefined value (or possibly a "halt"), and the definition of the environment must allow for this to happen.

13.38. (a) Here are two rules for the if-statement:

$$\frac{<E \mid Env> \;=>\; <V \mid Env>,\; V > 0}{<\text{'if'}\; E\; \text{'then'}\; L_1\; \text{'else'}\; L_2\; \text{'fi'} \mid Env> \;=>\; <L_1 \mid Env>}$$

$$\frac{<E \mid Env> \;=>\; <V \mid Env>,\; V \le 0}{<\text{'if'}\; E\; \text{'then'}\; L_1\; \text{'else'}\; L_2\; \text{'fi'} \mid Env> \;=>\; <L_2 \mid Env>}$$

It should be noted that these rules have a different flavor than the operational rules of the chapter, since the premise involves a complete reduction of an expression to a value, which may in general take many reduction steps. Such rules are called "big step" or "natural" operational semantics; it would be possible to recast the operational rules of this chapter in a consistent "big-step" form. Such rules are more closely related to denotational rules, since the function evaluation of denotational rules also in general requires many subevaluations.

(b) One might be tempted to try to rewrite the "while" rules as three rules like the original "if" rules, including a general condition reduction similar to rule 20. But this is incorrect, since this would imply that the condition of the "while" is evaluated only once, whereas it must be reevaluated if it evaluates to a *V* > 0. Admittedly, the stated "while" rules are really "big-step" rules; the way to get "small-step" rules for the while-statement is to use a skip statement (see Exercise 13.39).

Chapter 14

14.3. To sum *n* integers using *k* processors, with $k < n$, we could divide the *n* integers into *k* groups and assign each processor to sum a group in parallel. Then assign one processor to sum the sums of the *k* groups. This takes approximately *n* div $k + k$ steps, which is less than *n* if $2 \leq k \leq n$ div 2. A better method is to use the so-called binary fan-in technique, where *n* div 2 processors are assigned the task of adding two adjacent numbers (the *i*th processor adds the 2*i*th and $(2i + 1)$th number) in parallel. Then the results of these additions are added two at a time by *n* div 4 processors (these can be some of the same processors that performed the first set of additions). The cascade continues until there is only a single value, which is the sum of all the integers. This computes the sum in $\log_2(n)$ time using at most *n* processors (even if each level is scheduled on different processors). This is the best speedup that can be achieved on standard kinds of machines, so there is no advantage to using more than *n* processors. See, for example, Baase [1988], Chapter 10 (or the 3rd Edition of this book, Chapter 14 [2000]).

14.6. There is confusion between the use of the term fork-join to describe the creation of satellite processes that execute different code (MPMD) and the use of fork and join calls in Unix, where a forked process continues to execute (a copy of) the same code.

14.11.(a) (Note: the following solution ignores questions of an `InterruptedException` occurring, similar to the simple semaphore code of Figure 14.7 – indeed, the code below uses the code of that figure.)

```
class Printers {
// PrinterType assumed to be Object -
// may be a device name or memory address
  public Printers() {
  // initialize printer buffer
    for(int i=0; i < noPrinters; i++)
      printerBuffer[i] = ...; // some initialization here
  }
  public synchronized Object acquirePrinter()
      throws InterruptedException {
    available.delay();
printerBuffer[nextPrinter]);
    return printerBuffer[nextPrinter++];
  }
  public synchronized void releasePrinter(Object printer) {
    if (inUse(printer)) {
      printerBuffer[--nextPrinter] = printer;
      available.signal();
    }
  }

  private boolean inUse(Object printer) {
    for (int i = nextPrinter;i < noPrinters; i++)
      if (printerBuffer[i] == printer)
        return false;
    return true;
  }

  private final int noPrinters = 3;
  // buffer is used as a stack
  private Object[] printerBuffer = new Object[noPrinters];
  private int nextPrinter = 0;
  private Object tempPrinter; // a printer object
  private final Semaphore available = new Semaphore(noPrinters);

}
```

(b) (Note: because of the limitations of declarations inside a protected object in Ada, it is best to surround the protected object with a package.)

```
package PrintManager is
  protected printers is
  -- PrinterType assumed defined elsewhere - it
  -- may be a device name or memory address
    entry acquirePrinter(printer: out PrinterType);
    entry releasePrinter(printer: in out PrinterType);
  end printers;
end PrintManager;

package body PrintManager is

  noPrinters: constant Integer := 3;
  -- buffer is used as a stack
  printerBuffer: array (1..noPrinters) of PrinterType
       := (...); -- initialization
  nextPrinter: Integer := 1;
  count: Integer := noPrinters;
  tempPrinter: PrinterType;

  protected body printers is

    function inUse(printer: in PrinterType)
           return Boolean is
    begin
      for i in nextPrinter..noPrinters
      loop
        if printerBuffer(i) = printer
        then
          return false;
        end if;
      end loop;
      return true;
    end inUse;

    entry acquirePrinter(printer: out PrinterType)
    when count > 0 is
    begin
      printer := printerBuffer(nextPrinter);
      nextPrinter := nextPrinter + 1;
      count := count - 1;
    end acquirePrinter;

    entry releasePrinter(printer: in out PrinterType)
    when true is
    begin
      tempPrinter := printer;
      printer := ...; -- a null value
      if inUse(tempPrinter) then
        nextPrinter := nextPrinter - 1;
        printerBuffer(nextPrinter) := tempPrinter;
        count := count + 1;
      end if;
    end releasePrinter;

  end printers;
end PrintManager;
```

(c)

```
task printers is
-- PrinterType assumed defined elsewhere - it
-- may be a device name or memory address
  entry acquirePrinter(printer: out PrinterType);
  entry releasePrinter(printer: in out PrinterType);
end printers;

task body printers is
  noPrinters: constant Integer := 3;
  -- buffer is used as a stack
  printerBuffer: array(1.. noPrinters) of PrinterType
       := (...); -- initialization
  nextPrinter: Integer := 1;
  count: Integer := noPrinters;
  tempPrinter: PrinterType;

  function inUse(printer: in PrinterType)
         return Boolean is
  begin
    for i in nextPrinter..noPrinters
    loop
      if printerBuffer(i) = printer
      then
        return false;
      end if;
    end loop;
    return true;
  end inUse;
begin -- printers
  loop
    select
      when count > 0 =>
        accept acquirePrinter(printer: out PrinterType)
        do
          printer := printerBuffer(nextPrinter);
        end;
        nextPrinter := nextPrinter + 1;
        count := count - 1;
      or
        accept releasePrinter(printer: in out PrinterType)
        do
          tempPrinter := printer;
          printer := ...; -- a null value
        end;
        if inUse(tempPrinter) then
          nextPrinter := nextPrinter - 1;
          printerBuffer(nextPrinter) := tempPrinter;
          count := count + 1;
        end if;
      or terminate;
    end select;
  end loop;
end printers;
```

14.16. No. When a process forks in Unix a copy is made of all allocated memory, including dynamically allocated pointers. The pointers refer to memory local to each forked process only. (This is facilitated by the fact that all pointers are relative to the starting address of memory allocated to each process.)

14.23. Not really. Initializing a counting semaphore to a negative value means that a number of processes will need to call Signal before any process can be unblocked from a call to Wait. This does not make sense either for mutual exclusion or for access to resources.

14.32. The `accept` statement is within the scope of the parameters of the `entry`. Since some of the parameters may be `out` parameters, these can be assigned at any time during the `accept` statement, and the caller cannot resume execution until the values of all the `out` parameters are specified.

14.35. A task cannot automatically be terminated before it completes its processing, even if the end of its scope in its parent task is reached. For example, in the matrix multiplication of Figure 14.12, page 660 (or the solution to Exercise 14.42 below), the `ParMult` procedure must wait for the completion of the `Mult` tasks, since the result will not be available until all these tasks finish. On the other hand, it would be possible to write specific synchronizing statements (entry calls) that would force a parent to wait for such completion. The Ada designers probably considered that explicitly writing these synchronizing statements was likely to be forgotten by programmers. The alternative is that child processes that loop indefinitely must be explicitly terminated by a terminate statement in a `select` alternative, or the parent process will suspend indefinitely when it reaches the end of that child's scope. Thus the designers have opted for indefinite suspension as a less serious error than abrupt termination.

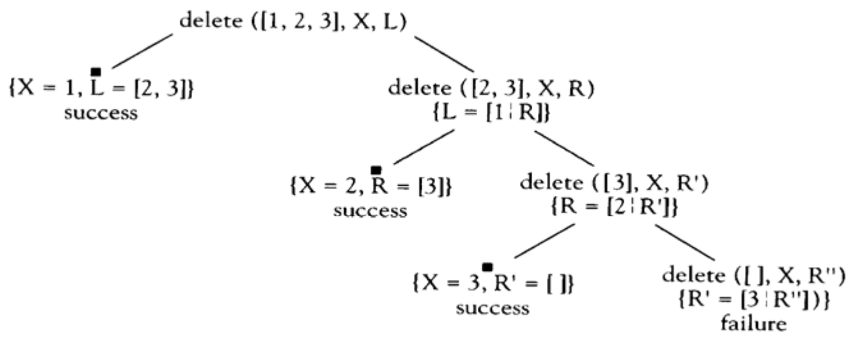
14.37. This Ada code is actually closer to the pseudocode description of the operation of *Signal* and *Delay* in Section 13.4, page 643. It will execute slightly more efficiently than the original code, since the original code will always increment `count` when accepting a `signal`, only to have it immediately decremented again if a process is suspended on `wait`. However, this code is also slightly more complex. Nevertheless, its closer approximation to the definition and its slightly improved speed make it preferable to the original code.

14.42. The following program uses dynamic scheduling of row computations. An alternative strategy is the fixed partitioning of row computations of Figure 14.4.

```
generic size: INTEGER;
package IntMatrices is
  type IntMatrix is array (1..size,1..size) OF Integer;
  function parMult(a,b: in IntMatrix; numProcs: in Integer)
        return IntMatrix;
end;

package body IntMatrices is
  function parMult(a,b: in IntMatrix; numProcs: in INTEGER)
        return IntMatrix is
    c: IntMatrix;
    nextRow: INTEGER := 1;

    task semaphore is
      entry signal;
      entry wait;
    end semaphore;

    task body semaphore is
      acquired: Boolean := false;
    begin
      loop
```

```
            select
              when not acquired =>
                accept wait do
                  acquired := true;
                end;
              or
                accept signal;
                acquired := FALSE;
              or
                terminate;
            end select;
          end loop;
        end semaphore;

        task type Mult is
        end Mult;

        task body Mult is
          iloc: Integer;
        begin
          loop
            semaphore.wait;
            iloc := nextRow;
            nextRow := nextRow + 1;
            semaphore.signal;
            exit when iloc > size;
            for j in 1..size loop
              c(iloc,j) := 0;
              for k in 1..size loop
                c(iloc,j) := c(iloc,j) + a(iloc,k) * b(k,j);
              end loop;
            end loop;
          end loop;
        end Mult;

      begin -- procedure ParMult
        declare m: array (1..numProcs) of mult;
        begin
          null;
        end;
        return c;
      end ParMult;
    end IntMatrices;
```

14.47. The search tree of the Prolog program is displayed here. Since alternatives at the same level are executed in parallel, we can get a sense of how much parallel computational payoff there is by comparing the height of the tree to the number of nodes in the tree. In this case there are seven nodes, but the height of the tree is four. Thus we expect a savings of about three steps, or as much as three sevenths of the execution time. (Because of unification and process overhead, this is not likely to be achieved.)


delete ([1, 2, 3], X, L)
{X = 1, L = [2, 3]}
success
delete ([2, 3], X, R)
{L = [1 | R]}
{X = 2, R = [3]}
success
delete ([3], X, R')
{R = [2 | R']}
{X = 3, R' = [ ]}
success
delete ([ ], X, R'')
{R' = [3 | R'']}
failure