Programming Languages - Principles and Practice
2nd Edition
by Kenneth C. Louden
Thomson Brooks/Cole 2002

Supplementary Exercise Answers

© Copyright Kenneth C. Louden 2003

Permission is granted for the use of this material only by instructors in regularly scheduled classes at a college or university requiring this text. All other uses, duplication, or distribution outside the classroom, including posting on the internet, are prohibited and constitute a violation of this copyright.

Chapter 1

1.1

```
int numdigits(int x)
/* x must be >= 0 */
{ if (x < 10) return 1;
  else return 1 + numdigits(x / 10);
}
```

1.2

(b) The following Scheme procedure is exactly equivalent to the C program of Exercise 1.1:

```
(define (numdigits x)
    (do ((n 1 (+ n 1))
         (temp x (quotient temp 10))
        )
        ((< temp 10) n)
    )
)
```

The following version is equivalent to the answer to Exercise 1.1 given above:

```
(define (numdigits x)
  (if (< x 10) 1 (+ 1 (numdigits (quotient x 10)))))
```

1.3

(a) Here is a typical C-style program:

```
int numdigits( int x, int base)
{ int t = x, n = 1;
  while ((t/=base) >= 1) ++n;
  return n;
}
```

Here is a functional version:

```
int numdigits(int x, int base)
/* base must be > 1, x must be >= 0 */
```

```
{ if (x < base) return 1;
  else return 1 + numdigits( x / base, base );
}
```

(f) Here are two possible solutions:

```
(define (numdigits x base)
  (if (< x base) 1
      (+ 1 (numdigits (quotient x base) base))))

(define (numdigits x base)
    (define (iter n temp)
     (if (>= temp base)
         (iter (+ n 1) (quotient temp base))
         n))
    (iter 1 x))
```

1.12

```
class IntWithGcd
{ public IntWithGcd( int val ) { value = val; }
  public int intValue() { return value; }
  public int gcd ( int v )
  { if(v == 0) return value;
    else return (new IntWithGcd(v)).gcd(value % v);
  }
  private int value;
}
```

1.13

```
class IntWithGcd
{
  public IntWithGcd( int val ) { value = val; }

  public int getValue() { return value; }

  public int gcd ( IntWithGcd v )
  { if (v.value == 0) return value;
    else return v.gcd(new IntWithGcd(value % v.value));
  }

  private int value;
}
```

The new **`gcd`** method is somewhat more object-oriented, since the expectation is that the **`gcd`** method should be symmetric with respect to the object on which it is called and the parameter it is called with. However, it is debatable whether this is a substantial improvement; best would probably be to offer both versions (Java allows this by overloading). A further modification would be to make even the value returned by **`gcd`** into an **`IntWithGcd`** object. Note however that Java will not allow overloading on the return type, so one must decide whether to offer a **`gcd`** with an **`IntWithGcd`** return value or just an **`int`** return value, since both cannot be offered.

1.23 (a) The value of a variable in any language is dynamic -- this is the definition of a variable. (Of course, some languages do not have variables, only constants.)

(b) The data type of a variable in Scheme is determined by its current value, not by its declaration (as in C and Java). Thus, the data type is dynamic. For example, in the Scheme code

© Kenneth C. Louden 2003 All rights reserved

```
(define x 2)
...
(set! x "hi!")
```

the variable **x** starts out having integer data type, but after it is set to the string **"hi!"** its data type also changes to string.

(c) The name of a variable is determined by a **define** statement, as in **(define x 2)** above. In Scheme all variables must be declared in a **define** statement before they are used, and their names are fixed by the **define** statement in which they appear. Thus the name of a variable is static. (Note that the actual location of the variable in memory is dynamic, since allocation does not occur until the program executes the define statement.)

1.24 We can show Turing completeness by showing how to imitate an if-statement in a general way using a while-statement. Suppose a program contains the following if-statement (in C-like syntax):

```
if (e) s1 else s2
```

where **e** is an arbitrary expression, and **s1** and **s2** are arbitrary statements. Assume without loss of generality that **e** evaluates to 0 or 1, with 0 meaning "false" and 1 meaning "true." Suppose further that **x** and **y** are (integer) variable names not used elsewhere in the program. Then the following code (in C-like syntax) has the same effect as the if-statement (except for the extra memory used by **x** and **y**) and can be used to replace it:

```
x = e;
y = 1 - x;
while (x)
{ s1;
  x = 0;
}
while (y)
{ s2;
  y = 0;
}
```

Each if-statement can thus be systematically replaced by similar code, leaving a program with no if-statements. This is what we wished to show.

Chapter 2

2.2 Here is one possibility:

Two numbers, 18 and 50.
What is the largest number that divides both?
Divide 18 into 50, leaving remainder 14.
Divide 14 into 18, leaving remainder 4.
Divide 4 into 14, leaving remainder 2.
Divide 2 into 4, leaving remainder 0.
The number is 2.
This is the procedure.

© Kenneth C. Louden 2003 All rights reserved

2.3 The appropriate formulas are P = 2l + 2w (for the perimeter) and A = lw, so if A = 20 and P = A - 2, then we can interpret the described computation as follows:

P/2 = l + w = 20/2 - 2/2 = 10 - 1 = 9
$(l + w)^2 = 9^2 = 81$
A * 4 = 4lw = 20 * 4 = 80
$(l + w)^2$ - 4lw = 81 - 80 = 1
$(l + w)^2 - 4lw = l^2 + 2lw + w^2 - 4lw$
$= l^2 - 2lw + w^2$
$= (l - w)^2$
(Famous Theorem = Binomial Theorem)
$(l - w)^2 = 1$
l - w = 1 (The special requirement: square root is of 1)
l + w - (l - w) = 2w = 9 - 1 = 8
w = 4
l = A / w = 20 / 4 = 5

2.12 The languages common to both lists, with some possible reasons for their continued existence, are as follows:

ALGOL. This language has now disappeared from the newsgroups, and, while it appears on the Google and ODP resource pages, it does not appear on the Yahoo or Cetus pages. Thus, we must conclude that interest in it is not high. Also, this listing does not say whether Algol60 or Algol68 is meant. Checking the links from Google, we can see that Algol60 interest is mainly limited to historical references by professors, while Algo68 still lives on in compilers for Linux and many other platforms. Why? Algol68 was a failure, but a very advanced language for its time, and so it still has features of current interest. That is likely to be the reason for its continued use.

APL. APL is still very much alive, probably because it is still one of the few languages that allow matrix computations to be expressed quickly. Computational mathematicians, physicists, and modelers still use this language extensively, apparently. For a while, this language suffered from the fact that its symbol set required a special terminal, and those terminals disappeared in the early 1980s. However, that seems to have been overcome by redefining the symbol set into something modern font-based computers can handle. This language also has a successor called J, but J (strangely enough) doesn't seem to be as popular as its older predecessor.

BASIC. This language will probably never die: it is so simple, and runs even on calculators, that it is still evidently enormously popular. Also, Microsoft has taken over the name in VisualBasic, which is a version of BASIC that has little relation to the original BASIC. There are also many other flavors of BASIC around that are used in different ways, mostly now to write GUI applications quickly. It is a reasonable question to ask if these modern versions really are examples of the continued existence of the original language.

COBOL. This language is a "legacy" language: many applications written in it are still being maintained and extended, but virtually no new applications are being written in it. Thus, COBOL programmers are likely to advertise themselves as "legacy systems experts." Why is there still so much running COBOL around? Probably because it is being used by banks and financial institutions that hate to change from working bug-free code to unknown, potentially risky code.

FORTRAN. Still the premier scientific computation language (though C is giving it a lot of competition), and used daily by many physicists, engineers, and others needing scientific data manipulation. One can also ask whether the new versions FORTRAN90 and FORTRAN95 are really FORTRAN as it began,

© Kenneth C. Louden 2003 All rights reserved

since they contain most or all of the features of newer languages (like recursion and user-defined data structures). However, old programs still DO compile, so I think a "yes" answer can be justified.

LISP. A surprisingly resilient language (note that Scheme even has its own newsgroup), probably because of the simplicity and uniformity of its design. It is the BASIC of the Artificial Intelligence community (but much better designed than most BASICs).

PL/1. (Listed as PL/I in the first list and PL1 in the second list, but it is the same language.) As noted in the text, this language was ahead of its time and suffered from being too large, not clearly defined (at first), and having unreliable translators. Over the years these faults have largely been corrected, and continuing support from IBM has allowed this language to survive and even flourish among its dedicated users.

SIMULA. This newsgroup has now also disappeared. Checking the other lists, it is clear that the language is primarily of historical interest. I believe that it was close to extinction in the early 1980s, but had a resurgence of interest as object-oriented programming became popular. Now I would guess that it has entirely been replaced by C++, Java, and Smalltalk.

SNOBOL. Like Algol and Simula, this newsgroup has disappeared too. Indeed, this language appears to be close to extinction, possibly due to its purpose – easy string searching and manipulation (which must be done fast) – coupled with the extreme slowness of its outdated interpreters. It lives on in a more modern language call ICON, which has modern implementations, but is also not a high-interest language. Why did SNOBOL survive so long? Probably because its application domain (string handling and searching) is so important for applications and not well supported in more general languages (at least until recently with the rise of Perl, Tcl, and other scripting languages with built-in regular expression support).

Chapter 3

3.2 Here are some additional examples of non-regularity:

Generality

1. When passing multidimensional arrays to a function, the size of the first index can be left unspecified, but all subsequent sizes must be specified:

```
void f(int x[], int y[][10]) /* legal */
void g(int y[][]) /* illegal! */
```

The reason is that C allows pointer arithmetic on arrays, so that **`y++`** should advance to the next memory location of **`y`** in either **`f`** or **`g`** above (that is, ++ doesn't just add one!). In **`f`**, the compiler knows to interpret **`y++`** as **`y+10*sizeof(int)`**, but in **`g`** it would have no idea how much to increment **`y`**. Thus, incomplete array types are incompatible with pointer arithmetic. (So is this *really* a lack of generality??)

2. The equality test **`==`** is not general enough. Indeed, assignment can be applied to everything except arrays, but equality tests can only be applied to scalar types:

```
struct { int i; double r;} x,y;
...
x = y; /* legal */
if(x==y)... /* illegal */
```

3. Arrays lack generality in C because they must be indexed by integers, and the index must always start with 0. (In Ada and Pascal, arrays can be indexed by characters, and can have any lower index bound.)

© Kenneth C. Louden 2003 All rights reserved

4. The C **`switch`** statement restricts the test expressions to integers, and also allows only one constant value per case. Thus, the switch statement is not sufficiently general.

Uniformity

1. The comma operator has non-uniform semantics in C. In an expression, such as

```
y = (x++,x+1);
```

it guarantees left to right evaluation order, while in a call such as

```
y = f(x++,x+1);
```

the first argument is not guaranteed to be evaluated before the second.

2. The semicolon is not used uniformly as a terminator in C: it must be used to terminate variable and type definitions:

```
int x; <-- semicolon required
```

but it cannot be used to terminate function definitions (because compound statements are *not* terminated by a semicolon):

```
int f(void) { ... }; <-- an error here
```

Orthogonality

1. Functions can return structures and unions, but not arrays. (Is this orthogonality or generality?). Actually this is false. While it is true that functions cannot be declared with a fixed-size array return type, using the array-pointer equivalence of C allows arrays to be returned as pointers. Thus, the following function **`f`** is illegal in C, but **`g`** is fine (nevertheless, there is a problem with **`g`**!):

```
typedef int ArType[10];

ArType f(void)
{ int y[10]; return y;}

int * g(void)
{ int y[10]; return y;}
```

3.4 This is a nonorthogonality, since it is an interaction between assignment and the datatypes of the subjects of the assignment. It could possibly be viewed as a nonuniformity, although we are really talking about an interaction within a single construct rather than a comparison of constructs. 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). C allows the assignment of a real to an integer (with a silent truncation or round), but this is highly questionable, since information is lost by this action, and its precise behavior is unclear from the code itself. One could call this a nonuniformity or nonorthogonality in C as well: assignment does not work the same way, with a special interaction (truncation) occurring for specific data types.

3.8

© Kenneth C. Louden 2003 All rights reserved

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.

3.15 The principle of locality implies that sufficient language constructs should be available to allow variables to be declared close to their use, and also to allow the restriction of access to variables in areas of the program where they are not supposed to be used. There are several constructs in C that promote the principle of locality. First, C allows blocks containing declarations (surrounded by curly brackets {...}) to occur anywhere a statement might occur, thus allowing local declarations a great deal of freedom. For example, if a temporary variable is only needed for a few lines of code, then we can write in C:

```
{ int temp = x;
  x = y;
  y = temp;
}
```

and **`temp`** is restricted to the region within the curly brackets. Ada, C++, and Java all permit this as well; additionally C++, Java, and (recently) C also allow declarations to occur *anywhere* in a block (Ada does not). C also allows local variables to be declared **`static`**, which allows static allocation while preserving restriction of scope. The **`static`** attribute can also be applied to external variables, where it restricts access

© Kenneth C. Louden 2003 All rights reserved

to the compilation unit (other separately compiled code files cannot access it). This also promotes the principle of locality. On the other hand, C only allows global function declarations -- no local functions are allowed. Thus, a function used in only one small part of a program must still be declared globally. This compromises the principle of locality. C also lacks a module mechanism to clearly distinguish what should and should not be visible among separately compiled files. C++ and Java offer the class, which allows a much finer control over access. Ada has the **`package`** construct, which allows significant control over access as well (though not as fine as C++ and Java).

3.22 We answer parts (a) and (b) together in the following.

1. **Design, don't hack**. Basically this means: have some design goals and a plan for achieving them. A set of clear goals was in fact a great strength of the C++ design effort; whether Stroustrup ever had an extensive plan for achieving them is less clear, especially since one of the goals was to allow the language to expand based on practical experience. However, the C++ design effort can be said to have substantially met this criterion.
2. **Study other designs**. Clearly, this means: know what other languages have done well and badly, and choose the appropriate mix of features from them. Definitely this criterion was met, as Stroustrup knew fairly early what features of Simula and C he wanted in the core of C++. Later, he also borrowed carefully from ML, CLU, Ada, and even Algol68.
3. **Design top-down**. Quote from FOLDOC (http://foldoc.doc.ic.ac.uk/foldoc/): "The software design technique which aims to describe functionality at a very high level, then partition it repeatedly into more detailed levels one level at a time until the detail is sufficient to allow coding." For the design of a programming language, this means: start with the most general goals and criteria, then collect a general set of features that will meet these goals, finally refine the feature descriptions into an actual language. The C++ design effort met this criterion probably the least of all, since it was mostly designed from the bottom up, adding one feature at a time beginning with C as the basic language.
4. **Know the application area**. Stroustrup clearly based his design effort on his own and others' practical experience, and he knew extremely well the particular simulation applications that he wanted C++ to solve. So the C++ design effort met this criterion very well.
5. **Iterate**. This means: don't try to make all the design decisions at once, but build up carefully from a core, expanding the target goals as subgoals are met. Clearly the C++ design effort met this goal, as Stroustrup made no attempt to add everything at once, but waited to see how one major feature would work out before adding another.

© Kenneth C. Louden 2003 All rights reserved

Chapter 4

4.5

(a) These are numbers with an optional exponential part: they consist of one or more digits, possibly followed by an "e" or an "E" and an optional + or –, followed by one or more digits. Some examples are `42`, `42e2`, `42E-02`, and `0e+111`.

(b) `[a-zA-Z_][a-zA-Z0-9_]*`

4.8 There are 32 legal sentences generated by the grammar. This is because any legal sentence can be generated from the string

article noun verb article noun **.**

and there are two choices for each of the five nonterminals, for a total of $2^5 = 32$ choices.

4.12

(a) *expr* → *expr* **+** *term* | *term*
term → *term* ***** *power* | *term* **%** *power* | *power*
power → *factor* **^** *power* | *factor*
factor → **(** *expr* **)** | *number*
number → *number digit* | *digit*
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

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

4.13

(b) BNF: *factor* → *factor1* | **-** *factor1*
factor1 → **(** *expr* **)** | *number*
(all other rules as before)

EBNF: *factor* → [**-**] *factor1*
factor1 → **(** *expr* **)** | *number*
(all other rules as before)

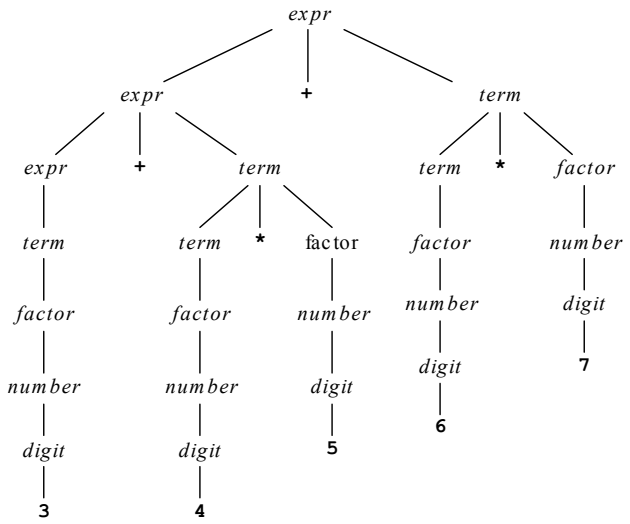
(c) BNF: *factor* → **(** *expr* **)** | *number* | **-** *factor*
(all other rules as before)

EBNF: same as BNF, or:
factor → {**-**} **(** *expr* **)** | {**-**} *number*

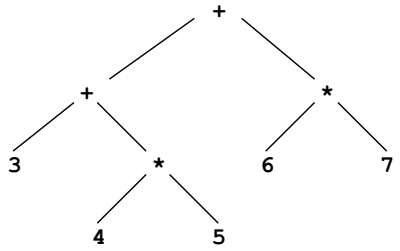
4.14

(b) The parse tree is

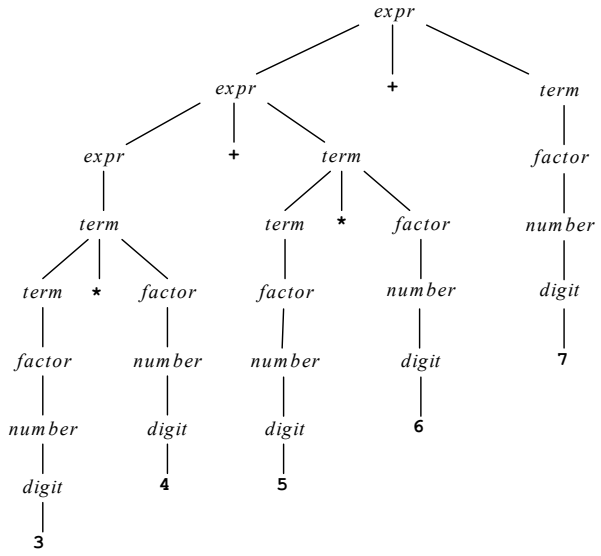
© Kenneth C. Louden 2003 All rights reserved



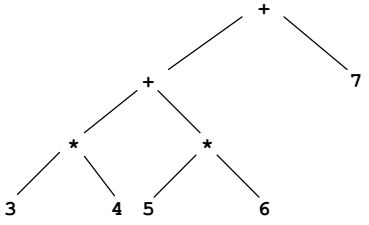
The abstract syntax tree is



(c) The parse tree is

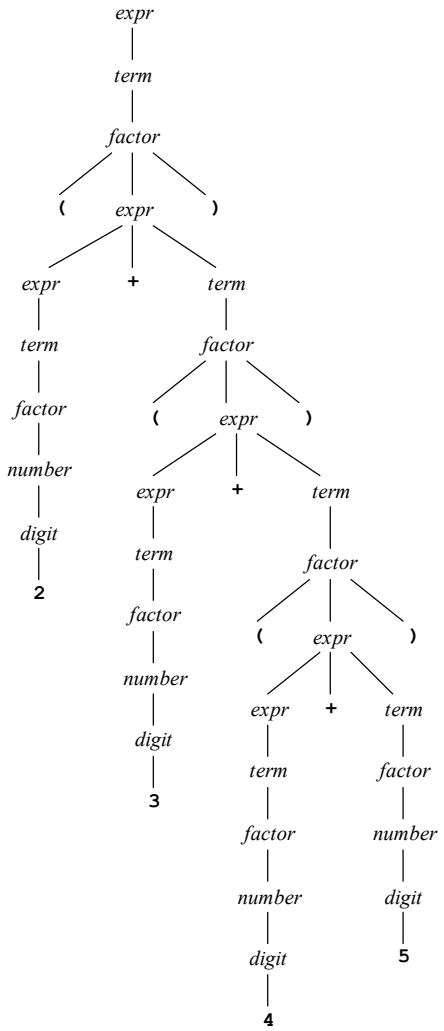


The abstract syntax tree is

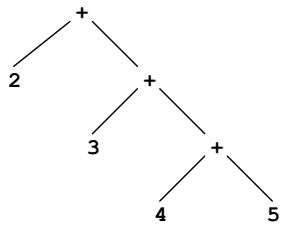


© Kenneth C. Louden 2003 All rights reserved

(e) The parse tree is



The abstract syntax tree is



4.19 (a) The changes needed occur only in **`expr()`** and **`term()`**, which now should appear as follows:

```
int expr(void)
/* expr -> term { '+' term | '-' term } */
{ int result = term();
  while (token == '+' || token == '-')
  { char temp = token;
    match(token);
    if (temp == '+') result += term();
    else result -= term();
  }
  return result;
}
```

© Kenneth C. Louden 2003 All rights reserved

```
int term(void)
/* term -> factor { '*' factor | '/' factor } */
{ int result = factor();
  while (token == '*' || token == '/')
  { char temp = token;
    match(token);
    if (temp == '*') result *= factor();
    else result /= factor();
  }
  return result;
}
```

4.20 (c) A new precedence level and corresponding procedure needs to be added for the power operation, which we call **`power()`** in the following. Changes also need to be made to **`factor()`** to incorporate the calls to power(), as well as to add the new remainder operation. These two procedures now should appear as follows:

```
int term(void)
/* term -> power { '*' power | '/' power | '%' power } */
{ int result = power();
  while (token == '*' || token == '/' || token == '%')
  { char temp = token;
    match(token);
    if (temp == '*') result *= power();
    else if (temp == '/') result /= power();
    else result %= power();
  }
  return result;
}

int power(void)
/* power -> factor [ '^' power ] */
{ int result = factor();
  if (token == '^')
  { int pow, temp;
    match(token);
    pow = power();
    temp = 1;
    while (pow-- > 0) /* works only for positive powers */
      temp *= result;
    result = temp;
  }
  return result;
}
```

4.21 All grammar rule procedures and temporary results, except for **`number()`** and **`digit()`**, need to be changed to **`double`**, and the output in **`command()`** needs its format changed from **`%d`** to **`%g`** or **`%f`**. Also, the following code needs to be added (with changes indicated for the **`factor()`** code):

```
double factor(void)
/* factor -> '(' expr ')' | decimal_number */
{ double result;
  if (token == '(')
  { match('(');
    result = expr();
    match(')');
```

© Kenneth C. Louden 2003 All rights reserved

```
  }
  else
    result = decimal_number();
  return result;
}

double decimal_number()
/* decimal_number -> number [ '.' frac_number ] */
{ double result = number();
  if (token == '.')
  { match(token);
    result += frac_number();
  }
  return result;
}

double frac_number(void)
/* frac_number -> digit [ frac_number ]
   (right recursive because it computes a fractional result
   more easily) */
{ double result = digit() / 10.0;
  if (isdigit(token))
    result += frac_number() / 10.0;
  return result;
}
```

4.29 The following grammar is one possible solution:

sentence → *cap-noun-phrase verb-phrase* **.**
cap-noun-phrase → *cap-article noun*
cap-article → **A** | **The**
noun → **girl** | **dog**
verb-phrase → *verb noun-phrase*
verb → **sees** | **pets**
noun-phrase → *article noun*
article → **a** | **the**

4.31 A scanner can't recognize all expressions, even though an expression is a repetition of terms and plus operators, for the following reason: a term may recursively contain expressions inside parentheses, and a scanner cannot recognize structures that can have the same structure as a substructure.

4.33 We give here the BNF and EBNF rules. If a statement sequence can be empty, we have the following rules:

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

If a statement sequence must contain at least one statement, we must write the rules as follows:

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

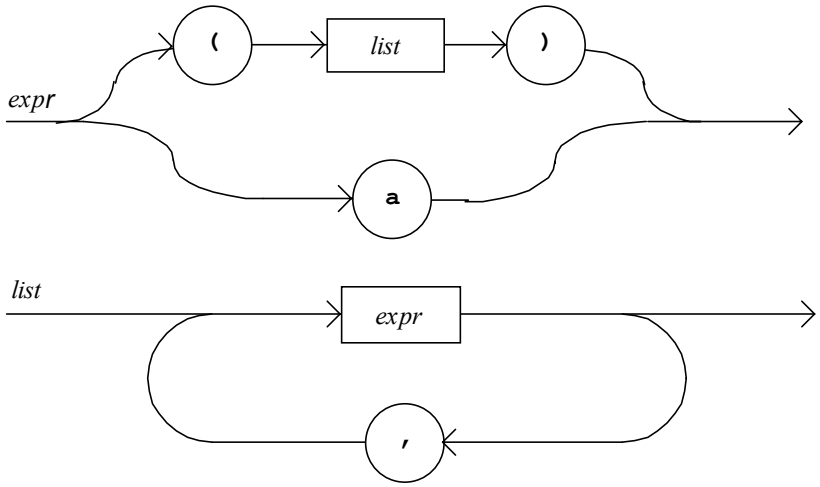
4.42

(a) EBNF rules are

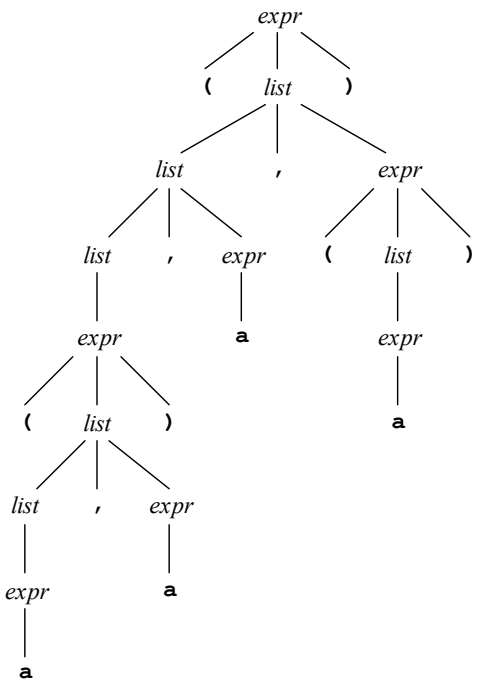
© Kenneth C. Louden 2003 All rights reserved

expr → **(** *list* **)** | **a**
list → *expr* { **,** *expr* }

Syntax diagrams are



(b)



(c) In C-like pseudocode, a recursive-descent recognizer is as follows (using a global variable **`token`**, a global **`match`** procedure, and global procedure **`error`** that prints an error message and exits) :

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

© Kenneth C. Louden 2003 All rights reserved

```
  else error();
}

void list(void)
{ expr();
  while (token == ',')
  { match(',');
    expr();
  }
}
```

Chapter 5

5.1

Binding Times

	Attribute	C	C++	Java
a	Maximum significant digits in a real number	Language implementation time (min of 10 decimal for **`double`**)	Same as C	Language definition time (approx. 18 decimal digits)
b	Meaning of char	Language definition time (but implementation time for converted **`int`** values)	Same as C	Language definition time (including int values)
c	Size of an array var	Translation time	Same as C	Execution time
d	Size of an array param	Execution time	Same as C	Execution time
e	Location of a local var	Execution time	Same as C	Execution time
f	Value of a constant	Translation time (**`#define`**) or execution time (local **`const`** declarations)	Same as C	Translation time (**`final`** declarations)
g	Location of a function	Load time	Load time (functions and non-virtual methods); execution time (methods)	Load time (static methods); execution time (methods)

5.2 The statement "early binding promotes security and efficiency" means that the earlier an attribute is computed, the more it can be used to perform security checks and to improve the efficiency of programs. A good example of this is the data type attribute. If it is computed statically, as in Pascal or C, then it can be used to type check a program (which provides an increased level of security for correct execution) as well as to make efficient allocation of memory. By contrast, Scheme programs are typed only during execution, so type errors cannot be caught in advance. For example, taking (car 3) produces a runtime error, rather than an error prior to execution. Scheme also must allocate all variables dynamically during execution (i.e. as pointers), so allocation is not as efficient as in Pascal or C.

The statement "late binding promotes flexibility and expressiveness" means that the later an attribute is computed, the more it can be computed in a way that allows the program to respond to new situations during execution, thus providing greater flexibility and relieving the programmer from dealing with many special cases. For example, in Scheme we can write a procedure that computes the average of two numbers as

```
(define (ave x y) (/ (+ x y) 2))
```

© Kenneth C. Louden 2003 All rights reserved

This procedure can be applied to both integers and real numbers: `(ave 2 4)` returns 3, and `(ave 2.1 3.4)` returns 2.75. In Pascal or C, we would have to write two separate procedures, one for integers and one for real numbers. This is a direct consequence of the fact that the data type is bound later in Scheme than in C or Pascal.

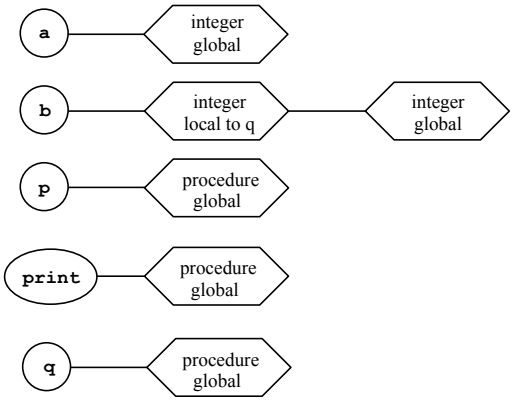
5.7

(a) definition
(b) definition
(c) declaration
(d) declaration
(e) definition
(f) declaration

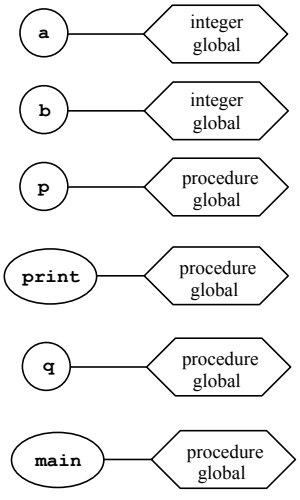
5.8. The symbol tables for Point 1 are given in the posted solutions; we give here the symbol tables at points 2 and 3.

(a) Using static scope:

Point 2:



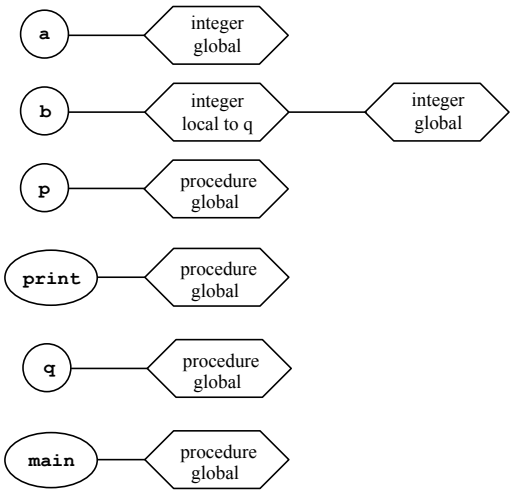
Point 3:



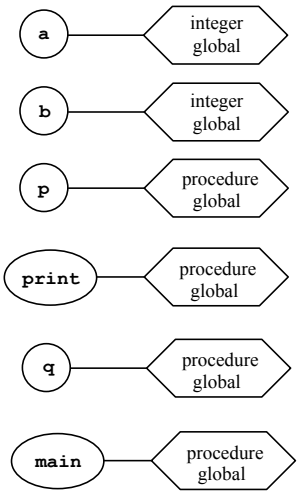
(b) Using dynamic scope:

Point 2:
(Note that this point can only be reached from `main` through the call `q()`.)

© Kenneth C. Louden 2003 All rights reserved



Point 3:
(Note: same as static scope.)



5.9 We do not show the symbol tables here. The program prints 3 1 2 (each number on its own line) using static scope, but 3 4 5 using dynamic scope.

5.14 After each call to **p**, the memory allocated for the execution of **p** is released, and may be reused by subsequent calls to **p**. Since the layout of this memory is the same for each call to **p**, and since an immediate call to **p** may allocate this memory before any reuse of the memory has occurred, the previously stored value of **y** may still be in this memory location, and will therefore be printed during the second call to **p**.

5.17

	(a) C++	(b) Java	(c) Ada
`x = pow(2,3);`	1	1	1
`y = pow(2,3);`	1	1	illegal
`x = pow(2,3.2);`	3	illegal	illegal
`y = pow(2,3.2);`	3	3	3
`x = pow(2.1,3);`	illegal	illegal	illegal
`y = pow(2.1,3);`	illegal	2	illegal
`x = pow(2.1,3.2);`	2	illegal	illegal

© Kenneth C. Louden 2003 All rights reserved

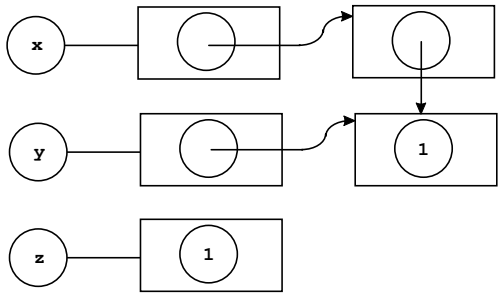
`y = pow(2.1,3.2);`	2	2	2

5.22 The basic problem is that any new declaration within the same scope would likely make it impossible to access the original declaration, by the stack-based behavior of the symbol table, and thus the original declaration may be completely inaccessible everywhere. For instance, if the C block

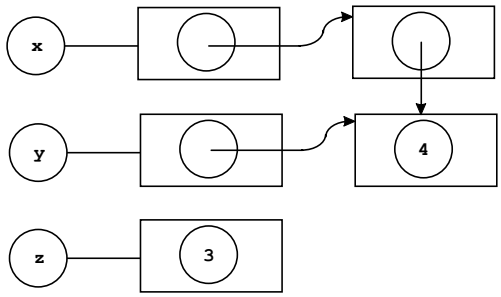
```
{ int x;
  char x;
  ...
}
```

were legal, the first declaration of **x** would be completely inaccessible, and therefore of no practical use. In addition to this problem, the programmer is likely to get very confused by two different meanings for **x** within the same scope. Since there is essentially no practical use for this facility, and it is likely to cause confusion, a design decision is usually made to make it illegal. Even if different data types might make it possible to disambiguate uses, significant ambiguity would result from the possibility of implicit conversions.

5.26 After the first assignment to ****x** (line 11) we have the following picture:



After the second (line 15) we have the following picture:



Thus, the same variables are aliases of each other after both the first and second assignments: ****x** and ***y** are aliases, and there are no other aliases. The program prints the values 1 1 3 (each on its own line).

5.27 The call to **malloc** on line 6 allocates ***x**, and the call on line 7 allocates ***y** (the two memory locations in the rightmost column of the figures in the answer to Exercise 5.26 above). These allocations are necessary prior to the assignments to ***x** and ***y** (on lines 10 and 9, respectively). If line 6 is left out, then the assignment on line 10 changes an arbitrary memory location pointed to by the potentially random value contained in **x**, which could cause either a runtime exception, unusual program behavior, or apparently correct behavior, depending on the value of **x** at the time. A similar situation occurs if line 7 is omitted.

5.33 This version of the code treats **gcd** as the name of a memory location containing the entry address for the code of the **gcd** function; this memory location is then copied into the **fun_var** variable, and then

© Kenneth C. Louden 2003 All rights reserved

dereferenced to find the actual entry point at the call site. This represents an extra level of indirection that is unnecessary, since **gcd** can as in Figure 5.29 be considered to be a constant representing the actual entry point itself. On the other hand, this extra level of indirection agrees more with the actual definition of **fun_var** using the * operator, and is therefore permitted. The real reason for the use of the * operator is that it is a syntactic crutch to distinguish function definitions from function variable definitions. Compilers will generate the same code in either case.

Chapter 6

6.4 (a) The reason is that **x** and **y** have the same type, since they are defined together in the same declaration. The variable **z**, on the other hand, has a different type, since it is defined in a separate **struct** declaration. (This is declaration equivalence in action.)
(b) One way to fix the problem is to add **z** to the definition of **x** and **y**:

```
struct
{ int i;
  double j;
} x, y, z;
```

This is not a very good solution, since the actual data type of **x**, **y**, and **z** is still anonymous. A better way is to give the **struct** type a name, and then to define **x**, **y**, and **z** as desired using this name:

```
struct S
{ int i;
  double j;
};
struct S x, y;
struct S z;
```

Best would be to use a **typedef** in C to define a name that includes the **struct** as well:

```
struct
{ int i;
  double j;
} S;
S x;
S y;
S z;
```

6.5 The assignment won't compile because C treats arrays as pointers which, however, are constant and cannot be reassigned (since the memory pointed to is allocated automatically on the stack). There are several ways the declarations could be modified so that the assignment would work. Perhaps the simplest is as follows:

```
int *x;
int y[10];
...
x = y; /* now it works */
x[2] = 17; /* ok to treat x as array */
...
```

In this version **x** becomes an alias for **y** (at least until **x** is reassigned).

© Kenneth C. Louden 2003 All rights reserved

6.7 C and C++ both use structural equivalence for function types, so the assignment **`f = g`** (or **`g = f`**) is fine, since the types are structurally identical. On the other hand, **`h`** has a different type (**`char`**) for its parameter, and so does not have an equivalent type. Thus, the assignment **`h = g`** generates an "incompatible pointer type" warning in C, and an error in C++ (since C++ takes a stricter view of such conversions). In fact, this assignment may or may not work, depending on the runtime environment: characters can indeed be viewed as integers, but they may have a different size, and this could throw off the management of the runtime stack. Further note: if **`char`** is viewed by a language as a subtype of **`int`**, then this assignment should work, since then the type **`int(*)(int)`** is a subtype of **`int(*)(char)`** [this is an example of a **contravariant** type constructor – see e.g. Kim Bruce, *Foundations of Object-Oriented Languages*, MIT Press, 2002, page 27].

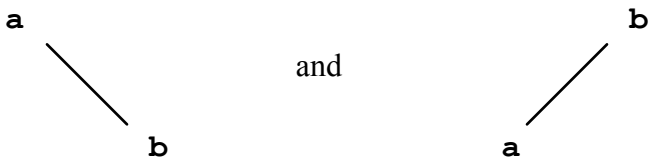
6.10 (a)
```
union CharTree
{ enum { nothing } emptyCharTree; /* no data */
  struct
  { char data;
    union CharTree left; /* still illegal */
    union CharTree right; /* still illegal */
  } charTreeNode;
};
```

(b) **`CharTree`** = **`{empty}`** ∪ **`char`** × **`CharTree`** × **`CharTree`**

(c) The same solution as on page 214 is a possible answer here:

`{empty}` ∪ **`char`** ∪ **`char`** × **`char`** ∪ **`char`** × **`char`** × **`char`** ∪ . . .

This is because a tree can contain no characters (the empty tree), or one character (the tree with one node), or two characters (a tree with two nodes, one either the left or right child of the first), etc. However, this does obscure the structure of the tree, since we are identifying the two trees



We *could* distinguish these trees by considering the left one above to be in the set **`{empty}`** × **`char`** × **`char`**, and the right one to be in the set **`char`** × **`char`** × **`{empty}`** (note that there cannot be any trees in the set **`char`** × **`{empty}`** × **`char`**). Then the appropriate solution to the equation would be

`{empty}` ∪ **`char`** ∪ **`{empty}`** × **`char`** × **`char`** ∪ **`char`** × **`char`** × **`{empty}`**
∪ **`char`** × **`char`** × **`char`** ∪ . . .

(d) We give a sketch of the proof. Let us call the above set by the name *T*:

T = **`{empty}`** ∪ **`char`** ∪ **`char`** × **`char`** ∪ **`char`** × **`char`** × **`char`** ∪ . . .

First we show that, given any solution *S* to the equation (so *S* = **`{empty}`** ∪ **`char`** × *S* × *S*), then *S* must contain *T*. We do this for each of the Cartesian products in *T* by induction on the number *n* of **`char`** components in the product. First, clearly the set **`{empty}`** is in *S*. Then **`char`** is in *S*, since **`char`** ×

© Kenneth C. Louden 2003 All rights reserved

{empty} × **{empty}** = **char** must be in *S*. Now suppose that **char** × **char** × . . . × **char** (*n*−1 times) is in *S*. Then since **{empty}** is in *S* we must have that

$$\underbrace{\mathbf{char} \times (\mathbf{char} \times \mathbf{char} \times \ldots \times \mathbf{char})}_{(n-1 \text{ times})} \times \mathbf{\{empty\}} = \underbrace{\mathbf{char} \times \mathbf{char} \times \mathbf{char} \times \ldots \times \mathbf{char}}_{(n \text{ times})}$$

is in *S*. This shows that *T* is always in *S*. If *T* is also a solution, then it clearly must be the smallest such solution.

So it remains to show that *T* is itself a solution to the equation. By similar resoning to the above, we can easily see that $T \subset \mathbf{\{empty\}} \cup \mathbf{char} \times T \times T$. Conversely, consider the fact that

$$\mathbf{char} \times T \times T = \mathbf{char} \times \left(\{\text{empty}\} \cup \bigcup_{n=1}^{\infty} \mathbf{char}^n \right) \times \left(\{\text{empty}\} \cup \bigcup_{n=1}^{\infty} \mathbf{char}^n \right)$$

$$= \left(\bigcup_{n=1}^{\infty} \mathbf{char}^n \right) \subset T. \qquad \text{(We are handwaving a bit here!)}$$

Thus $\mathbf{\{empty\}} \cup \mathbf{char} \times T \times T \subset T$. Quod Erat Demonstrandum!

(e)
```
struct CharTree
{ char data;
  struct CharTree *left, *right;
};
```

6.11 The given type declarations are all legal in C. Their Ada equivalents are also legal:

```
type Rec1;
type Rec2;
type Ptr1 is access Rec1;
type Ptr2 is access Rec2;
type Rec1 is record
         data: Integer;
         next: Ptr2;
         end record;
type Rec2 is record
         data: Integer;
         next: Ptr1;
         end record;
```

Indeed, a type name that is used indirectly (after a pointer constructor) in Ada can be declared at any point later on in the same scope. In C there is even no requirement that the struct types be declared at all prior to their indirect use (as shown by the code in the book). This can actually be of practical use in complex data structures, since, for example, class records may wish to point to student records, and vice versa:

```
typedef struct ClassRec * ClassPtr;
typedef struct StudRec * StudPtr;
struct ClassRec
{ ...
  StudPtr students;
  ...
};
struct StudRec
{ ...
```

© Kenneth C. Louden 2003 All rights reserved

```
    ClassPtr classes;
    ...
};
```

6.15 The problem is that a **bool** value typically occupies one byte of memory, but an **int** value occupies 4. When, for example, the assignment **x.i = 20000** takes place, the four bytes of **x** will now appear in hexadecimal as **00004E20**. When **x.b = true** take place, the last two digits of this value (**20**) are replaced by the internal value for **true** (usually **1**), and the result is **00004E01**, or 19969 in decimal. Thus, this code will likely print out **19969**, and not **1** as one might expect. Of course, initializing **x.i** to **0** first will typically correct this problem, so that the following code is more likely to be correct (although there still might be architectures for which this will fail!):

```
union
{ int i;
  bool b;
} x;
…
x.i = 0;
x.b = true;
cout << x.i << endl;
```

6.17 The main problem in formulating an initialization requirement is to find distinguished values for all data types to use as the initial values. For built-in types like *integer*, *real*, *char*, or *boolean* this is not too difficult: 0, 0.0, the null character (chr(0)), and *false* are often used as default values. For arrays this can be extended to initialize each component with the initial value for its type, and for records similarly. Pointers can be initialized to null. For user-defined enumerations, the first value can be used as an initial value (so that **red** would be the initial value for the type **enum Color{red,green,blue}**). A problem occurs with (undiscriminated) unions, since now no guess can be made as to which fields are active. Similar problems may occur with specialized data types like *set* and *file*.

A programming language can allow the programmer to specify initial values. Usually this is done in variable declarations, as in the C declaration **int x = 0;**. A problem occurs for structured types, since some syntax must be invented to allow the assignment of multiple values. C uses curly brackets for arrays and structures:

```
int x[5] = {0,1,2,3,4};
struct { int i; double r; } y = {0,1.2};
```

It would also be good to allow the user to specify an intial value with each new type declaration, so that this initialization can be assumed for each variable of that type unless explicitly overriden. A problem with this is that two types that are equivalent must have the same initializations, so initial values can only be associated to declarations that create new types according to the type equivalence algorithm. The language itself must specify initializations for all predefined types.

Typical approaches taken by languages are exemplified by Pascal, and C. The language definition of Pascal states that no initializations for any variables can be assumed. Also, no provision is made in for user-defined initializations. (Nevertheless, both default and user-defined initializations are sometimes provided in implementations as a language extension.) In C, user-defined variable initializations are allowed, but there are no initializations associated with types. Default initializations are provided only for static variables:

> A static object not explicitly initialized is initialized as if it (or its members) were assigned the constant 0. The initial value of an automatic object not explicitily initialized is undefined. (Kernighan & Ritchie [1988], page 219).

© Kenneth C. Louden 2003 All rights reserved

Union variables are treated in a special way in C: the first member of the union is the only field to be initialized.

6.19 Yes: the comparison **`x == y`** generates no warning or error message (but always returns 0, since **`x`** and **`y`** are allocated different locations in memory). Indeed **`x`** and **`y`** are viewed as equivalent (constant) "pointer to integer" types; their sizes are not part of their types.

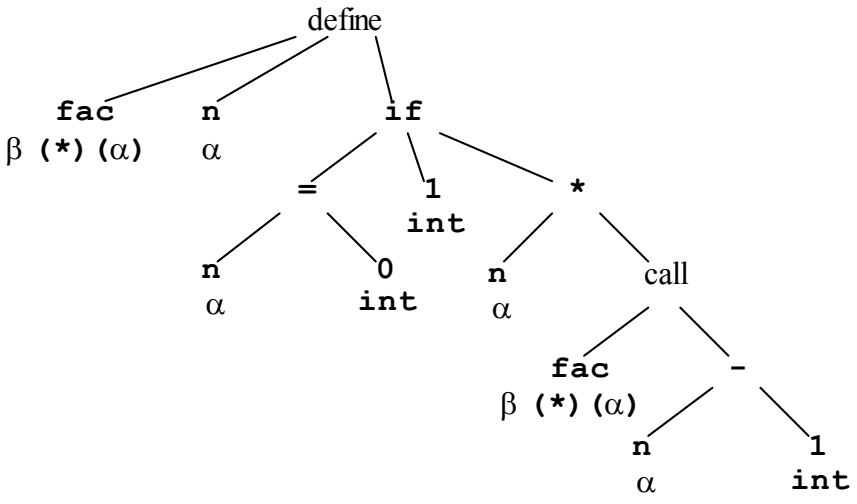
6.22 (a) Variables **`a`**, **`b`**, **`c`**, and **`d`** are all structurally equivalent.
(b) Variables **`a`** and **`b`** are name equivalent, and no other variables are name equivalent.
(c) Variables **`a`**, **`b`**, and **`c`** are equivalent in C, since C uses declaration equivalence (see p. 224), and the definitions of all three variable refer to the same **`struct`** declaration (**`Rec1`**). Variable **`d`** is not equivalent to the others, since it comes from a different **`struct`** declaration.

6.25 Equality testing can be used, even though it doesn't test the right thing: it always returns false, since it tests for identity in memory, i.e. pointer equality. But in this problem we don't care about the result, just whether a compiler message will be generated. Consider, for example, the following C code:

```
int x[10];
int y[5];
char z[20];
if (x == y) ; /* test ok */
if (x == z); /* test not ok */
```

For example, the Gnu C compiler generates the following warning message at the last line in the above code: "comparison of distinct pointer types lacks a cast". Thus, C uses structural equivalence for arrays. Note that assignment cannot be used to test this, since assignment to an array variable is illegal for other reasons.

6.28 A syntax tree for this definition, similar to the tree on page 240, is as follows, with the initial types filled in is as follows:



Now a simple left-to-right bottom-up traversal of this tree would discover first that α = **`int`** (from the comparison of **`n`** to **`0`**), and then that β = **`int`** (from the multiplication of **`n`** and the result of the recursive call to **`fac`**).

6.30 (a) (this function exists in the C++ standard library already):

```
template <typename T>
```

© Kenneth C. Louden 2003 All rights reserved

```
void swap(T& x, T& y)
{ T t = x;
  x = y;
  y = t;
}
```

(b)

```
fun swap(x,y) = let val t = !x in x := !y; y := t end;
```

6.36 Indeed C++ polymorphism *is* let-bound, since any polymophic function argument must be fully instantiated to an actual function at the point of call. Consider, for example, the following code (which uses the code of Exercise 6.34):

```
(1)   template <typename T>
(2)   T id (T x)
(3)   { return x; }

(4)   template <typename First, typename Second>
(5)   struct Pair
(6)   { First first;
(7)     Second second;
(8)   };

(9)   template <typename First, typename Second, typename Third>
(10)  Pair<First,Second> makePair(First x, Second y, Third f)
(11)  { Pair<First,Second> p;
(12)    p.first = f(x); p.second = f(y);
(13)    return p;
(14)  }

(15)  int main()
(16)  { Pair<int,char> z = makePair(1,'a',id); // what is id's type?
(17)  cout << z.first << endl;
(18)  cout << z.second << endl;
(19)  return 0;
(20)  }
```

The use of the uninstantiated **id** in line 16 causes the following error message to be generated by the Gnu C++ compiler:

```
no matching function for call to `makePair(int, char, <unknown  type>)'
```

Thus, we must supply an explicitly instantiated **id** function (such as **id<int>**), and this same function will be used for both calls to **f** in the **makePair** function.

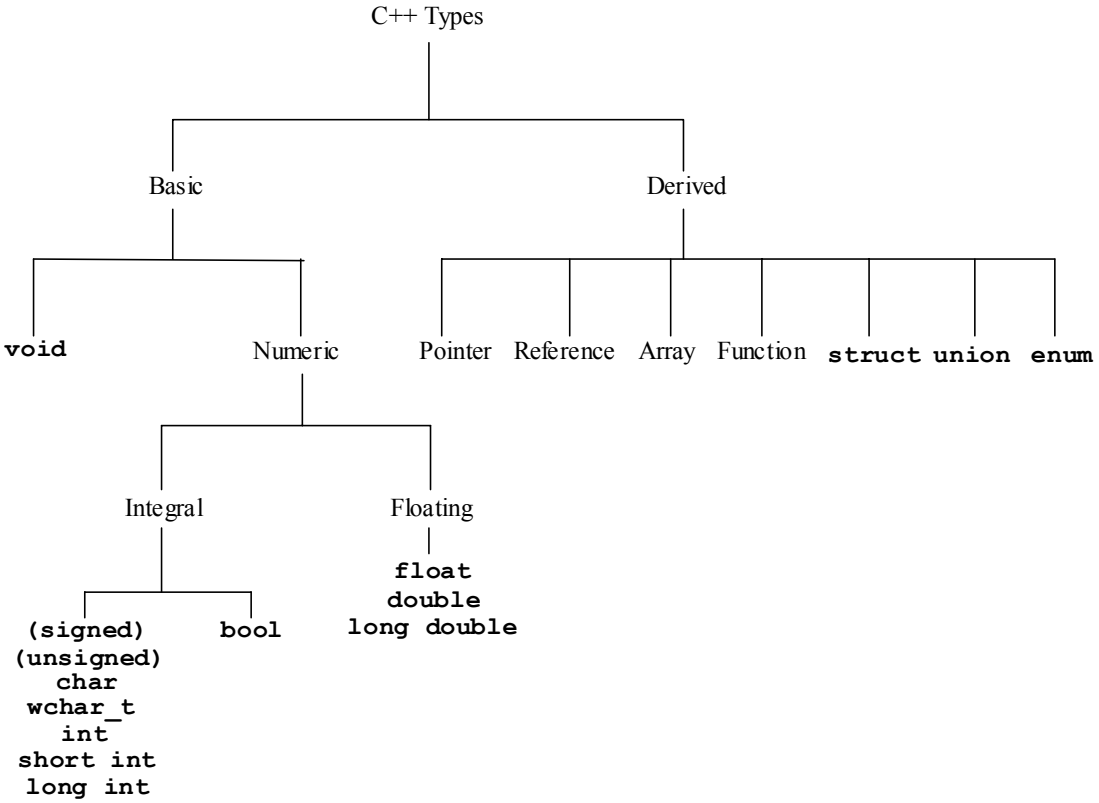
6.38 ML arrives at the type **fn : bool -> bool** for the function **f** given in this exercise, which is definitely not the most general type that could be achieved (**fn : ''a -> bool**). The reason is simple: during the type check, when the inference engine arrives at the call **(f false)**, **f** still has an indeterrminate type, and the Boolean argument cause the checker to restrict the parameter type to **bool**. The same thing happens in Haskell, except that it *is* possible to override the type checker with an explicit type that achieves the most general type (see Chapter 11 for the Haskell type system):

```
-- the following is legal Haskell:
f:: Eq a => a -> Bool
f x = if x == x then True else (f False)
```

© Kenneth C. Louden 2003 All rights reserved

This is not possible in ML, since the user cannot override the type checker in this way.

6.44 Stroustrup [1997], page 70, is a useful reference for the answer to this exercise. C++ of course adds the **`class`** type constructor to those of C. Additionally, C++ has reference types (see page 212), which differ from pointer types, so reference types also become new derived types in C++. To complete this view of C++ types, we should also note that the **`enum`** construct is raised in C++ to the level of a genuine type constructor (see Exercise 6.6 and its answer in the publicly available answers on my web site). C++ also provides a **`bool`** type, which is (unlike Java) an integral numeric type. Finally (for completeness), C++ offers a new **`wchar_t`** type to hold Unicode characters. Thus, the complete picture is as follows:



6.51 We consider the innermost expression first, namely **`-1 + x`**. The value -1 is of integer type, and since **`x`** is **`unsigned`**, -1 is converted to **`unsigned`** and the result is **`unsigned`**. Call this result **`a`**, and consider the next step in the expression, namely **`1.0 * a`**. The value 1.0 is of type **`float`** and is converted to **`double`**. Then **`a`** is converted to **`double`**, and the result is **`double`**. Call this result **`b`**. Finally, consider the outermost expression **`'0' + b`**. Since **`'0'`** is of type **`char`**, it is converted to **`int`**. Then, since **`b`** is **`double`**, this integer is converted to **`double`**, and the result is **`double`**.

The resulting value of the expression depends on (1) what kind of representation is used for integers and unsigned integers, and (2) what kind of representation is used for characters. Thus, different answers may be given, depending on the machine and the compiler. On most machines, the format for integer is two's complement, and unsigned integers have the same size as integers (the high bit is used for the value, rather than the sign). In this case **`-1`** = **`111...1`** internally, and **`1`** = **`000...01`**. Thus, regardless of the size of these numbers, **`-1 + x`** will always be 0 (ignoring the internal overflow for unsigned addition). Then the result of **`1.0 * a`** will be 1.0*0.0 = 0.0. Finally, if characters are represented as ASCII values (this is true most, but not all, of the time), **`'0'`** has ASCII value 48. This will be converted to 48.0, and the final result will be 48.0 + 0.0 = 48.0.

© Kenneth C. Louden 2003 All rights reserved

Chapter 7

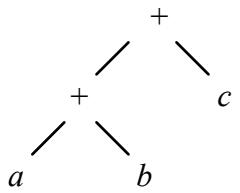
7.2 (b) *exp* → *op exp exp* | *number*
op → **+** | *
number → *number digit* | *digit*
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

7.6 The following program prints the two lines, instead of exiting abnormally (which it would do if either **&&** or **||** were not short-circuit):

```
#include <stdio.h>
main()
{ if (!((1==0)&&(1/0==1))) printf("%s\n","&& is short-circuit");
  if ((1==1)||(1/0==1)) printf("%s\n","|| is short-circuit");
  return 0;
}
```

7.7 A C program that does this appears in Figure 7.1, page 264 of the text. If we wanted to do an exhaustive test, we would also want to interchange the parameters **f()** and **x** in the call to **p**, as well as replace **x** by a more complicated expression, such as **2*x-1** or **abs(x)**. Whether optimizations are turned on or off can also have an effect on the result. Typical C compiler, in the absence of other issues, will generally evaluate arguments in a call right to left, for reasons discussed in Chapter 8. However, using the code of Figure 7.1 and the Gnu C compiler, I found that reversing **f()** and **x** as described above indeed reverses the order of evaluation. [This is in fact a code optimization whose properties are beyond the scope of this book.]

7.9 There is no contradiction. Indeed the + operator is always considered as a binary operator (that is, as a function of two parameters), and so the expression *a* + *b* + *c* is considered (by left associativity) to be equivalent to *plus*(*plus*(*a*,*b*), *c*). In this expression the only choice in the evaluation order is whether to evaluate the two parameters left to right or right to left. Left to right evaluation results in evaluating *plus*(*a*,*b*) [which results in evaluating *a* followed by *b*], and then evaluating *c*. Right to left evaluation results in evaluating *c*, and then evaluating *plus*(*a*,*b*) [which results in evaluating *b* followed by *a*]. This can also be visualized by using the syntax tree for *a* + *b* + *c*:



Left to right evaluation corresponds to a left to right traversal of this tree (so that the evaluation order of the leaves is *a*, *b*, *c*), while right to left evaluation corresponds to a right to left traversal (so that the evaluation order of the leaves is *c*, *b*, *a*). In each case associativity is preserved by the structure of the tree, and is independent of the evaluation order. The same argument applies to the subtraction operator.

7.12 (a) The first expression has a value in C, since short circuit evaluation is used for logical expressions (the "and" must of course be replaced by the C **&&** operator). The second expression, on the other hand, does not have a value (and its evaluation causes a "division by zero" runtime error).
(b) Neither expression has a value in Ada, since short circuit evaluation is not used for the "and" operator. If, however, the "and" operator is replaced by the "and then" operator, then the first expression does have a value in Ada as in C [in Ada syntax: **(x /= 0) and then (y mod x = 0)**]. Also as in C, the second expression still causes a runtime error (a **CONSTRAINT_ERROR** exception).

© Kenneth C. Louden 2003 All rights reserved

(c) The only way both of these expressions could have a value is if expressions were evaluated speculatively in arbitrary order, and any order that produces a value would be returned. Such language constructs are called **nondeterministic**, and it is possible for a language to specify a nondeterministic construct. Usually, however, this means that only one of the possible orders is selected for execution, and this would still not guarantee a result in this case. On the other hand, if a language allowed **parallel** evaluation of subexpressions, and in the case of Booleans ignored the failure of a subexpression computation if the final result was computable, then in this case it is possible for both expressions to have values. I know of no language that has this kind of flexibility.

7.14 A normal order evaluation would proceed as follow:

```
sum(cube(2),cube(3),cube(4)) => cube(2) + cube(3) + cube(4) => 2*2*2 +
3*3*3 + 4*4*4 => 8 + 27 + 64 => 99
```

An applicative order evaluation, on the other hand, would proceed as follows:

```
sum(cube(2),cube(3),cube(4)) => sum(2*2*2,3*3*3,4*4*4) => sum(8,27,64) => 8
+ 27 + 64 => 99
```

In both cases (since all expressions are distinct), there are 8 multiplications and 2 additions.

7.18 There is no ambiguity, because the **`switch`** statement has only one syntax rule (the following grammar rule taken from an online Yacc grammar):

```
selection_statement
      : IF '(' expression ')' statement
      | IF '(' expression ')' statement ELSE statement
      | SWITCH '(' expression ')' statement
      ;
```

To get cases into a **`switch`** statement, you must use a compound statement (curly brackets), and put the cases, including the **`default`**, inside the braces, which avoids any ambiguity.

7.25 In principle, yes, but there are several issues to worry about. Here, for example, is a definition in Haskell syntax (which can be executed in Haskell):

```
while e1 e2 =
  if e1 then
     do e2
        while e1 e2
  else return False
```

Notice that the result of this while expression is always the Boolean value **`False`**. This is the first issue: what value should a **`while`** function have? False is one possible answer; a void or unit value would be another. The second issue is: will delayed evaluation provide the correct evaluations for the expressions **`e1`** and **`e2`**? The answer here is yes: **`e1`** will be evaluated once (when the if is executed) and then (if its value is true) passed (unevaluated) to the next call of **`while`**. If the value of **`e1`** is true, **`e2`** is also evaluated and passed unevaluated to the next call of **`while`**. Finally, there is a third issue: if this function is to avoid an infinte recursive loop, then the value of **`e1`** must eventually change to false. This change represents a side effect, and side effects (as noted elsewhere) are generally incompatible with delayed evaluation. So this probably makes such a function relatively useless in practice.

7.29 Here is the applicable exerpt from the official Java grammar:

© Kenneth C. Louden 2003 All rights reserved

Statement:

StatementWithoutTrailingSubstatement
LabeledStatement
IfThenStatement
IfThenElseStatement
WhileStatement
ForStatement

StatementNoShortIf:

StatementWithoutTrailingSubstatement
LabeledStatementNoShortIf
IfThenElseStatementNoShortIf
WhileStatementNoShortIf
ForStatementNoShortIf

StatementWithoutTrailingSubstatement:

Block
EmptyStatement
ExpressionStatement
SwitchStatement
DoStatement
BreakStatement
ContinueStatement
ReturnStatement
SynchronizedStatement
ThrowStatement
TryStatement

IfThenStatement:

`if (` *Expression* **`)`** *Statement*

IfThenElseStatement:

`if (` *Expression* **`)`** *StatementNoShortIf* **`else`** *Statement*

IfThenElseStatementNoShortIf:

`if (` *Expression* **`)`** *StatementNoShortIf* **`else`** *StatementNoShortIf*

From this we can see how Java fixes the dangling else ambiguity. First statements that participate in the ambiguity (those with potential trailing substatements) are separated from those that don't. Next, each statement that can contain an ambiguity is given two forms: one without a "short" if (i.e. an if statement without an else part), and one with a potential short if. Then the "then" part in an if-then-else statement is required to have no short if statements, thus associating the else with the closest unmatched if. By my count, this solution requires seven extra grammar rules.

7.31 **(a)** 4; **(b)** 2; **(c)** 10; **(d)** 16. The general rule is that this code doubles the value of **`n`** if **`n`** > 0.

7.36 The Java code is as follows:

© Kenneth C. Louden 2003 All rights reserved

```
outer:
  for (int i = 0; i < n; i++)
  inner:
  { for (int j = 0; j < n; j++)
      if (x[i][j] != 0) break inner;
    System.out.println("First all-zero row is: " + i);
    break outer;
  }
```

This code is clearly more readable, since the two jumps are both of the same form (**`break inner`** and **`break outer`**, respectively), and they clearly indicate their purpose. The only possible confusion, in my view, is in the placement of the **`inner`** label: one might try putting it just before the inner **`for`** statement (inside the left brace), but this is wrong, since it results in an immediate fall-through to the print statement.

7.44 **(a)** Building on the answer to Exercise 7.43 (in the set of answers available on the book's web site), we can see that it would be easily possible to add cases to the **`valid`** test function that would distinguish white space from other extraneous tokens.
(b) However, this is not a reasonable alternative to skipping white space in a scanner, since white space between two digits should still be flagged as an error, while the code now does not: **`9 8 + 7 6`** is viewed as 98+76, which is really an error. A scanner that recognizes numbers and other tokens can distinguish white space between numbers from other white space, but the parser has trouble doing this (it can still be done by having a separate white space recognizer within the **`number`** function).

7.49 The trouble with this code is that it reports every error as a general error, and thus will report no range errors at all. This is because catch clauses act like cases in a switch – the first matching catch is always the one that is executed. Thus, this code can be easily fixed by changing the order of the catch clauses:

```
try
{ // do something
}
catch (range_error) { cout << "range error\n"; }
catch (...) { cout << "general error!\n"; }
```

Chapter 8

8.1 One *could* certainly arrange for a translator to leave out an implied return at the end of a function or procedure. The code would then fall through to the lexically following procedure or function. However, there are several problems with this mechanism. First, it would leave the runtime stack in a disturbed state, with the arguments to the call still on the stack, and the subsequent code would have no information on how to pop these arguments or readjust the stack. Second, this would make the execution behavior of the code dependent on the order in which the functions are arranged in the source code, and that is contrary to the notion of a function being an independent entity whose location in the source code should not affect its behavior (of course, nonlocal references within the function code also can violate this principle). Thus, in practice, such a mechanism would make little sense.

8.3 Such a prodedure (method) cannot be written in Java, since Java passes all arguments (even references) by value. For example, the following Java code fails to actually swap its integer values:

```
class SwapTry
{ public static void swap(int x, int y)
  { int temp = x; x = y; y = temp; }
```

© Kenneth C. Louden 2003 All rights reserved

```
  public static void main(String[] args)
  { int x = 1; int y = 2; swap(x,y);
     System.out.println(x);
     System.out.println(y);
  }
}
```

Even for objects this will fail, since object parameters, while references, are still passed by value (and thus not affected by reference reassignment):

```
import java.awt.Point;
class SwapTry2
{ public static void swap(Point x, Point y)
  { Point temp = x; x = y; y = temp; }

  public static void main(String[] args)
  { Point x = new Point(1,0); Point y = new Point(0,2); swap(x,y);
     System.out.println(x); // still (1,0)
     System.out.println(y); // still (0,2)
  }
}
```

The only way to perform such a swap of objects is to use mutator methods (or instance variable assignment) as provided by the class in question:

```
import java.awt.Point;
class SwapTry3
{ public static void swap(Point x, Point y)
  { Point temp = new Point(x); x.setLocation(y); y.setLocation(temp); }

  public static void main(String[] args)
  { Point x = new Point(1,0); Point y = new Point(0,2); swap(x,y);
     System.out.println(x); // now it works!
     System.out.println(y); //
  }
}
```

Of course, there is still no way to adapt this to non-object data such as integers (but wrapper classes could be used).

8.5 There are two things wrong here. First, similar to the Java examples of the answer to Exercise 8.3, this procedure will fail to assign a new value to `a` in the caller, since `a` is passed by value (even though it is a pointer). Second, even if we used a second level of indirection (or the pass by reference available in C++) this would not work, because arrays are *fixed* (constant) pointers allocated on the stack by the runtime system and therefore cannot be dynamically allocated. Thus the following C code still generates a compile-time error at the indicated place:

```
void init(int (*a)[], int size)
{ *a = (int*) malloc(size*sizeof(int)); /* error */
}

int main()
{ int (*a)[10];
  init(a, 5);
  return 0;
}
```

© Kenneth C. Louden 2003 All rights reserved

Of course, we could change the code to use all pointers rather than arrays and then this would work:

```
void init(int **a, int size)
{ *a = (int*) malloc(size*sizeof(int));
}

main()
{ int *a;
  init(&a, 5);
  return 0;
}
```

8.6 Yes, it would be legal to include all these declarations in the same file, since they all are equivalent and there is only one actual definition among them (the last declaration). Indeed, in a non-definitional declaration, the names of the parameters are completely immaterial, since declarations are only used to establish types. Thus, only the types of the parameters and the type of the returned value are needed.

8.9

pass by value	pass by reference	pass by value-result	pass by name
1 2 1 0 2 1 0	1 2 1 0 2 0 0	1 2 1 0 2 1 0	0 2 1 2 0 1 2

8.12 A function is supposed to only return a value, not also cause side effects. If the parameters were **out** or **in-out** parameters, the function would be written explicitly to cause side effects, which the Ada designers wanted to rule out. Note, however, that Ada functions can still cause side effects by non-local reference.

8.14

(a) An example of a rule 1 conflict is the program of Figure 8.1 (page 322): the local variable `j` inside function `p` conflicts with the variable `j` in procedure `q` that is passed to `p` in the call `p(i+j)` inside `q`.

(b) An example of a rule 2 conflict would be the same program as in part (a) *if* procedure `q` also had a local variable `i` that blocked the global `i` meant inside `p`. (The program as it stands has no rule 2 conflicts, since the same global `i` is meant inside `q` as inside `p`.)

(c) Only the local `j` of `p` needs replacement, and we replace it by the name `j1`. Then the body of `p` becomes:

```
  int j1 = (i + j);
  i++;
  return j1 + (i + j);
```

Inserting this code at the call site inside `q` then results in approximately the following code:

```
void q(void)
{ int j, j1, p;
  j = 2;
  i = 0;
  j1 = (i + j);
  i++;
  p = j1 + (i + j);
  printf("%d\n",p);
}
```

8.15

(b)

© Kenneth C. Louden 2003 All rights reserved

```
int sum (int (*a)(void), int* index, int size)
{ int temp = 0;
  for (*index = 0; *index < size; (*index)++)  temp += a();
  return temp;
}
```

Note that we have turned the **index** parameter into a pointer, since **sum** must update this parameter nonlocally. Now we define the function **f** in the scope of the variables **x** and **i** (see page 324) as

```
int f(void)
{ return x[i]; }
```

and pass it in the call to **sum** as follows:

```
sum(f,&i,10)
```

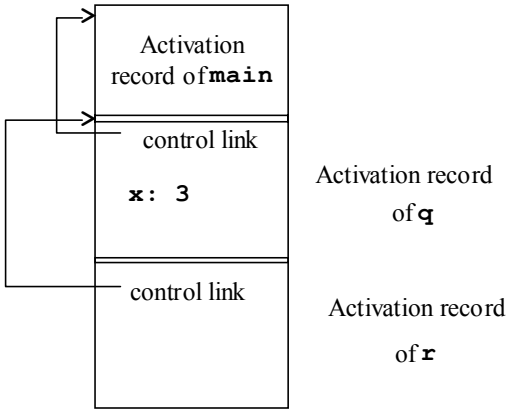
This will actually work in C exactly as pass by name.

8.16

(a) The scope of **x** begins just after its declaration as a parameter (i.e., just before the right parenthesis in the first line) and extends throughout the body of **p** (including the local declarations of **y** and **z**).
(b) The first declaration (of a procedure **p** with a parameter **p**) is legal in C, since a procedure declaration creates a new block (so that the parameters are contained in a nested block and do not conflict with the declaration of the procedure itself (which is in the surrounding block). The second declaration (of a parameter **x** and a local variable **x**) is also legal in C, since the body of a procedure is yet another new block (a compound statement, surrounded by curly brackets). Most compilers will issue a warning, however, since the parameter can never be accessed and thus is useless. Some compilers will even generate an error here, even though it is legal C.

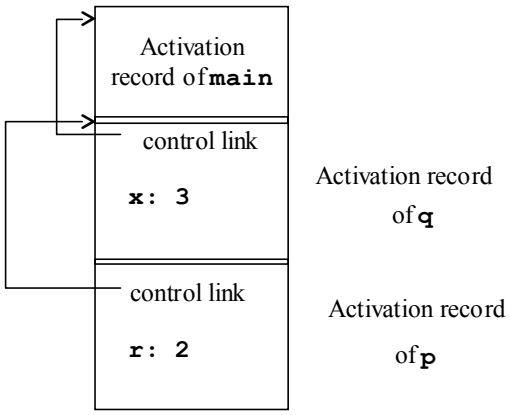
8.17

(a)



(b)

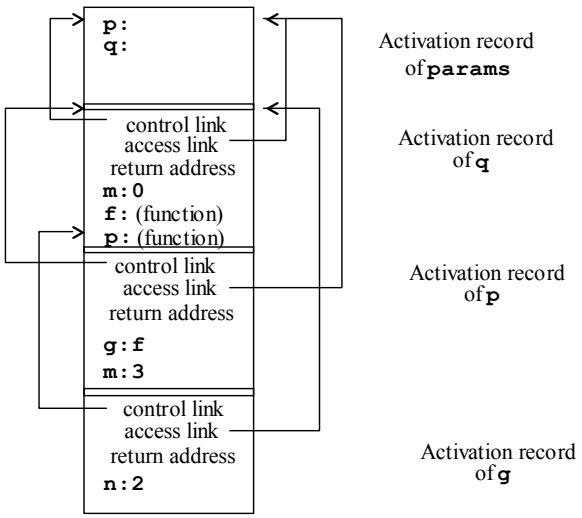
© Kenneth C. Louden 2003 All rights reserved



(c) The variable **r** is found in the current activation record by offset from the environment pointer (ep). The variable **x** is global and is not (usually) found in the stack of activation records. Instead, it is found by direct access as in Fortran.

8.19

(a)



(b) The program prints 2 (not 5), since **g** is **f** inside **q** when it is called. Thus, the parameter 2 and the value of **m** inside **q** (which is 0) are added, and then printed.

8.24 There are no parameters or return address, so these fields would be unnecessary in an activation record for a non-procedure block. However, such blocks are easy to implement without a new activation record: space for the new variables can be simply added to the end of the current activation record.

8.27

(a) Yes, this rule can be checked statically: the symbol table will tell us which variables are local and which are not, and any assignment or return statement can be checked so that it satisfies the rule.

(b) Unfortunately, this does not solve the problem. One way of bypassing the stated rule is with the following code:

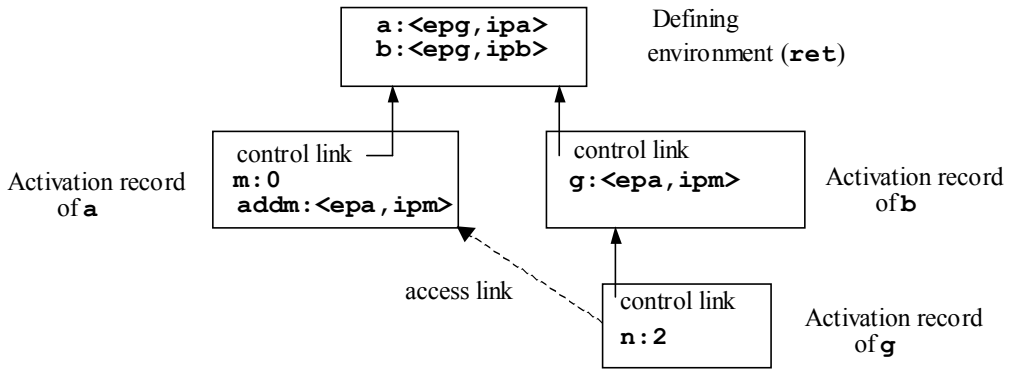
```
int* dangle(void)
{ int* y;
  int x;
  y = &x;
```

© Kenneth C. Louden 2003 All rights reserved

```
    return y;
}
```

Now the address of **x** is being assigned to a local variable, which is legal, and then the *value* of the local variable **y** is returned (also legal). Parameters can also be used to bypass the rule (parameters must be considered local). There is no good way to avoid this situation, as long as the address of a local variable is available via the **&** operator.

8.29 The text did not discuss the role of access links for dynamic environments. Here is a picture similar to that on page 339, with the access link for **g** added (since that is how **g** finds the nonlocal variable **m**):



The program should print 2 (= 2+0=n+m).

Chapter 9

9.3 The basic problem is that writing an axiom system for complex numbers that completely and consistently specifies all of the properties of the complex numbers that we might want is a nontrivial mathematical exercise. First, there will be a large number of such axioms. Second, it will not be clear whether such an axiom system is complete, consistent, or independent. It is true that mathematicians have largely solved these problems, but the required mathematical knowledge usually substantially exceeds that of typical programmers, or even computer scientists. Additionally, such an axiom system gives no implementation hints, unlike the equations that are based on real number properties, which can actually be used as code for an implementation.

9.9 Here is a queue ADT implementation in Ada. Note that the package specification is exactly the same as that of Figure 9.10.

Package Specification:

```
generic
  type T is private;
package Queues is
  type Queue is private;
  function createq return Queue;
  function enqueue(q:Queue;elem:T) return Queue;
  function frontq(q:Queue) return T;
  function dequeue(q:Queue) return Queue;
  function emptyq(q:Queue) return Boolean;
private
  type Queuerep;
  type Queue is access Queuerep;
```

© Kenneth C. Louden 2003 All rights reserved

```
end Queues;
```

Package Implementation:

```
package body Queues is

  type QueueNode;
  type QueuePtr is access QueueNode;

  type QueueNode is
  record
    data: T;
    next, prev: QueuePtr;
  end record;

  type Queuerep is
  record
    front, rear: QueuePtr;
  end record;

  function createq return Queue is
    temp: Queue;
  begin
    temp := new Queuerep;
    temp.front := null;
    temp.rear := null;
    return temp;
  end createq;

  function enqueue(q:Queue;elem:T) return Queue is
    temp: QueuePtr;
  begin
    temp := new QueueNode;
    temp.data := elem;
    if (q.rear /= null) then
       temp.prev := q.rear;
       q.rear.next := temp;
       temp.next := null;
       q.rear := temp;
    else
       temp.next := null;
       temp.prev := null;
       q.front := temp;
       q.rear := temp;
    end if;
    return q;
  end enqueue;

  function frontq(q:Queue) return T is
  begin
    return q.front.data;
  end frontq;

  function dequeue(q:Queue) return Queue is
  begin
    if q.front = q.rear then
       q.front := null;
       q.rear := null;
    else
```

© Kenneth C. Louden 2003 All rights reserved

```
          q.front := q.front.next;
        end if;
        return q;
      end dequeue;

      function emptyq(q:Queue) return Boolean is
      begin
        return q.front = null and q.rear = null;
      end emptyq;

    end Queues;
```

9.14

variables: *s*: SymbolTable; *n*, *m*: name; *v*: value
axioms:

lookup(create,*n*) = error
lookup(enter(*s*,*n*,*v*),*m*) = if *n* = *m* then *v* else lookup(*s*,*m*)
isin(create,*n*) = false
isin(enter(*s*,*n*,*v*),*m*) = if *n* = *m* then true else isin(*s*,*m*)

9.15 The operations create and enter are constructors. The operations lookup and isin are inspectors. Operation lookup is a selector, while isin is a predicate. By the rule of thumb, there should be four axioms (2 constructors × 2 inspectors).

9.17 Note that of the 8 operations, 3 are constructors and 5 are inspectors. Thus, we should need 15 axioms. Note also that the data type element *must* have comparison operations ≤ and ≥.

variables: *b*,*l*,*r*: bstree; *x*,*y*: element;
axioms:

empty(create) = true
left(create) = error
right(create) = error
data(create) = error
isin(create,*x*) = false
empty(make(*l*,*x*,*r*)) = false
left(make(*l*,*x*,*r*)) = *l*
right(make(*l*,*x*,*r*)) = *r*
data(make(*l*,*x*,*r*)) = *x*
isin(make(*l*,*x*,*r*),*y*) = if *y* = *x* then true else if *y* < *x* then isin(*l*,*y*) else isin(*r*,*y*)
empty(insert(*b*,*x*)) = false
left(insert(*b*,*x*)) = if empty(*b*) then create
else if *x* ≥ data(*b*) then left(*b*) else insert(left(*b*),*x*)
right(insert(*b*,*x*)) = if empty(*b*) then create
else if *x* ≤ data(*b*) then right(*b*) else insert(right(*b*),*x*)
data(insert(*b*,*x*)) = if empty(*b*) then *x* else data(*b*)
isin(insert(*b*,*x*),*y*) = if *y* = *x* then true else isin(*b*,*y*)

9.20

(b) front(enqueue(dequeue(enqueue(create,*x*)),*y*))
= front(enqueue(if empty(create) then create else
enqueue(dequeue(create),*x*),*y*)) (by the 6th axiom)
= front(enqueue(create,*y*)) (by the 1st axiom)
= if empty(create) then *y* else front(create) (by the 4th axiom)
= *y* (by the 1st axiom)

© Kenneth C. Louden 2003 All rights reserved

(c) front(dequeue(enqueue(enqueue(create,*x*)),*y*))
= front(if empty(enqueue(create,*x*)) then enqueue(create,*x*)
else enqueue(dequeue(enqueue(create,*x*)),*y*)) (by the 6th axiom)
= front(enqueue(dequeue(enqueue(create,*x*)),*y*) (by the 2nd axiom)
= front(enqueue((if empty(create) then create
else enqueue(dequeue(create),*x*)), *y*)) (by the 6th axiom)
= front(enqueue(create,*y*)) (by the 1st axiom)
= if empty(create) then *y* else front(create) (by the 4th axiom)
= *y* (by the 1st axiom)

9.21

(a) The *first* axiom is more appropriate for a symbol table in a translator for a block structured language, since it only deletes the most recent insertion of a variable name, not all occurrences of that name, as does the second axiom. Deleting only the most recent insertion is important when leaving a nested scope, since we want to uncover previous declarations of the same name. For example, if we had the symbol table

enter(enter(create,*x*,5),*x*,6)

then *x* currently has the value 6, but previously had the value 5. Applying the first axiom gives

delete(enter(enter(create,*x*,5),*x*,6),*x*) = enter(create,*x*,5)

so that now we have uncovered the previous value of 5 for *x*. On the other hand, applying the second axiom gives

delete(enter(enter(create,*x*,5),*x*,6),*x*) = delete(enter(create,*x*,5),*x*) = create

so that now all previous values have been deleted along with the current value. This second axiom is more appropriate in a situation where *x* is being assigned a new value during execution (so that the symbol table would now be representing the environment), since assignment should destroy any previously assigned values.

(b) Strictly speaking, delete is a constructor. However, in the text we have called an operation such as the dequeue operation of a queue (page 363) an inspector, since it can be written in terms of other constructors (and recursive calls to itself). [Strictly speaking we should have divided the constructor class into *essential* constructors, or generators, and other ("inessential") constructors, such as delete, pop, dequeue, etc.] Following the text (i.e. assuming delete is an inspector), we need only write one additional axiom for delete(create,*x*). Depending on our view, the equation could be either

delete(create,*x*) = create
or
delete(create,*x*) = error

Which equation we use of the two given in the text does not make any difference to our choice here.

9.24

type stack(element) **imports** boolean, integer
operations:

create	:		→ stack
push	:	stack × element	→ stack
pop	:	stack	→ stack
top	:	stack	→ stack

© Kenneth C. Louden 2003 All rights reserved

empty : stack → boolean
maxsize : → integer
full : stack → boolean
size : stack → integer

variables: *s*: stack; *x*: element; *n*: integer

axioms:

pop(create) = error
top(create) = error
empty(create) = true
full(create) = false
size(create) = 0
pop(push(*s*,*x*)) = if size(*s*) < maxsize then *s* else pop(*s*)
top(push(*s*,*x*)) = if size(*s*) < maxsize then *x* else top(*s*)
empty(push(*s*,*x*)) = false
full(push(*s*,*x*)) = size(push(*s*,*x*)) = maxsize
size(push(*s*,*x*)) = if size(*s*) < maxsize then size(*s*)+1 else size(*s*)

9.31

(a) The basic problem is that the make operation can mix up the order by using incorrect subtrees. However, the left and right operations also allow for insertions to occur into incorrect subtrees, thus possibly also destroying the order properties of a supertree, unless the left and right operations make complete copies of the subtrees they are returning. It is relatively easy to suppress the make operation in an implementation by making it private, but removing left and right operations would make the data operation relatively useless, and without these operations this ADT would become just a simplified version of the symbol table ADT (Exercise 9.14). Thus, to maintain safety the left and right operations must return copies of the actual subtrees, and this can lead to unexpected consequences for a user (e.g. subtrees no longer change to reflect subsequent insertions). Additionally, the left and right operations cannot be used internally by the implementation during an insert operation. We leave the implementation details (parts (b) and (c) of this problem) to the reader. The principal issue here is that a binary search tree is a mixture of two competing properties, those of a simple binary tree and those of an order relation. Perhaps the best approach would be to separate these issues altogether in an abstract specification.

9.34 Consider the term push(push(create)). Is push(push(create)) = push(create)? In the initial semantics these two terms are not equal, while in the final semantics they are (since there is no inspector that distinguishes them). To make the initial semantics equal to the final semantics, we must add the axiom push(push(*s*)) = push(*s*) for any item *s*. Alternatively, we could make the final semantics equal to the initial semantics by adding an inspector that can distinguish push(push(*s*)) from push(*s*). One way of doing this would be to define a count operation from items to the integers with the following axioms: count(create) = 0 and count(push(*s*)) = count(*s*) + 1 for any item *s*.

9.36

(a) An extensional definition for function equality is as follows: $f = g$ if the domain and range of f is the same as the domain and range of g, if for all x in the domain of f (or g), $f(x)$ is defined if and only if $g(x)$ is defined, and if for all x for which $f(x)$ is defined, $f(x) = g(x)$.

(b) This definition cannot be implemented in a programming language, since it would in general require checking function values for an infinite number of values. (Note, however, that if the functions are *finite*, as an array would be, then this definition *is* checkable.)

(c) ML does not allow function equality to be tested at all (any function type is not an equality type). C allows functions to be tested for equality, but the equality that is tested is pointer equality (i.e entry address), so a function is only equal to itself. A similar situation holds for Ada, except that the equality test is made explicitly on pointers by forcing the programmer to write the access attribute after the function name (as in `f'access = g'access`, etc.), thus making clear what the result will be.

© Kenneth C. Louden 2003 All rights reserved

Chapter 10

10.3 *Import* refers to the inclusion of code from outside the current program or program piece. This is usually expressed in C++ by an **`#include`** preprocessor directive. (In Ada the corresponding statement begins with the keyword **`with`**, and in Java with the keyword **`import`**.) Import of code makes the current code depend on the imported code. Frequently, this is needed to define a class composed of other, imported, classes (so that the defined class stands in a *has-a* relationship to these other classes), or to define objects of imported classes (so that the current code becomes a client of these classes).

By contrast, inheritance creates an *is-a* relationship between classes, so that inheritance *extends* an existing class, rather than simply *using* an existing class. Of course, in order to extend a class it may still need to be imported, since a dependency on the base class is also created. But the two concepts are distinct.

10.4 Overloading refers to the reuse of existing names, typically of functions, in new contexts, where the translator can distinguish the different uses by context. (In C++ and Java, overloading is only permitted if the different uses can be distinguished by the types of the parameters.) Inheritance also provides a mechanism for reusing the names of existing methods by redefining them in subclasses. In a sense, this use of inheritance is just implicit overloading on the class type of the implicit object parameter to the method. However, inheritance also makes dynamic binding available, while overloading is strictly a static concept. (Dynamic binding could be thought of as a restricted form of "dynamic overloading.")

10.7 In C it *is* possible to put function fields into **`struct`**s and to call them through the data object. However, there are several problems with this mechanism. First, there is no implicit **`this`** parameter, so all such functions operate as static methods and cannot refer to the current object unless a pointer to the current object is explicitly passed as a parameter to each function. Second, there are no protection mechanisms to ensure that data is manipulated only through the provided functions. Third, there is no automatic initialization through constructors, so such a **`struct`** would have to rely on the user to perform the appropriate initializations by hand. Finally (but not something that arises for the complex numbers), there is no inheritance or dynamic binding of such functions.

10.10

(a) The problem is that with private inheritance, objects of a derived class no longer behave as objects of the privately inherited base class:

```
class A
{ public:
    void f();
  ...
};

class B : private A
{...}; // no public redefinition of f

A x;
B y;
x = y; // must be an error since x.f() is then illegal
       // but the subtype principle says this is ok
```

(b) One way to avoid this problem – adopted by C++ – is to disallow assignments to variables of private base classes, so that the above assignment **`x = y`** generates an error message in C++. Thus, the subtype principle only applies to public inheritance in C++. An alternative is of course to disallow private (and protected) inheritance, since private inheritance is often used incorrectly in place of using a private data field (i.e. confusing the "is-a" and "has-a" relations).

(c) Java in fact chooses the second of the alternatives described in part (b): there is no protection applied to inheritance: it must always be public. Thus, the subtype principle always applies to Java inheritance.

© Kenneth C. Louden 2003 All rights reserved

10.12 If we could create an object of an abstract class, then we could call a method that has no code associated with it, and this would generate a runtime error. It is much better to generate a compile-time error instead (i.e. disallow the creation of such an object). Of course, variables of an abstract class may still be declared as long as this does not yet create an actual object. In C++, this means that a pointer variable can be declared, but a direct variable cannot be declared. It is still impossible to create an object of an abstract class. For instance, suppose **A** is an abstract class, and **B** is a derived class that is no longer abstract. Then we have the following situation in C++:

```
A a; // illegal
A* p; // legal
p = new A; // illegal
p = new B; // legal
```

In Java, all object variables are implicitly pointers, so abstract class (and interface) variables can always be declared. However, as in C++, **new** can only be called on concrete subclasses.

10.13 Abstract methods (deferred methods in C++ terminology) are *required* to be virtual (i.e. obey dynamic binding) in C++, and this makes sense, since you always want to call a redefined deferred method, rather than the corresponding undefined base method. In Java, of course, there is no choice: *all* methods are dynamically bound, whether or not they are abstract.

10.15 A type (or data type) represents a collection of *values* with similar properties, while a class represents a collection of *objects* with similar properties. In a very pure object-oriented language like Smalltalk, there are no values, only objects, and so the class structure takes the place of types. In a less pure language, the class structure can be incorporated into the type system, and so classes can also become types. This is the case for C++ and Java, where classes are used in static type checking.

10.20 The code prints:

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

The first call to **r** calls **A.p** and **A.q**, since dynamic binding is not used in C++ for direct variables, only for reference or pointer variables. Indeed, the variable **a** is only allocated space for an **A** object, so must always be an **A** object. The second call to **r** is through the pointer **ap**, however, so dynamic binding *is* used, and since a **B** object is currently pointed to by **ap**, the **B** version of the method **p** is called. However, **q** is not declared virtual in **A**, so again C++ does not use dynamic binding for calls to **A.q**, and thus **A.q** is still called. Finally, **ap** is set to point to a **C** object before the final call to **r**, and since **r** is overridden in C and **r** is virtual, **C.r** is called, resulting in calls to **C.q** and **C.p**, which is why **C.q** is printed, even though **q** is not virtual. Finally, the call to **C.p** is actually a call to **B.p**, since **p** is not redefined in **C** but is redefined in **B**.

10.23 The code for the methods of the **False** object is simply the reverse of the code at the top of page 451:

```
ifTrue : aBlock
  ↑ nil
ifFalse: aBlock
  ↑ aBlock value
```

© Kenneth C. Louden 2003 All rights reserved

10.25 Most of this has already been explained in the text at the bottom of page 449. The only point to realize is that **self** refers to the class **LinkableObject** itself, not an object of **LinkableObject**. Thus, the call to **self new** is actually a call to the parameterless **new** method of the **LinkableObject** class, which always exists (and is actually a call to the **new** method of the class **Class** parent object of every class object).

10.28

```
public class Queue
{ public Queue()
  { rear = null; }

  public boolean empty()
  { return rear == null; }

  public Object front()
  { return rear.next.data; }

  public void dequeue()
  { Linkable temp = rear.next;
    if (temp == rear) rear = null;
    else rear.next = rear.next.next;
  }

  public void enqueue( Object item)
  { Linkable temp = new Linkable();
    temp.data = item;
    if (empty())
    { rear = temp;
      rear.next = temp;
    }
    else
    { temp.next = rear.next;
      rear.next = temp;
      rear = temp;
    }
  }

  protected class Linkable
  { public Object data;
    public Linkable next;
  }

  private Linkable rear;
}
```

10.30 A **ComparableObject** abstract class would have to define a comparison operator to another object of the same class. In Java, the standard language library already contains a **Comparable** interface that will do this job, with a single (abstract) method declared as follows:

```
public int compareTo(Object o)
```

Note that this method is defined for any **Object** parameter (to avoid compile-time errors), but can only compare objects of the same class. Thus, it generates an exception (a **ClassCastException**) if the object **o** is not a member of the same class as the object on which **compareTo** is called. Note also that it returns an integer value, with negative values (typically –1) implying "less than," positive values (usually 1) implying "greater than," and zero implying equality (which should be the same as a call to **equals**). Now,

© Kenneth C. Louden 2003 All rights reserved

using an inner class to represent links (see Exercise 10.28), we would define a **SortedList** class to store only **Comparable** objects, and the inner **Linkable** class would also only accept **Comparable** objects. Alternatively, we could define an abstract **ComparableLinkableObject** class that inherits from **ComparableObject** and implements the **Comparable** interface (with an abstract method), and use that as the storable element in the **SortedList** interface. In C++ this would look similar, except that we could also overload one of the comparison operators (**<**, say) in the definition of a **ComparableObject** (and no **Object** superclass is available, so we have to use the same class as a parameter):

```
class ComparableObject
{ public:
    virtual int operator < (const comparableObject&) const
      = 0 ; // other comparisons can be defined using
           // this one
};
```

10.35 Here is a solution in C++:

```
class LinkableObject
{ public:
      LinkableObject() : link(0) {}
      LinkableObject(LinkableObject *p = 0) : link(p) {}
      LinkableObject* next() { return link;}
      void linkto(linkableObject* p) { link = p;}
      virtual void printLinkable() = 0; // pure virtual
  private:
      linkableObject* link;
};

class Queue
{ public:
      Queue(): rear(0) {}
      bool empty() { return rear == 0; }
      void enqueue(LinkableObject* item);
      void dequeue();
      LinkableObject* front() { return rear->next(); }
      void printQueue(); // not virtual or pure
      ~Queue();
  protected:
      linkableObject* rear;
};

void Queue::printQueue()
{ linkableObject* temp = rear;
  if (temp == 0) return;
  do
  { temp = temp->next();
    temp->printLinkable();
  } while (temp != rear);
}
```

A similar solution can be written in Java, although, in truth, a solution that overrides **toString** in **Queue** and any descendant of **LinkableObject** is better, since it automatically gives printing capability via **System.out.print**.

10.38 (b) The question is whether a SyntaxTree might have *some* implementation details associated with it, such as a non-abstract method, a constructor, or a data field. As shown in the exercise, SyntaxTree has

© Kenneth C. Louden 2003 All rights reserved

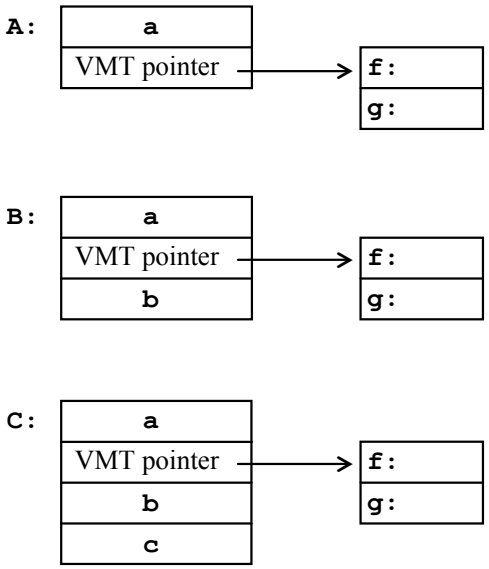
none of these, so it might just as well be declared as an interface. I can think of no situation where constructors, data, or non-abstract methods could be given in advance of some knowledge of the type of tree node desired.

10.41

```
import java.lang.reflect.Method;

public class PrintMethodUtil
{ public static void printMethods(Object obj)
  { Method[] methods = obj.getClass().getMethods();
    for (int i = 0; i < methods.length; i++)
      System.out.println(methods[i].getName());
  }
}
```

10.44 In C++ a method only obeys dynamic binding if it declared with the keyword **`virtual`**, and in Figure 10.19 (pages 458-9) **`f`** is not declared **`virtual`**. Thus, dynamic binding does not apply to **`f`**, so any calls to **`f`** are resolved statically (i.e. at compile-time), just as a normal function call, and **`f`** does not appear in the VMT. In detail, if a variable **`a`** is declared of class **`A`**, a call **`a.f()`** is found by looking at the declaration of **`A`** at compile time, and calling the function **`f`** found there.

10.48



Chapter 11

11.1

(a)

```
double power (double a, int b)
{ if (b == 0) return 1;
  else return a * power(a,b-1);
}
```

(b)

```
double power1(double a, int b, double sofar)
{ if (b == 0) return sofar;
```

© Kenneth C. Louden 2003 All rights reserved

```
  else return power1(a,b-1,a*sofar);
}

double power(double a, int b)
{ return power1(a,b,1.0); }
```

11.5

(a) The factorial function is referentially transparent, because its value depends only on the value of its parameter.
(b) Any input function is **not** referentially transparent, since it will return an unpredictable value, depending on what the user does. Indeed, such an input function has no parameters, and a referentially transparent function with no parameters must be a constant.
(c) A function that counts the number of times it has been called cannot be referentially transparent, since it has no parameters but is not constant.

11.8

(a)

Scheme:

```
(define (len1 lis sofar)
  (if (null? lis) sofar
      (len1 (cdr lis) (+ sofar 1))))

(define (len lis) (len1 lis 0))
```

ML:

```
fun len1 [] sofar = sofar
 | len1 (x::xs) sofar = len1 xs (sofar+1);

fun len lis = len1 lis 0;
```

Haskell:

```
len1 [] sofar = sofar
len1 (x:xs) sofar = len1 xs (sofar+1)

len lis = len1 lis 0
```

(b)

Scheme:

```
(define (maxmin L)
  (cond
    ((null? L) '()) ;; empty list represents error
    ((null? (cdr L)) (list (car L) (car L))) ;; list of 2 items
    (else (let ((mmtemp (maxmin (cdr L)))
                (first (car L)))
            (cond
              ((> first (car mmtemp))
               (cons first (cdr mmtemp)))
              ((< first (car (cdr mmtemp)))
               (list (car mmtemp) first))
              (else mmtemp))))))
```

ML:

© Kenneth C. Louden 2003 All rights reserved

```
fun maxmin [] = raise Fail "maxmin: empty list"
 | maxmin [x] = (x,x) (* a pair *)
 | maxmin (x::xs) =
     let val (max,min) = maxmin xs in
         if x < min then (max,x)
         else if x > max then (x,min)
         else (max,min)
     end;
```

Haskell:

```
maxmin [] = error "maxmin: empty list"
maxmin [x] = (x,x) -- a pair
maxmin (x:xs) =
  let (a,b) = maxmin xs in
    if x > a then (x,b)
    else if x < b then (a,x)
    else (a,b)
```

(Note about this Haskell definition: even though we have defined **maxmin** in the first line to call **error** on the empty list, if we try the call **maxmin []**, Haskell will generate an "overload resolution" error instead. This is because the empty list is too unspecified in this expression (it has type "list of any"). To remove this problem, one can either give **maxmin** an explicit type (such as **[Int] -> (Int,Int)**) or specify the type of the empty list in the call, i.e. **maxmin ([]::[Int])** to see the explicit call to **error**.)

(d)

Scheme:

```
(define (ex11.8d)
  (display "Enter a 0-ended list of integers:")
  (newline)
  (let ((L (collect)))
    (display "The list is: ")
    (display L)
    (newline)
    (let ((mm (maxmin L)))
      (display "The maximum of the list is ")
      (display (car mm))
      (newline)
      (display "The minimum of the list is ")
      (display (car (cdr mm)))
      (newline))))
```

(f)

Scheme:

```
(define (merge fst snd)
  (cond
    ((null? fst) snd)
    ((null? snd) fst)
    (else
     (let ((x (car fst)) (y (car snd)))
       (if (< x y) (cons x (merge (cdr fst) snd))
           (cons y (merge fst (cdr snd))))))))
```

© Kenneth C. Louden 2003 All rights reserved

```
(define (split lis)
  (cond
    ((null? lis) (cons '() '()))
    ((null? (cdr lis)) (cons (list (car lis)) '()))
    (else
     (let ((a (car lis)) (b (car (cdr lis))) (c (split (cdr (cdr lis)))))
       (cons (cons a (car c)) (cons b (cdr c)))))))

(define (msort lis)
  (cond
    ((null? lis) '())
    ((null? (cdr lis)) lis)
    (else
     (let* ((c (split lis)) (fst (car c)) (snd (cdr c)))
       (merge (msort fst) (msort snd))))))
```

ML:

```
open List;

fun merge [] ys = ys
 |  merge xs [] = xs
 |  merge (x::xs) (y::ys) =
      if x <= y then x :: merge xs (y::ys)
      else  y :: merge (x::xs) ys;

fun msort [] = []
 |  msort [x] = [x]
 |  msort lis =
       let val n = (length lis) div 2;
           val fsthalf = take(lis,n);
           val sndhalf = drop(lis,n);
           val lis1 = msort fsthalf;
           val lis2 = msort sndhalf;
       in
         merge lis1 lis2
       end;
```

Haskell:

```
msort [] = []
msort [x] = [x]
msort lis = merge lis1 lis2
            where
            n = (length lis) / 2
            fsthalf = take n lis
            sndhalf = drop n lis
            lis1 = msort fsthalf
            lis2 = msort sndhalf
```

(The following is from the standard prelude:)

```
merge                :: Ord a => [a] -> [a] -> [a]
merge []     ys       = ys
merge xs     []       = xs
merge (x:xs) (y:ys)
        | x <= y      = x : merge xs (y:ys)
        | otherwise   = y : merge (x:xs) ys
```

© Kenneth C. Louden 2003 All rights reserved

(g)

Haskell: (this is also a solution to Exercise 11.30(b))

```
pyth n = [(x,y,z) | x <- [1..n], y <- [x..n], z <- [y..n], z^2==x^2+y^2]
```

(i)

Scheme:

```
(define (inc_n n)
  (lambda (x) (+ n x)))
```

ML:

```
fun inc_n x y = x + y;
```

Haskell:

```
inc_n = (+)
```

or

```
inc_n x y = x + y
```

(Note: the first Haskell definition of **inc_n** results in restricting the type of **inc_n** to integers, by a quirk of the Haskell overloading system, while the second definition gives **inc_n** the same type as the overloaded **+** operator.)

11.10

(a)

```
(define (member x L)
  (cond
    ((null? L) #f)
    ((equal? x (car L)) #t)
    (else (member x (cdr L)))))

(define (insert x L)
  (if (member x L) L
      (cons x L)))
```

(c)

```
member x [] = False
member x (a:as) =
  if x == a then True
  else member x as

insert x lis =
   if member x lis then lis
   else x:lis
```

11.15

a of the first list = **(car (car (car (car firstlist)))))**
3 of the second list = **(car (car (cdr (car (cdr secondlist)))))**
c of the third list = **(car (car (car (cdr thirdlist))))**

11.18

```
("c" ("a" () ("b" () ())) ("f" () ()))
```

© Kenneth C. Louden 2003 All rights reserved

11.19

```
(define (insert item tree)
  (cond
    ((null? tree) (list item '() '()))
    ((string<? item (data tree))
     (list (data tree)
           (insert item (leftchild tree)) (rightchild tree)))
    (else (list (data tree) (leftchild tree)
                (insert item (rightchild tree))))))
```

11.24 The function **`(make-double - )`** is the constant 0 function whose value is zero for all arguments. The function **`(make-double / )`** is the constant 1 function EXCEPT for 0, where it is undefined.

11.29

(a)

```
len :: Num a => [b] -> a
```

(b)

```
maxmin :: Ord a => [a] -> (a,a)
```

(f)

```
mergesort :: Ord a => [a] -> [a]
```

(h)

```
twice :: (a -> a) -> a -> a
```

(i)

```
inc_n :: Num a => a -> a -> a
```

(or **`Integer -> Integer -> Integer`** – see the answer to Exercise 11.8(i))

11.30

(b)

```
pyth n = [(x,y,z) | x <- [1..n], y <- [x..n], z <- [y..n], z^2==x^2+y^2]
```

(c)

```
properfactors n =
    [i | i <- [1.. div n 2], mod n i == 0]

perfects =
    [n | n <- [2..], sum (properfactors n) == n]
```

11.32 One advantage to requiring all functions to be total would be that the programmer is forced to write explicit error cases for those values for which a function is not defined, as for example the **`maxmin`** function of Exercise 11.8(b) (see the ML solution above), where a warning is generated if the **`maxmin`** function has no definition for the empty list (in fact, **`maxmin`** has no value for the empty list, so an exception must be generated). Of course, the execution system itself could generate such error messages – if we leave off the first line in the definition of **`maxmin`**, ML responds to a call **`maxmin []`** in essentially the same way, except without the nice error message:

```
- maxmin [];

uncaught exception nonexhaustive match failure
  raised at: ex11-8.sml:3.6
-
```

© Kenneth C. Louden 2003 All rights reserved

But it can be argued that ML should not even allow the function to be executed if the programmer does not state explicitly when an error will occur. There is a certain overhead to checking this, since the translator must do a case analysis. However, the big problem with this is that in recursive situations, the compiler cannot tell when an infinite recursive loop will be entered, and (at least in theory) there may be cases when the programmer cannot tell either:

```
fun mystery 0 = 1
  | mystery 1 = 2
  | mystery n = if n mod 2 = 0 then mystery (n div 2)
                else mystery (n + 5);
```

The standard computer solution to situations when not all cases can be checked is to check none of them (perhaps because checking some gives a false sense of security).

11.36

(a) in Scheme:

```
(define (filter p L)
  (cond
    ((null? L) L)
    ((p (car L)) (cons (car L)
                       (filter p (cdr L))))
    (else (filter p (cdr L)))))

(define (not-divisible-by n)
  (lambda(m) (not (= 0 (remainder m n)))))

(define (sieve L)
  (if (null? L) L
      (cons (car L)
            (sieve
               (filter (not-divisible-by (car L))
                       (cdr L))))))

(define (intlist m n) ;; from page 509
  (if (> m n) '() (cons m (intlist (+ 1 m) n))))

(define (primesto n)
  (sieve (intlist 2 n)))
```

(b)

```
(define (filter p L)
  (cond
    ((null? L) L)
    ((p (car (force L)))
     (delay (cons (car (force L))
                  (filter p (cdr (force L))))))
    (else (filter p (cdr (force L))))))

(define (sieve L)
  (delay
    (cons (car (force L))
          (sieve (filter
                  (not-divisible-by (car (force L)))
                  (cdr (force L)))))))

(define (intsfrom m) ;; adapted from page 510
```

© Kenneth C. Louden 2003 All rights reserved

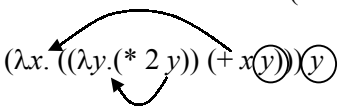
```
    (delay (cons m (intsfrom (+ 1 m)))))

(define primes
  (sieve (intsfrom 2)))
```

11.40 (Note: see code correction in the Errata for this exercise.) Here is the output from one Scheme interpreter, showing that **show** is called only once on each number:

```
> (take 5 L)
1
2
3
4
5
(1 2 3 4 5)
> (take 7 L)
6
7
(1 2 3 4 5 6 7)
>
```

11.45

(a) Free and bound variables (bound vars have arrows to their bindings, free are circled):



Normal order reduction:

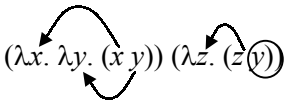
$(\lambda x.\ ((\lambda y.(*\ 2\ y))\ (+\ x\ y)))\ y \Rightarrow_\beta ((\lambda y.(*\ 2\ y))\ (+\ y\ y)) \Rightarrow_\beta (*\ 2\ (+\ y\ y))$

[= $(*\ 4\ y)$ by properties of the constants]

Applicative order reduction:

$(\lambda x.\ ((\lambda y.(*\ 2\ y))\ (+\ x\ y)))\ y \Rightarrow_\beta (\lambda x.\ (*\ 2\ (+\ x\ y)))\ y \Rightarrow_\beta (*\ 2\ (+\ y\ y))$

(b) Free and bound variables (bound vars have arrows to their bindings, free are circled):



Normal order reduction:

$(\lambda x.\ \lambda y.\ (x\ y))\ (\lambda z.\ (z\ y)) \Rightarrow_\alpha (\lambda x.\ \lambda w.\ (x\ w))\ (\lambda z.\ (z\ y)) \Rightarrow_\beta \lambda w.\ ((\lambda z.\ (z\ y))\ w)$
$\Rightarrow_\beta \lambda w.\ (w\ y)$

Applicative order reduction: Same as normal order.

11.46

Using the definition of cons, we have cons $a\ b = (\lambda x.\ \lambda y.\ \lambda f.\ f\ x\ y)\ a\ b = \lambda f.\ f\ a\ b$ and
(car (cons $a\ b$)) = (car $(\lambda f.\ f\ a\ b)$) = $((\lambda z.\ z\ (\lambda x.\ \lambda y.\ x))\ (\lambda f.\ f\ a\ b)) =_\beta (\lambda f.\ f\ a\ b)\ (\lambda x.\ \lambda y.\ x)$
$=_\beta (\lambda x.\ \lambda y.\ x)\ a\ b =_\beta (\lambda y.\ a)\ b = a$ and (cdr (cons $a\ b$)) = (cdr $(\lambda f.\ f\ a\ b)$) =
$((\lambda z.\ z\ (\lambda x.\ \lambda y.\ y))\ (\lambda f.\ f\ a\ b)) =_\beta (\lambda f.\ f\ a\ b)\ (\lambda x.\ \lambda y.\ y)$
$=_\beta (\lambda x.\ \lambda y.\ y)\ a\ b =_\beta (\lambda y.\ y)\ b = b$

11.47

(c) $add = \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$
$mult = \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ (n\ f)\ x$

© Kenneth C. Louden 2003 All rights reserved

(d) Here is Haskell code that will demo these operations:

```
zero f x = x
one f x = f x
two f x = f (f x)
three = add one two
four = mult two two
seven = add three four
twentyeight = mult four seven
add m n f x = m f (n f x)
mult m n f x = m (n f) x
shownum num = num (+1) 0
```

A call **`shownum twentyeight`** will now print the value 28.

11.49 For any H, we have

$$Y\,H = (\lambda h.\ (\lambda x.\ h\ (x\ x))\ (\lambda x.\ h\ (x\ x))\ )\ H \Rightarrow\ (\lambda x.\ H\ (x\ x))\ (\lambda x.\ H\ (x\ x)) \Rightarrow$$
$$H\ (\ (\lambda x.\ H\ (x\ x))\ (\lambda x.\ H\ (x\ x))\ ) \Leftarrow H\ (Y\,H)$$

so equivalence is proved.

11.50

(a) The type of *y* in Haskell is **`(a -> a) -> a`**

(b) The following code *does* in fact work in Haskell:

```
y h = h (y h)
h_fact g n = if n == 0 then 1 else n * g (n-1)
fact = y h_fact
```

The reason is that Haskell only evaluates enough of the potentially infinite calls to y in the first definition in order to compute the requested value, by lazy evaluation, as in the following normal order reductions:

```
fact 2 = y h_fact 2 = h (y h) 2
=> if 2 == 0 then 1 else 2 * (y h)(2 - 1)
=> 2 * (y h)(2 - 1)
=> 2 * (h (y h)) 1
=> 2 * (if 1 == 0 then 1 else 1 * (y h)(1 - 1))
=> 2 * (1 * (h (y h)) 0)
=> 2 * 1 * (if 0 == 0 then 1 else 1 * (y h)(0 - 1))
=> 2 *  1 * 1 = 2
```

In Scheme, on the other hand, the definitions above lead to an immediate infinite recursive loop in the third line:

```
(define (y h) (h (y h)))
(define (h_fact g)
  (lambda (n) (if (= n 0) 1 (* n (g (- n 1))))))
(define fact (y h_fact))
```

To make this work in Scheme, we have to use **`delay`** and **`force`**:

© Kenneth C. Louden 2003 All rights reserved

```
(define (y h) (h (delay (y h))))
(define (h_fact g)
  (lambda (n) (if (= n 0) 1 (* n ((force g) (- n 1))))))
(define fact (y h_fact))
```

Chapter 12

12.4

raining $\vee$ snowing $\rightarrow$ precipitation
freezing $\wedge$ precipitation $\rightarrow$ snowing
$\neg$ freezing $\wedge$ precipitation $\rightarrow$ raining
$\rightarrow$ snowing

12.5

```
precipitation :- raining.
precipitation :- snowing.
snowing :- freezing,precipitation.
raining :- not(freezing),precipitation.
snowing.
```

Depending on which Prolog system is used, Prolog may either answer "no" when given the query "freezing," or may give an error message that no clause for "freezing" exists. In fact, given these clauses, it cannot be proven whether or not it is freezing, and so the closed world assumption would force Prolog to the conclusion that it is not freezing. Thus, the error message is perhaps the better approach, since it indicates a lack of information. For those Prolog systems that answer "no" to the previous question (or for those with an error message, the clause `freezing :- fail` can be added to the above), Prolog goes into an infinite loop when given the query "raining." This is because there is only one clause whose head matches `raining`; this clause in turn sets up the subgoals `not(freezing)` and `precipitation`. As we have just seen, `not(freezing)` succeeds because `freezing` fails, and so the goal `precipitation` is tried. The first clause for `precipitation` however leads back to the original goal `raining`. Thus, an infinite loop is entered. This particular loop can be avoided by reversing the order of the first two clauses. Then Prolog will answer "yes" to the "raining" query. However, this is not a completely satisfactory solution, because queries in other situations may still get into an infinite loop. For instance, if we add the fact `freezing` to the program, then Prolog will still get into a loop on the query `snowing`. Again in this example, if we write the facts first, the loop will be avoided. However, no order will prevent loops in all cases.

12.8 There are several possibilities. Either we could write

```
legs(X,4):- mammal(X),arms(X,0).
legs(X,2):- mammal(X),arms(X,2).
```

or

```
arms(X,0):- mammal(X),legs(X,4).
arms(X,2):- mammal(X),legs(X,2).
```

or both. Which is best depends on what facts are given. If the facts are about arms, then the first is best; if the facts are about legs, then the second is best. Using both will cover both possibilities but runs the risk of loops.

12.9 The answer here depends on which solution we adopt in Exercise 12.8. If we use the first solution (because there is a fact about the arms of a horse), then Prolog will in fact derive that a horse has 4 legs. If

© Kenneth C. Louden 2003 All rights reserved

we use the second solution, then Prolog will not derive any facts about the number of legs, since there are no clauses that infer anything about legs. If all four clauses are used together, then it depends on where the fact `arms(horse,0)` is entered. If it entered before all other clauses, then Prolog will still be able to derive that a horse has 4 legs. From this exercise and Exercise 12.5, we can see that it is always best to enter facts first in a Prolog program.

12.10

```
grandparent(X,Y):- parent(X,Z),parent(Z,Y).
sibling(X,Y):- parent(Z,X),parent(Z,Y).
cousin(X,Y):- parent(Z,X),parent(W,Y),sibling(Z,W).
```

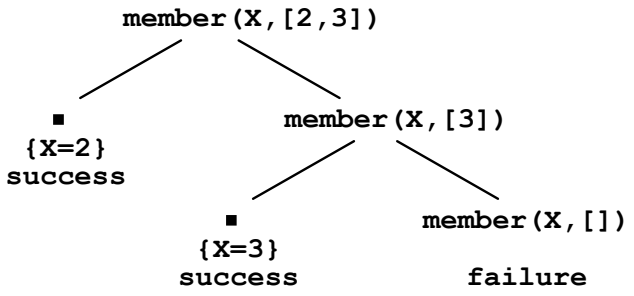
An alternative for cousin is:

```
cousin(X,Y):- grandparent(Z,X),grandparent(Z,Y).
```

12.19

```
member(X,[X|_]).
member(X,[_|Y]):- member(X,Y).
```

The query `member(X,[2,3])` produces the two answers `X = 2` and `X = 3`. This results from the following search tree:

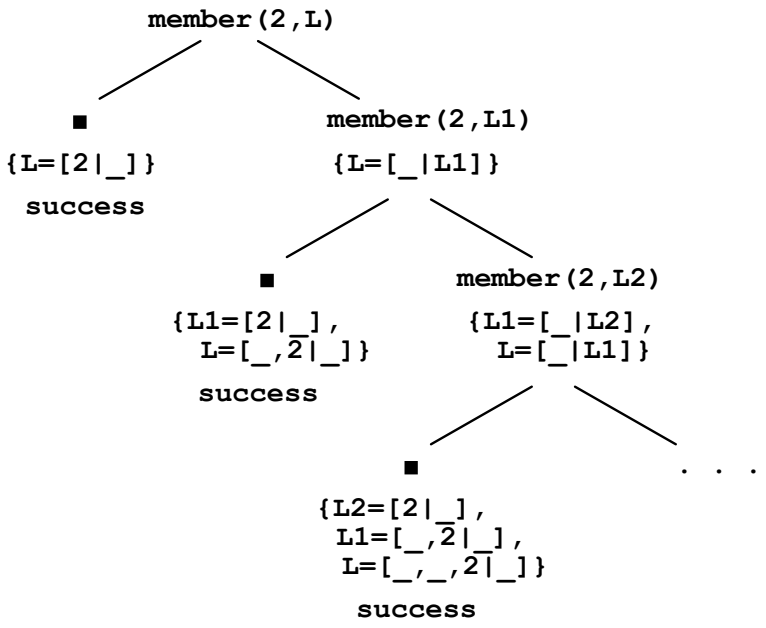


The query `member(2,L)` results in an infinite number of answers, in something like the following form:

```
Y = [2|_194]
More (y/n)? y

Y = [_190,2|_206]
More (y/n)? y

Y = [_190,_202,2|_218]
More (y/n)? y

Y = [_190,_202,_214,2|_230]
More (y/n)? ...
```

This results from the following infinite search tree:

© Kenneth C. Louden 2003 All rights reserved



12.29 The problem is that the goals on the right hand side are in the wrong order, since the goal `permutation(S,T)` instantiates `T`, and therefore `T` will be uninstantiated when the goal `sorted(T)` is executed, and this will only succeed in instantiating `T` to either `[]` or `[X]`, where `X` is uninstantiated. (See the code for `sorted` on page 567. The third clause always fails for an uninstantiated parameter, since the goal `X =< Y` fails if `X` and `Y` are both instantiated.) Thus, the goal `sort(S,T)` will only succeed if `S` is either empty or a singleton.

Chapter 13

No supplemental exercises currently available.

Chapter 14

No supplemental exercises currently available.

© Kenneth C. Louden 2003 All rights reserved