6 Control Flow

Having considered the mechanisms that a compiler uses to enforce semantic rules (Chapter 4) and the characteristics of the target machines for which compilers must generate code (Chapter 5), we now return to core issues in language design. Specifically, we turn in this chapter to the issue of *control flow* or *ordering* in program execution. Ordering is fundamental to most (though not all) models of computing. It determines what should be done first, what second, and so forth, to accomplish some desired task. We can organize the language mechanisms used to specify ordering into seven principal categories.

1. *sequencing:* Statements are to be executed (or expressions evaluated) in a certain specified order—usually the order in which they appear in the program text.
2. *selection:* Depending on some run-time condition, a *choice* is to be made among two or more statements or expressions. The most common selection constructs are `if` and `case` (`switch`) statements. Selection is also sometimes referred to as *alternation*.
3. *iteration:* A given fragment of code is to be executed repeatedly, either a certain number of times or until a certain run-time condition is true. Iteration constructs include `while`, `do`, and `repeat` loops.
4. *procedural abstraction:* A potentially complex collection of control constructs (a *subroutine*) is encapsulated in a way that allows it to be treated as a single unit, often subject to parameterization.
5. *recursion:* An expression is defined in terms of (simpler versions of) itself, either directly or indirectly; the computational model requires a stack on which to save information about partially evaluated instances of the expression. Recursion is usually defined by means of self-referential subroutines.
6. *concurrency:* Two or more program fragments are to be executed/evaluated "at the same time," either in parallel on separate processors or interleaved on a single processor in a way that achieves the same effect.

7. *nondeterminacy:* The ordering or choice among statements or expressions is deliberately left unspecified, implying that any alternative will lead to correct results. Some languages require the choice to be random, or fair, in some formal sense of the word.

Though the syntactic and semantic details vary from language to language, these seven principal categories cover all of the control-flow constructs and mechanisms found in most programming languages. A programmer who thinks in terms of these categories, rather than the syntax of some particular language, will find it easy to learn new languages, evaluate the tradeoffs among languages, and design and reason about algorithms in a language-independent way.

Subroutines are the subject of Chapter 8. Concurrency is the subject of Chapter 12. The bulk of this chapter (Sections 6.3 through 6.7) is devoted to a study of the five remaining categories. We begin in Section 6.1 by examining expression evaluation. We consider the syntactic form of expressions, the precedence and associativity of operators, the order of evaluation of operands, and the semantics of the assignment statement. We focus in particular on the distinction between variables that hold a *value* and variables that hold a *reference to* a value; this distinction will play an important role many times in future chapters. In Section 6.2 we consider the difference between *structured* and *unstructured* (`goto`-based) control flow.

The relative importance of different categories of control flow varies significantly among the different classes of programming languages. Sequencing, for example, is central to imperative (von Neumann and object-oriented) languages, but plays a relatively minor role in functional languages, which emphasize the evaluation of expressions, deemphasizing or eliminating statements (e.g., assignments) that affect program output in any way other than through the return of a value. Similarly, functional languages make heavy use of recursion, whereas imperative languages tend to emphasize iteration. Logic languages tend to deemphasize or hide the issue of control flow entirely: the programmer simply specifies a set of inference rules; the language implementation must find an order in which to apply those rules that will allow it to deduce values that satisfy some desired property.

6.1 Expression Evaluation

An expression generally consists of either a simple object (e.g., a literal constant, or a named variable or constant) or an *operator* or function applied to a collection of operands or arguments, each of which in turn is an expression. It is conventional to use the term *operator* for built-in functions that use special, simple syntax, and to use the term *operand* for the argument of an operator. In Algol-family languages, function calls consist of a function name followed by a parenthesized, comma-separated list of arguments, as in

EXAMPLE 6.1
A typical function call

```
my_func(A, B, C)
```

EXAMPLE 6.2 Typical operators

Algol-family operators are simpler: they typically take only one or two arguments, and dispense with the parentheses and commas:

```
a + b
- c
```

As we saw in Section 3.6.2, some languages define the operators as *syntactic sugar* for more "normal"-looking functions. In Ada, for example, `a + b` is short for `"+"(a, b)`; in C++, `a + b` is short for `a.operator+(b)`.

In general, a language may specify that function calls (operator invocations) employ prefix, infix, or postfix notation. These terms indicate, respectively, whether the function name appears before, among, or after its several arguments. Most imperative languages use infix notation for binary operators and prefix notation for unary operators and other functions (with parentheses around the arguments). Lisp uses prefix notation for all functions but places the function name *inside* the parentheses, in what is known as *Cambridge Polish*[1] notation:

EXAMPLE 6.3 Cambridge Polish (prefix) notation

```
(* (+ 1 3) 2)              ; that would be (1 + 3) * 2 in infix
(append a b c my_list)
```

A few languages, notably the R scripting language, allow the user to create new infix operators. Smalltalk uses infix notation for *all* functions (which it calls messages), both built-in and user-defined. The following Smalltalk statement sends a "`displayOn: at:`" message to graphical object `myBox`, with arguments `myScreen` and `100@50` (a pixel location). It corresponds to what other languages would call the invocation of the "`displayOn: at:`" function with arguments `myBox`, `myScreen`, and `100@50`.

EXAMPLE 6.4 Mixfix notation in Smalltalk

```
myBox displayOn: myScreen at: 100@50
```

EXAMPLE 6.5 Conditional expressions

This sort of multiword infix notation occurs occasionally in Algol-family languages as well.[2] In Algol one can say

```
a := if b <> 0 then a/b else 0;
```

Here "`if`... `then` ... `else`" is a three-operand infix operator. The equivalent operator in C is written "... `?` ... `:` ...":

```
a = b != 0 ? a/b : 0;
```

Postfix notation is used for most functions in Postscript, Forth, the input language of certain hand-held calculators, and the intermediate code of some com-

1 Prefix notation was popularized by Polish logicians of the early 20th century; Lisp-like parenthesized syntax was first employed (for noncomputational purposes) by philosopher W. V. Quine of Harvard University (Cambridge, MA).

2 Most authors use the term "infix" only for binary operators. Multiword operators may be called "mixfix" or left unnamed.

pilers. Postfix appears in a few places in other languages as well. Examples include the pointer dereferencing operator (`^`) of Pascal and the post-increment and -decrement operators (`++` and `--`) of C and its descendants.

6.1.1 Precedence and Associativity

EXAMPLE 6.6
A complicated Fortran expression

Most languages provide a rich set of built-in arithmetic and logical operators. When written in infix notation, without parentheses, these operators lead to ambiguity as to what is an operand of what. In Fortran, for example, which uses `**` for exponentiation, how should we parse `a + b * c**d**e/f`? Should this group as

```
((((a + b) * c)**d)**e)/f
```

or

```
a + (((b * c)**d)**(e/f))
```

or

```
a + ((b * (c**(d**e)))/f)
```

or yet some other option? (In Fortran, the answer is the last of the options shown.) ■

In any given language, the choice among alternative evaluation orders depends on the *precedence* and *associativity* of operators, concepts we introduced in Section 2.1.3. Issues of precedence and associativity do not arise in prefix or postfix notation.

EXAMPLE 6.7
Precedence in four influential languages

Precedence rules specify that certain operators, in the absence of parentheses, group "more tightly" than other operators. Associativity rules specify that sequences of operators of equal precedence group to the right or to the left. In most languages multiplication and division group more tightly than addition and subtraction. Other levels of precedence vary widely from one language to another. Figure 6.1 shows the levels of precedence for several well-known languages. ■

The precedence structure of C (and, with minor variations, of its descendants, C++, Java, and C#) is substantially richer than that of most other languages. It is, in fact, richer than shown in Figure 6.1, because several additional constructs, including type casts, function calls, array subscripting, and record field selection, are classified as operators in C. It is probably fair to say that most C programmers do not remember all of their language's precedence levels. The intent of the language designers was presumably to ensure that "the right thing" will usually happen when parentheses are not used to force a particular evaluation order. Rather than count on this, however, the wise programmer will consult the manual or add parentheses.

EXAMPLE 6.8
A "gotcha" in Pascal precedence

It is also probably fair to say that the relatively flat precedence hierarchy of Pascal is a mistake. In particular, novice Pascal programmers frequently write conditions like

Fortran	Pascal	C	Ada
		`++`, `--` (post-inc., dec.)	
`**`	`not`	`++`, `--` (pre-inc., dec.), `+`, `-` (unary), `&`, `*` (address, contents of), `!`, `~` (logical, bit-wise not)	`abs` (absolute value), `not`, `**`
`*`, `/`	`*`, `/`, `div`, `mod`, `and`	`*` (binary), `/`, `%` (modulo division)	`*`, `/`, `mod`, `rem`
`+`, `-` (unary and binary)	`+`, `-` (unary and binary), `or`	`+`, `-` (binary)	`+`, `-` (unary)
		`<<`, `>>` (left and right bit shift)	`+`, `-` (binary), `&` (concatenation)
`.eq.`, `.ne.`, `.lt.`, `.le.`, `.gt.`, `.ge.` (comparisons)	`<`, `<=`, `>`, `>=`, `=`, `<>`, `IN`	`<`, `<=`, `>`, `>=` (inequality tests)	`=`, `/=` , `<`, `<=`, `>`, `>=`
`.not.`		`==`, `!=` (equality tests)	
		`&` (bit-wise and)	
		`^` (bit-wise exclusive or)	
		`|` (bit-wise inclusive or)	
`.and.`		`&&` (logical and)	`and`, `or`, `xor` (logical operators)
`.or.`		`||` (logical or)	
`.eqv.`, `.neqv.` (logical comparisons)		`?:` (if…then…else)	
		`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `^=`, `|=` (assignment)	
		`,` (sequencing)	

Figure 6.1 Operator precedence levels in Fortran, Pascal, C, and Ada. The operators at the top of the figure group most tightly.

```
    if A < B and C < D then (* ouch *)
```

Unless `A`, `B`, `C`, and `D` are all of type Boolean, which is unlikely, this code will result in a static semantic error, since the rules of precedence cause it to group as `A < (B and C) < D`. (And even if all four operands are of type Boolean, the result is almost sure to be something other than what the programmer intended.) Most languages avoid this problem by giving arithmetic operators higher precedence than relational (comparison) operators, which in turn have higher precedence than the logical operators. Notable exceptions include APL and Smalltalk, in which all operators are of equal precedence; parentheses *must* be used to specify grouping. ■

EXAMPLE 6.9
Common rules for associativity

Associativity rules are somewhat more uniform across languages, but still display some variety. The basic arithmetic operators almost always associate left-to-right, so `9 - 3 - 2` is `4` and not `8`. In Fortran, as noted above, the exponentiation operator (`**`) follows standard mathematical convention and associates right-to-left, so `4**3**2` is `262144` and not `4096`. In Ada, exponentiation does not associate: one must write either `(4**3)**2` or `4**(3**2)`; the language syntax does not allow the unparenthesized form. In languages that allow assignments inside expressions (an option we will consider more in Section 6.1.2), assignment associates right-to-left. Thus in C, `a = b = a + c` assigns `a + c` into `b` and then assigns the same value into `a`. ■

Because the rules for precedence and associativity vary so much from one language to another, a programmer who works in several languages is wise to make liberal use of parentheses.

6.1.2 Assignments

In a purely functional language, expressions are the building blocks of programs, and computation consists entirely of expression evaluation. The effect of any individual expression on the overall computation is limited to the value that expression provides to its surrounding context. Complex computations employ recursion to generate a potentially unbounded number of values, expressions, and contexts.

In an imperative language, by contrast, computation typically consists of an ordered series of changes to the values of variables in memory. Assignments provide the principal means by which to make the changes. Each assignment takes a pair of arguments: a value and a reference to a variable into which the value should be placed.

In general, a programming language construct is said to have a *side effect* if it influences subsequent computation (and ultimately program output) in any way other than by returning a value for use in the surrounding context. Purely functional languages have no side effects. As a result, the value of an expression in such a language depends only on the referencing environment in which the expression is evaluated, *not* on the time at which the evaluation occurs. If an expression yields a certain value at one point in time, it is guaranteed to yield the same value at any point in time. In fancier terms, expressions in a purely functional language are said to be *referentially transparent*.

By contrast, imperative programming is sometimes described as "computing by means of side effects." While the evaluation of an assignment may sometimes yield a value, what we really care about is the fact that it changes the value of a variable, thereby affecting the result of any later computation in which the variable appears.

Many (though not all) imperative languages distinguish between *expressions*, which always produce a value, and may or may not have side effects, and *statements*, which are executed *solely* for their side effects, and return no useful value.

References and Values

On the surface, assignment appears to be a very straightforward operation. Below the surface, however, there are some subtle but important differences in the semantics of assignment in different imperative languages. These differences are often invisible, because they do not affect the behavior of simple programs. They have a major impact, however, on programs that use pointers, and will be explored in further detail in Section 7.7. We provide an introduction to the issues here.

EXAMPLE 6.10
L-values and r-values

Consider the following assignments in C:

```
d = a;
a = b + c;
```

In the first statement, the right-hand side of the assignment refers to the *value* of `a`, which we wish to place into `d`. In the second statement, the left-hand side refers to the *location* of `a`, where we want to put the sum of `b` and `c`. Both interpretations—value and location—are possible because a variable in C (and in Pascal, Ada, and many other languages) is a named container for a value. We sometimes say that languages like C use a *value model* of variables. Because of their use on the left-hand side of assignment statements, expressions that denote locations are referred to as *l-values*. Expressions that denote values (possibly the value stored in a location) are referred to as *r-values*. Under a value model of variables, a given expression can be either an l-value or an r-value, depending on the context in which it appears. ■

EXAMPLE 6.11
L-values in C

Of course, not all expressions can be l-values, because not all values have a location, and not all names are variables. In most languages it makes no sense to say `2 + 3 = a`, or even `a = 2 + 3`, if `a` is the name of a constant. By the same token, not all l-values are simple names; both l-values and r-values can be complicated expressions. In C one may write

```
(f(a)+3)->b[c] = 2;
```

In this expression `f(a)` returns a pointer to some element of an array of structures (records). The assignment places the value 2 into the `c`-th element of field `b` of the third structure after the one to which `f`'s return value points. ■

EXAMPLE 6.12
L-values in C++

In C++ it is even possible for a function to return a "reference" to a structure, rather than a pointer to it, allowing one to write

```
g(a).b[c] = 2;
```
■

We will consider references further in Section 8.3.1.

Several languages make the distinction between l-values and r-values more explicit by employing a *reference model* of variables. In Clu, for example, a variable

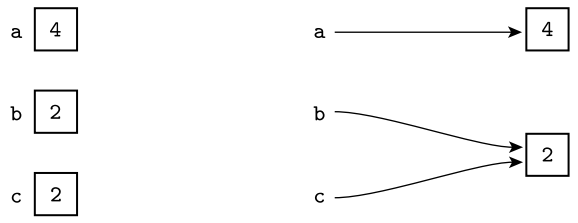


Figure 6.2 The value (left) and reference (right) models of variables. Under the reference model, it becomes important to distinguish between variables that refer to the same object and variables that refer to different objects whose values happen (at the moment) to be equal.

EXAMPLE 6.13
Variables as values and references

is not a named container for a value; rather, it is a named *reference to* a value. The following fragment of code is syntactically valid in both Pascal and Clu.

```
b := 2;
c := b;
a := b + c;
```

A Pascal programmer might describe this code by saying: "We put the value 2 in b and then copy it into c. We then read these values, add them together, and place the resulting 4 in a." The Clu programmer would say: "We let b refer to 2 and then let c refer to it also. We then pass these references to the + operator, and let a refer to the result, namely 4."

These two ways of thinking are illustrated in Figure 6.2. With a value model of variables, as in Pascal, any integer variable can contain the value 2. With a reference model of variables, as in Clu, there is (at least conceptually) only *one* 2—a sort of Platonic Ideal—to which any variable can refer. The practical effect is the same in this example, because integers are *immutable*: the value of 2 never changes, so we can't tell the difference between two copies of the number 2 and two references to "the" number 2. ■

In a language that uses the reference model, every variable is an l-value. When it appears in a context that expects an r-value, it must be *dereferenced* to obtain the value to which it refers. In most languages with a reference model (including Clu), the dereference is implicit and automatic. In ML, the programmer must

DESIGN & IMPLEMENTATION

Implementing the reference model

It is tempting to assume that the reference model of variables is inherently more expensive than the value model, since a naive implementation would require a level of indirection on every access. As we shall see in Section 7.7.1, however, most compilers for languages with a reference model use multiple copies of immutable objects for the sake of efficiency, achieving exactly the same performance for simple types that they would with a value model.

use an explicit dereference operator, denoted with a prefix exclamation point. We will revisit ML pointers in Section 7.7.1.

The difference between the value and reference models of variables becomes particularly important (specifically, it can affect program output and behavior) if the values to which variables refer can change "in place," as they do in many programs with linked data structures, or if it is possible for variables to refer to different objects that happen to have the "same" value. In this latter case it becomes important to distinguish between variables that refer to the same object and variables that refer to different objects whose values happen (at the moment) to be equal. (Lisp, as we shall see in Sections 7.10 and 10.3.3, provides more than one notion of equality, to accommodate this distinction.) We will discuss the value and reference models of variables further in Section 7.7. Languages that employ (some variant of) the reference model include Algol 68, Clu, Lisp/Scheme, ML, Haskell, and Smalltalk.

Java uses a value model for built-in types and a reference model for user-defined types (classes). C# and Eiffel allow the programmer to choose between the value and reference models for each individual user-defined type. A C# `class` is a reference type; a `struct` is a value type.

Boxing

EXAMPLE 6.14
Wrapper objects in Java 2

A drawback of using a value model for built-in types is that they can't be passed uniformly to methods that expect class typed parameters. Early versions of Java, for example, required the programmer to "wrap" objects of built-in types inside corresponding predefined class types in order to insert them in standard container (collection) classes:

```
import java.util.Hashtable;
...
Hashtable ht = new Hashtable();
...
Integer N = new Integer(13);        // Integer is a "wrapper" class
ht.put(N, new Integer(31));
Integer M = (Integer) ht.get(N);
int m = M.intValue();
```
■

EXAMPLE 6.15
Boxing in Java 5

More recent versions of Java perform automatic *boxing* and *unboxing* operations that avoid the need for wrappers in many cases:

```
ht.put(13, 31);
int m = (Integer) ht.get(13);
```

Here the compiler creates hidden `Integer` objects to hold the values `13` and `31`, so they may be passed to `put` as references. The `Integer` cast on the return value is still needed, to make sure that the hash table entry for `13` is really an integer and not, say, a floating-point number or string. ■

EXAMPLE 6.16
Boxing in C#

C# "boxes" not only the arguments, but the cast as well, eliminating the need for the `Integer` class entirely. C# also provides so-called *indexers* (Section 9.1, page 474), which can be used to overload the subscripting (`[]`) operator, giving the hash table array-like syntax:

```
ht[13] = 31;
int m = (int) ht[13];
```

Orthogonality

One of the principal design goals of Algol 68 was to make the various features of the language as *orthogonal* as possible. Orthogonality means that features can be used in any combination, the combinations all make sense, and the meaning of a given feature is *consistent*, regardless of the other features with which it is combined. The name is meant to draw an explicit analogy to orthogonal vectors in linear algebra: none of the vectors in an orthogonal set depends on (or can be expressed in terms of) the others, and all are needed in order to describe the vector space as a whole.

Algol 68 was one of the first languages to make orthogonality a principal design goal, and in fact few languages since have given the goal such weight. Among other things, Algol 68 is said to be *expression-oriented*: it has no separate notion of statement. Arbitrary expressions can appear in contexts that would call for a statement in a language like Pascal, and constructs that are considered to be statements in other languages can appear within expressions. The following, for example, is valid in Algol 68:

EXAMPLE 6.17
Expression orientation in Algol 68

```
begin
    a := if b < c then d else e;
    a := begin f(b); g(c) end;
    g(d);
    2 + 3
end
```

Here the value of the `if`... `then` ... `else` construct is either the value of its `then` part or the value of its `else` part, depending on the value of the condition. The value of the "statement list" on the right-hand side of the second assignment is the value of its final "statement," namely the return value of `g(c)`. There is no need to distinguish between procedures and functions, because every subroutine call returns a value. The value returned by `g(d)` is discarded in this example. Finally, the value of the code fragment as a whole is 5, the sum of 2 and 3.

C takes an approach intermediate between Pascal and Algol 68. It distinguishes between statements and expressions, but one of the classes of statement is an "expression statement," which computes the value of an expression and then throws it away. In effect, this allows an expression to appear in any context that would require a statement in most other languages. C also provides special expression forms for selection and sequencing. Algol 60 defines `if`... `then` ... `else` as both a statement and an expression.

Both Algol 68 and C allow assignments within expressions. The value of an assignment is simply the value of its right-hand side. Unfortunately, where most of the descendants of Algol 60 use the := token to represent assignment, C follows Fortran in simply using =. It uses == to represent a test for equality (Fortran uses .eq.). Moreover, C lacks a separate Boolean type. (C99 has a new _Bool type, but it's really just a one-bit integer.) In any context that would require a Boolean value in other languages, C accepts an integer (or anything that can be coerced to be an integer). It interprets zero as false; any other value is true. As a result, both of the following constructs are valid—common—in C.

EXAMPLE 6.18
A "gotcha" in C conditions

```
if (a == b) {
    /* do the following if a equals b */

if (a = b) {
    /* assign b into a and then do
       the following if the result is nonzero */
```

Programmers who are accustomed to Ada or some other language in which = is the equality test frequently write the second form above when the first is what is intended. This sort of bug can be very hard to find. ■

Though it provides a true Boolean type (bool), C++ shares the problem of C, because it provides automatic coercions from numeric, pointer, and enumeration types. Java and C# eliminate the problem by disallowing integers in Boolean contexts. The assignment operator is still =, and the equality test is still ==, but the statement if (a = b) ... will generate a compile-time type clash error unless a and b are both boolean (Java) or bool (C#), which is generally unlikely.

Combination Assignment Operators

EXAMPLE 6.19
Updating assignments

Because they rely so heavily on side effects, imperative programs must frequently *update* a variable. It is thus common in many languages to see statements like

```
a = a + 1;
```

or worse,

```
b.c[3].d = b.c[3].d * e;
```

Such statements are not only cumbersome to write and to read (we must examine both sides of the assignment carefully to see if they really are the same), they also result in redundant address calculations (or at least extra work to eliminate the redundancy in the code improvement phase of compilation). ■

EXAMPLE 6.20
Side effects and updates

If the address calculation has a side effect, then we may need to write a pair of statements instead. Consider the following code in C:

```
void update(int A[], int index_fn(int n)) {
    int i, j;
    /* calculate i */
    ...
    j = index_fn(i);
    A[j] = A[j] + 1;
}
```

Here we cannot safely write

```
A[index_fn(i)] = A[index_fn(i)] + 1;
```

We have to introduce the temporary variable `j` because we don't know whether `index_fn` has a side effect or not. If it is being used, for example, to keep a log of elements that have been updated, then we shall want to make sure that `update` calls it only once. ■

To eliminate the clutter and compile- or run-time cost of redundant address calculations, and to avoid the issue of repeated side effects, many languages, beginning with Algol 68 and including C and its descendants, provide so-called *assignment operators* to update a variable. Using assignment operators, the statements in Example 6.19 can be written as follows.

EXAMPLE 6.21
Assignment operators

```
a += 1;
b.c[3].d *= e;
```

Similarly, the two assignments in the `update` function can be replaced with

```
A[index_fn(i)] += 1;
```

In addition to being aesthetically cleaner, the assignment operator form guarantees that the address calculation is performed only once. ■

As shown in Figure 6.1, C provides 10 different assignment operators, one for each of its binary arithmetic and bit-wise operators. C also provides prefix and postfix increment and decrement operations. These allow even simpler code in `update`:

EXAMPLE 6.22
Prefix and postfix inc/dec

```
A[index_fn(i)]++;
```

or

```
++A[index_fn(i)];
```

More significantly, increment and decrement operators provide elegant syntax for code that uses an index or a pointer to traverse an array:

```
A[--i] = b;
*p++ = *q++;
```

When prefixed to an expression, the `++` or `--` operator increments or decrements its operand *before* providing a value to the surrounding context. In the postfix form, `++` or `--` updates its operand *after* providing a value. If `i` is 3 and `p` and `q` point to the initial elements of a pair of arrays, then `b` will be assigned into `A[2]` (not `A[3]`), and the second assignment will copy the initial elements of the arrays (not the second elements). ■

EXAMPLE 6.23
Advantages of postfix inc/dec

The prefix forms of `++` and `--` are syntactic sugar for `+=` and `-=`. We could have written

```
A[i -= 1] = b;
```

above. The postfix forms are not syntactic sugar. To obtain an effect similar to the second statement above we would need an auxiliary variable and a lot of extra notation:

```
*(t = p, p += 1, t) = *(t = q, q += 1, t);
```

■

Both the assignment operators (`+=`, `-=`) and the increment and decrement operators (`++`, `--`) do "the right thing" when applied to pointers in C. If `p` points to an object that occupies *n* bytes in memory (including any bytes required for alignment, as discussed in Section 5.1), then `p += 3` points 3*n* bytes higher in memory.

Multiway Assignment

EXAMPLE 6.24
Simple multiway assignment

We have already seen that the right associativity of assignment (in languages that allow assignment in expressions) allows one to write things like `a = b = c`. In several languages, including Clu, ML, Perl, Python, and Ruby, it is also possible to write

```
a, b := c, d;
```

Here the comma in the right-hand side is *not* the sequencing operator of C. Rather, it serves to define an expression, or *tuple*, consisting of multiple r-values. The comma operator on the left-hand side produces a tuple of l-values. The effect of the assignment is to copy `c` into `a` and `d` into `b`.[3] ■

EXAMPLE 6.25
Advantages of multiway assignment

While we could just as easily have written

```
a := c; b := d;
```

the multiway (tuple) assignment allows us to write things like

```
a, b := b, a;
```

which would otherwise require auxiliary variables. Moreover, multiway assignment allows functions to return tuples, as well as single values:

```
a, b, c := foo(d, e, f);
```

This notation eliminates the asymmetry (nonorthogonality) of functions in most programming languages, which allow an arbitrary number of arguments but only a single return. ■

ML generalizes the idea of multiway assignment into a powerful *pattern-matching* mechanism; we will examine this mechanism in more detail in Section 7.2.4.

✓ CHECK YOUR UNDERSTANDING

1. Name seven major categories of control-flow mechanisms.
2. What distinguishes *operators* from other sorts of functions?

3 The syntax shown here is for Clu. Perl, Python, and Ruby follow C in using = for assignment. ML requires parentheses around each tuple.

3. Explain the difference between *prefix*, *infix*, and *postfix* notation. What is *Cambridge Polish* notation? Name two programming languages that use postfix notation.
4. Why don't issues of associativity and precedence arise in Postscript or Forth?
5. What does it mean for an expression to be *referentially transparent*?
6. What is the difference between a *value* model of variables and a *reference* model of variables? Why is the distinction important?
7. What is an *l-value*? An *r-value*?
8. Why is the distinction between *mutable* and *immutable* values important in the implementation of a language with a reference model of variables?
9. Define *orthogonality* in the context of programming language design.
10. What does it mean for a language to be *expression-oriented*?
11. What are the advantages of updating a variable with an *assignment operator*, rather than with a regular assignment in which the variable appears on both the left- and right-hand sides?

6.1.3 **Initialization**

Because they already provide a construct (the assignment statement) to set the value of a variable, imperative languages do not always provide a means of specifying an initial value for a variable in its declaration. There are at least two reasons, however, why such initial values may be useful:

1. In the case of statically allocated variables (as discussed in Section 3.2), an initial value that is specified in the context of the declaration can be placed into memory by the compiler. If the initial value is set by an assignment statement instead, it will generally incur execution cost at run time.
2. One of the most common programming errors is to use a variable in an expression before giving it a value. One of the easiest ways to prevent such errors (or at least ensure that erroneous behavior is repeatable) is to give every variable a value when it is first declared.

Some languages (e.g., Pascal) have no initialization facility at all; all variables must be given values by explicit assignment statements. To avoid the expense of run-time initialization of statically allocated variables, many Pascal implementations provide initialization as a language extension, generally in the form of a `:=` *expr* immediately after the name in the declaration. Unfortunately, the extension is usually nonorthogonal, in the sense that it only works for variables of simple, built-in types. A more complete and orthogonal approach to initialization requires a notation for *aggregates*: built-up structured values of user-defined composite types. Aggregates can be found in several languages, including C, Ada, Fortran 90, and ML; we will discuss them further in Section 7.1.5. It should be emphasized that initialization saves time only for variables that are statically allocated. Variables allocated in the stack or heap at run time must be initialized at run time.[4] It is also worth noting that the problem of using an uninitialized variable occurs not only after elaboration, but also as a result of any operation that destroys a variable's value without providing a new one. Two of the most common such operations are explicit deallocation of an object referenced through a pointer and modification of the *tag* of a variant record. We will consider these operations further in Sections 7.7 and 7.3.4, respectively.

If a variable is not given an initial value explicitly in its declaration, the language may specify a default value. In C, for example, statically allocated variables for which the programmer does not provide an initial value are guaranteed to be represented in memory as if they had been initialized to zero. For most types on most machines, this is a string of zero bits, allowing the language implementation to exploit the fact that most operating systems (for security reasons) fill newly allocated memory with zeros. Zero-initialization applies recursively to the subcomponents of variables of user-defined composite types. The designers of C chose not to incur the run-time cost of automatically zero-filling uninitialized variables that are allocated in the stack or heap. The programmer can specify an initial value if desired; the effect is the same as if an assignment had been placed at the beginning of the code for the variable's scope.

Constructors

Many object-oriented languages allow the programmer to define types for which initialization of dynamically allocated variables occurs automatically, even when no initial value is specified in the declaration. C++ also distinguishes carefully between initialization and assignment. Initialization is interpreted as a call to a *constructor* function for the variable's type, with the initial value as an argument. In the absence of coercion, assignment is interpreted as a call to the type's assignment operator or, if none has been defined, as a simple bit-wise copy of the value on the assignment's right-hand side. The distinction between initialization and assignment is particularly important for user-defined abstract data types that perform their own storage management. A typical example occurs in variable-length character strings. An assignment to such a string must generally deallocate the space consumed by the old value of the string before allocating space for the new value. An initialization of the string must simply allocate space. Initialization with a nontrivial value is generally cheaper than default initialization followed by assignment because it avoids deallocation of the space allocated for the default value. We will return to this issue in Section 9.3.2.

4 For variables that are accessed indirectly (e.g., in languages that employ a reference model of variables), a compiler can often reduce the cost of initializing a stack or heap variable by placing the initial value in static memory, and only creating the pointer to it at elaboration time.

Neither Java nor C# distinguishes between initialization and assignment, or between declaration and definition. Java uses a reference model for all variables of user-defined object types, and provides for automatic storage reclamation, so assignment never copies values. C# allows the programmer to specify a value model when desired (in which case assignment does copy values), but otherwise it mirrors Java. We will return to these issues again in Chapter 9 when we consider object-oriented features in more detail.

Definite Assignment

Java and C# require that a value be "definitely assigned" to a variable before that variable is used in any expression. Both languages provide a precise definition of "definitely assigned," based on the control flow of the program. Roughly speaking, every possible control path to an expression must assign a value to every variable in that expression. This is a conservative rule; it can sometimes prohibit programs that would never actually use an uninitialized variable. In Java:

EXAMPLE 6.26
Programs outlawed by definite assignment

```
int i;
final static int j = 3;
...
if (j > 0) {
    i = 2;
}
...
if (j > 0) {
    System.out.println(i);
    // error: "i might not have been initialized"
}
```

DESIGN & IMPLEMENTATION

Safety v. performance

A recurring theme in any comparison between C++ and Java is the latter's willingness to accept additional run-time cost in order to obtain cleaner semantics or increased reliability. Definite assignment is one example: it may force the programmer to perform "unnecessary" initializations on certain code paths, but in so doing it avoids the many subtle errors that can arise from missing initialization in other languages. Similarly, the Java specification mandates automatic garbage collection, and its reference model of user-defined types forces most objects to be allocated in the heap. As we shall see in Chapters 7 and 9, Java also requires both dynamic binding of all method invocations and run-time checks for out-of-bounds array references, type clashes, and other dynamic semantic errors. Clever compilers can reduce or eliminate the cost of these requirements in certain common cases, but for the most part the Java design reflects an evolutionary shift away from performance as *the* overriding design goal.

While a human being might reason that `i` will only be used when it has previously been given a value, it is uncomputable to make such determinations in the general case, and the compiler does not attempt it. ■

Dynamic Checks

Instead of giving every uninitialized variable a default value, a language or implementation can choose to define the use of an uninitialized variable as a dynamic semantic error, and can catch these errors at run time. The advantage of the semantic checks is that they will often identify a program bug that is masked or made more subtle by the presence of a default value. With appropriate hardware support, uninitialized variable checks can even be as cheap as default values, at least for certain types. In particular, a compiler that relies on the IEEE standard for floating-point arithmetic can fill uninitialized floating-point numbers with a *signaling NaN* value, as discussed in Section © 5.2.1. Any attempt to use such a value in a computation will result in a hardware interrupt, which the language implementation may catch (with a little help from the operating system), and use to trigger a semantic error message.

For most types on most machines, unfortunately, the costs of catching all uses of an uninitialized variable at run time are considerably higher. If every possible bit pattern of the variable's representation in memory designates some legitimate value (and this is often the case), then extra space must be allocated somewhere to hold an initialized/uninitialized flag. This flag must be set to "uninitialized" at elaboration time and to "initialized" at assignment time. It must also be checked (by extra code) at every use—or at least at every use that the code improver is unable to prove is redundant. Dynamic semantic checks for uninitialized variables are common in interpreted languages, which already incur significant overhead on every variable access. Because of their cost, however, the checks are usually not performed in languages that are compiled.

6.1.4 Ordering Within Expressions

EXAMPLE 6.27
Indeterminate ordering

While precedence and associativity rules define the order in which binary infix operators are applied within an expression, they do not specify the order in which the operands of a given operator are evaluated. For example, in the expression

```
a - f(b) - c * d
```

we know from associativity that `f(b)` will be subtracted from `a` before performing the second subtraction, and we know from precedence that the right operand of that second subtraction will be the result of `c * d`, rather than merely `c`, but without additional information we do not know whether `a - f(b)` will be evaluated before or after `c * d`. Similarly, in a subroutine call with multiple arguments

```
f(a, g(b), c)
```

we do not know the order in which the arguments will be evaluated. ■

There are two main reasons why the order can be important:

EXAMPLE 6.28
A value that depends on ordering

1. *Side effects:* If `f(b)` may modify `d`, then the value of `a - f(b) - c * d` will depend on whether the first subtraction or the multiplication is performed first. Similarly, if `g(b)` may modify `a` and/or `c`, then the values passed to `f(a, g(b), c)` will depend on the order in which the arguments are evaluated. ■

EXAMPLE 6.29
An optimization that depends on ordering

2. *Code improvement:* The order of evaluation of subexpressions has an impact on both register allocation and instruction scheduling. In the expression `a * b + f(c)`, it is probably desirable to call `f` before evaluating `a * b`, because the product, if calculated first, would need to be saved during the call to `f`, and `f` might want to use all the registers in which it might easily be saved. In a similar vein, consider the sequence

```
a := B[i];
c := a * 2 + d * 3;
```

   Here it is probably desirable to evaluate `d * 3` before evaluating `a * 2`, because the previous statement, `a := B[i]`, will need to load a value from memory. Because loads are slow, if the processor attempts to use the value of `a` in the next instruction (or even the next few instructions on many machines), it will have to wait. If it does something unrelated instead (i.e., evaluate `d * 3`), then the load can proceed in parallel with other computation. ■

Because of the importance of code improvement, most language manuals say that the order of evaluation of operands and arguments is undefined. (Java and C# are unusual in this regard: they require left-to-right evaluation.) In the absence of an enforced order, the compiler can choose whatever order results in faster code.

Applying Mathematical Identities

Some language implementations (e.g., for dialects of Fortran) allow the compiler to *rearrange* expressions involving operators whose mathematical abstractions are commutative, associative, and/or distributive, in order to generate faster code. Consider the following Fortran fragment.

EXAMPLE 6.30
Optimization and mathematical "laws"

```
a = b + c
d = c + e + b
```

Some compilers will rearrange this as

```
a = b + c
d = b + c + e
```

They can then recognize the *common subexpression* in the first and second statements, and generate code equivalent to

```
a = b + c
d = a + e
```

Similarly,

```
a = b/c/d
e = f/d/c
```

may be rearranged as

```
t = c * d
a = b/t
e = f/t
```

Unfortunately, while mathematical arithmetic obeys a variety of commutative, associative, and distributive laws, computer arithmetic is not as orderly. The problem is that numbers in a computer are of limited precision. With 32-bit arithmetic, the expression `b - c + d` can be evaluated safely left to right if `a`, `b`, and `c` are all integers between two billion and three billion (2^{32} is a little less than 4.3 billion). If the compiler attempts to reorganize this expression as `b + d - c`, however (e.g., in order to delay its use of `c`), then arithmetic overflow will occur.

Many languages, including Pascal and most of its descendants, provide dynamic semantic checks to detect arithmetic overflow. In some implementations these checks can be disabled to eliminate their run-time overhead. In C and C++, the effect of arithmetic overflow is implementation-dependent. In Java, it is well defined: the language definition specifies the size of all numeric types, and requires two's complement integer and IEEE floating-point arithmetic. In C#, the programmer can explicitly request the presence or absence of checks by tagging an expression or statement with the `checked` or `unchecked` keyword. In a completely different vein, Scheme, Common Lisp, and several scripting languages place no a priori limit on the size of numbers; space is allocated to hold extra-large values on demand.

Even in the absence of overflow, the limited precision of floating-point arithmetic can cause different arrangements of the "same" expression to produce sig-

DESIGN & IMPLEMENTATION

Evaluation order

Expression evaluation represents a difficult tradeoff between semantics and implementation. To limit surprises, most language definitions require the compiler, if it ever reorders expressions, to respect any ordering imposed by parentheses. The programmer can therefore use parentheses to prevent the application of arithmetic "identities" when desired. No similar guarantee exists with respect to the order of evaluation of operands and arguments. It is therefore unwise to write expressions in which a side effect of evaluating one operand or argument can affect the value of another. As we shall see in Section 6.3, some languages, notably Euclid and Turing, outlaw such side effects.

EXAMPLE 6.31
Reordering and numerical stability

nificantly different results, invisibly. Single-precision IEEE floating-point numbers devote 1 bit to the sign, 8 bits to the exponent (power of 2), and 23 bits to the mantissa. Under this representation, `a + b` is guaranteed to result in a loss of information if $|\log_2(a/b)| > 23$. Thus if `b = -c`, then `a + b + c` may appear to be zero, instead of `a`, if the magnitude of `a` is small, while the magnitude of `b` and `c` is large. In a similar vein, a number like 0.1 cannot be represented precisely, because its binary representation is a "repeating decimal": 0.0001001001.... For certain values of `x`, `(0.1 + x) * 10.0` and `1.0 + (x * 10.0)` can differ by as much as 25%, even when `0.1` and `x` are of the same magnitude. ■

6.1.5 Short-Circuit Evaluation

EXAMPLE 6.32
Short-circuited expressions

Boolean expressions provide a special and important opportunity for code improvement and increased readability. Consider the expression `(a < b) and (b < c)`. If `a` is greater than `b`, there is really no point in checking to see whether `b` is less than `c`; we know the overall expression must be false. Similarly, in the expression `(a > b) or (b > c)`, if `a` is indeed greater than `b` there is no point in checking to see whether `b` is greater than `c`; we know the overall expression must be true. A compiler that performs *short-circuit evaluation* of Boolean expressions will generate code that skips the second half of both of these computations when the overall value can be determined from the first half. ■

EXAMPLE 6.33
Saving time with short-circuiting

Short-circuit evaluation can save significant amounts of time in certain situations:

```
if (very_unlikely_condition && very_expensive_function()) ...
```
■

EXAMPLE 6.34
Short-circuit pointer chasing

But time is not the only consideration, or even the most important one. Short-circuiting changes the *semantics* of Boolean expressions. In C, for example, one can use the following code to search for an element in a list.

```
p = my_list;
while (p && p->key != val)
    p = p->next;
```

C short-circuits its `&&` and `||` operators, and uses zero for both nil and false, so `p->key` will be accessed if and only if `p` is non-nil. The syntactically similar code in Pascal does not work, because Pascal does not short-circuit `and` and `or`:

```
p := my_list;
while (p <> nil) and (p^.key <> val) do     (* ouch! *)
    p := p^.next;
```

Here both of the `<>` relations will be evaluated before `and`-ing their results together. At the end of an unsuccessful search, `p` will be `nil`, and the attempt to access `p^.key` will be a run-time (dynamic semantic) error, which the compiler may or may not have generated code to catch. To avoid this situation, the Pascal programmer must introduce an auxiliary Boolean variable and an extra level of nesting:

```
 1. function tally(word : string) : integer;
 2.     (* Look up word in hash table.  If found, increment tally; If not
 3.         found, enter with a tally of 1.  In either case, return tally. *)
    ...
 4. function misspelled(word : string) : Boolean;
 5.     (* Check to see if word is mis-spelled and return appropriate
 6.         indication.  If yes, increment global count of mis-spellings. *)
    ...
 7. while not eof(doc_file) do begin
 8.     w := get_word(doc_file);
 9.     if (tally(w) = 10) and misspelled(w) then
10.         writeln(w)
11. end;
12. writeln(total_misspellings);
```

Figure 6.3 Pascal code that counts on the evaluation of Boolean operands.

```
p := my_list;
still_searching := true;
while still_searching do
    if p = nil then
        still_searching := false
    else if p^.key = val then
        still_searching := false
    else
        p := p^.next;
```

EXAMPLE 6.35 Short-circuiting and other errors

Short-circuit evaluation can also be used to avoid out-of-bound subscripts:

```
const MAX = 10;
int A[MAX];                     /* indices from 0 to 9 */
...
if (i >= 0 && i < MAX && A[i] > foo) ...
```

division by zero:

```
if (d <> 0 && n/d > threshold) ...
```

and various other errors.

EXAMPLE 6.36 When not to use short-circuiting

Short-circuiting is not necessarily as attractive for situations in which a Boolean subexpression can cause a side effect. Suppose we wish to count occurrences of words in a document, and print a list of all misspelled words that appear ten or more times, together with a count of the total number of misspellings. Pascal code for this task appears in Figure 6.3. Here the `if` statement at line 9 tests the conjunction of two subexpressions, both of which have important side effects. If short-circuit evaluation is used, the program will not compute the right result. The code can be rewritten to eliminate the need for non-short-circuit evaluation, but one might argue that the result is more awkward than the version shown.

So now we have seen situations in which short-circuiting is highly desirable, and others in which at least some programmers would find it undesirable. A few languages, among them Clu, Ada, and C, provide both regular *and* short-circuit Boolean operators. (Similar flexibility can be achieved with `if`...`then`...`else` in an expression-oriented language such as Algol 68; see Exercise 6.10.)

EXAMPLE 6.37
Optional short-circuiting

In Clu, the regular Boolean operators are `and` and `or`; the short-circuit operators are `cand` and `cor` (for *conditional* `and` and `or`):

```
if d ~= 0 cand n/d > threshold then ...
```

In Ada, the regular operators are also `and` and `or`; the short-circuit operators are the two-word operators `and then` and `or else`:

```
found_it := p /= null and then p.key = val;
```

(Clu and Ada use `~=` and `/=`, respectively, for "not equal.") C's logical `&&` and `||` operators short-circuit; the bit-wise `&` and `|` operators can be used as non-short-circuiting alternatives when their arguments are logical (zero or one) values. ■

When used to determine the flow of control in a selection or iteration construct, short-circuit Boolean expressions do not really have to calculate a Boolean value; they simply have to ensure that control takes the proper path in any given situation. We will look more closely at the generation of code for short-circuit expressions in Section 6.4.1.

✓ **CHECK YOUR UNDERSTANDING**

12. Given the ability to assign a value into a variable, why is it useful to be able to specify an *initial* value?
13. What are *aggregates*? Why are they useful?
14. Explain the notion of *definite assignment* in Java and C#.
15. Why is it generally expensive to catch all uses of uninitialized variables at run time?
16. Why is it impossible to catch all uses of uninitialized variables at compile time?
17. Why do most languages leave unspecified the order in which the arguments of an operator or function are evaluated?
18. What is *short-circuit* Boolean evaluation? Why is it useful?

6.2 Structured and Unstructured Flow

EXAMPLE 6.38
Control flow with `goto`s in Fortran

Control flow in assembly languages is achieved by means of conditional and unconditional jumps (branches). Early versions of Fortran mimicked the low-level approach by relying heavily on goto statements for most nonprocedural control flow:

```
    if A .lt. B goto 10          ! ".lt." means "<"
    ...
10
```

The 10 on the bottom line is a *statement label*.

Goto statements also feature prominently in other early imperative languages. In Cobol and PL/I they provide the only means of writing logically controlled (while-style) loops. Algol 60 and its successors provide a wealth of non-goto-based constructs, but until recently most Algol-family languages still provided goto as an option.

Throughout the late 1960s and much of the 1970s, language designers debated hotly the merits and evils of gotos. It seems fair to say the detractors won. Ada and C# allow gotos only in limited contexts. Modula (1, 2, and 3), Clu, Eiffel, and Java do not allow them at all. Fortran 90 and C++ allow them primarily for compatibility with their predecessor languages. (Java reserves the token goto as a keyword, to make it easier for a Java compiler to produce good error messages when a programmer uses a C++ goto by mistake.)

The abandonment of gotos was part of a larger "revolution" in software engineering known as *structured programming*. Structured programming was the "hot trend" of the 1970s, in much the same way that object-oriented programming was the trend of the 1990s. Structured programming emphasizes top-down design (i.e., progressive refinement), modularization of code, structured types (records, sets, pointers, multidimensional arrays), descriptive variable and constant names, and extensive commenting conventions. The developers of structured programming were able to demonstrate that within a subroutine, almost any well-designed imperative algorithm can be elegantly expressed with only sequencing, selection, and iteration. Instead of labels, structured languages rely on the boundaries of lexically nested constructs as the targets of branching control.

Many of the structured control-flow constructs familiar to modern programmers were pioneered by Algol 60. These include the if...then...else construct and both enumeration (for) and logically (while) controlled loops. The case statement was introduced by Wirth and Hoare in Algol W [WH66] as an alternative to the more unstructured computed goto and switch constructs of Fortran and Algol 60, respectively. Case statements were adopted in limited form by Algol 68, and more completely by Pascal, Modula, C, Ada, and a host of modern languages.

6.2.1 Structured Alternatives to goto

Once the principal structured constructs had been defined, most of the controversy surrounding gotos revolved around a small number of special cases, each of which was eventually addressed in structured ways.

EXAMPLE 6.39
Leaving the middle of a loop

Mid-loop exit and continue: A common use of gotos in Pascal was to break out of the middle of a loop:

```
    while not eof do begin
        readln(line);
        if all_blanks(line) then goto 100;
        consume_line(line)
    end;
100:
```

Less commonly, one would also see a label *inside* the end of a loop, to serve as the target of a goto that would terminate a given iteration early. As we shall see in Section 6.5.5, mid-loop exits are supported by special "one-and-a half" loop constructs in languages like Modula, C, and Ada. Some languages also provide a statement to skip the remainder of the current loop iteration: continue in C; cycle in Fortran 90; next in Perl.

EXAMPLE 6.40
Returning from the middle of a subroutine

Early returns from subroutines: Gotos were used fairly often in Pascal to terminate the current subroutine:

```
    procedure consume_line(var line: string);
    ...
    begin
        ...
            if line[i] = '%' then goto 100;
                (* rest of line is a comment *)
        ...
100:
    end;
```

At a minimum, this goto statement avoids putting the remainder of the procedure in an else clause. If the terminating condition is discovered within a deeply nested if... then ... else, it may avoid introducing an auxiliary variable that must be tested repeatedly in the remainder of the procedure (if not comment_line then ...).

The obvious alternative to this use of goto is an explicit return statement. Algol 60 does not have one, and neither does Pascal, but Fortran always has, and most modern Algol descendants have adopted it.

EXAMPLE 6.41
Escaping a nested subroutine

Multilevel returns: Returns and (local) gotos allow control to return from the current subroutine. On occasion it may make sense to return from a *surrounding* routine. Imagine, for example, that we are searching for an item matching some desired pattern with a collection of files. The search routine might invoke several nested routines, or a single routine multiple times, once for each place in which to search. In such a situation certain historic languages, including Algol 60, PL/I, and Pascal, permit a goto to branch to a lexically visible label *outside* the current subroutine:

```
    function search(key : string) : string;
    var rtn : string;
    ...
        procedure search_file(fname : string);
        ...
        begin
            ...
            for ... (* iterate over lines *)
                ...
                if found(key, line) then begin
                    rtn := line;
                    goto 100;
                end;
                ...
        end;
    ...


    begin (* search *)
        ...
        for ... (* iterate over files *)
            ...
            search_file(fname);
            ...
100:    return rtn;
    end;
```

In the event of a nonlocal goto, the language implementation must guarantee to repair the run-time stack of subroutine call information. This repair operation is known as *unwinding*. It requires not only that the implementation deallocate the stack frames of any subroutines from which we have escaped, but also that it perform any bookkeeping operations, such as restoration of register contents, that would have been performed when returning from those routines.

As a more structured alternative to the nonlocal goto, Common Lisp provides a `return-from` statement that names the lexically surrounding function or block from which to return, and also supplies a return value (eliminating the need for the artificial `rtn` variable in Example 6.41).

But what if `search_file` were not nested inside of `search`? We might, for example, wish to call it from routines that search files in different orders. In this case the `goto` of Pascal does not suffice. Algol 60 and PL/I allow labels to be passed as parameters, so a dynamically nested subroutine can perform a `goto` to a caller-defined location. PL/I also allows labels to be stored in variables. If a nested routine needs to return a value it can assign it to some variable in a scope that surrounds all calls. Alternatively, we can pass a reference parameter into every call, into which the result should be written.

EXAMPLE 6.42
Structured nonlocal transfers

Common Lisp again provides a more structured alternative, also available in Ruby. In either language an expression can be surrounded with a `catch` block, whose value can be provided by any dynamically nested routine that executes a matching `throw`. In Ruby we might write

```
def searchFile(fname, pattern)
    file = File.open(fname)
    file.each {|line|
        throw :found, line if line =~ /#{pattern}/
    }
end

match = catch :found do
    searchFile("f1", key)
    searchFile("f2", key)
    searchFile("f3", key)
    "not found\n"           # default value for catch,
end                         # if control gets this far
print match
```

Here the `throw` expression specifies a *tag*, which must appear in a matching `catch`, together with a value (`line`) to be returned as the value of the `catch`. (The `if` clause attached to the `throw` performs a regular-expression pattern match, looking for `pattern` within `line`. We will consider pattern matching in more detail in Section 13.4.2.) ■

Errors and other exceptions: The notion of a multilevel return assumes that the callee knows what the caller expects, and can return an appropriate value. In a related and arguably more common situation, a deeply nested block or subroutine may discover that it is unable to proceed with its usual function and, moreover, lacks the contextual information it would need to recover in any graceful way. The only recourse in such a situation is to "back out" of the nested context to some point in the program that is able to recover. Conditions that require a program to "back out" are usually called *exceptions*. We saw an example in Section 2.3.4, where we considered phrase-level recovery from syntax errors in a recursive-descent parser.

EXAMPLE 6.43
Error-checking with status codes

The most straightforward but generally least satisfactory way to cope with exceptions is to use auxiliary Boolean variables within a subroutine (`if still_ok then ...`) and to return status codes from calls:

```
status := my_proc(args);
if status = ok then ...
```
■

The auxiliary Booleans can be eliminated by using a nonlocal `goto` or multilevel return, but the caller to which we return must still inspect status codes explicitly. As a structured alternative, many modern languages provide an *exception handling* mechanism for convenient, nonlocal recovery from exceptions. We will discuss exception handling in more detail in Section 8.5. Typically the programmer appends a block of code called a *handler* to any computation in which an exception may arise. The job of the handler is to take whatever remedial action is required to recover from the exception. If the protected computation completes in the normal fashion, execution of the handler is skipped.

Multilevel returns and structured exceptions have strong similarities. Both involve a control transfer from some inner, nested context back to an outer context, unwinding the stack on the way. The distinction lies in where the computing occurs. In a multilevel return the inner context has all the information it needs. It completes its computation, generating a return value if appropriate, and transfers to the outer context in a way that requires no post-processing. At an exception, by contrast, the inner context cannot complete its work. It performs an "abnormal" return, triggering execution of the handler.

Common Lisp and Ruby provide mechanisms for both multilevel returns and exceptions, but this dual support is relatively rare. Most languages support only exceptions; programmers implement multilevel returns by writing a trivial handler. In an unfortunate overloading of terminology, the names `catch` and `throw`, which Common Lisp and Ruby use for multilevel returns, are used for exceptions in several other languages.

6.2.2 Continuations

The notion of nonlocal `goto`s that unwind the stack can be generalized by defining what are known as *continuations*. In low-level terms, a continuation consists of a code address and a referencing environment to be restored when jumping to that address. In higher-level terms, a continuation is an abstraction that captures a *context* in which execution might continue. Continuations are fundamental to denotational semantics. They also appear as first-class values in certain languages (notably Scheme and Ruby), allowing the programmer to define new control-flow constructs.

Continuation support in Scheme takes the form of a general purpose function called `call-with-current-continuation`, sometimes abbreviated `call/cc`.

DESIGN & IMPLEMENTATION

Cleaning up continuations

The implementation of continuations in Scheme and Ruby is surprisingly straightforward. Because local variables have unlimited extent in both languages, activation records must in general be allocated on the heap. As a result, explicit deallocation is neither required nor appropriate when jumping through a continuation; frames that are no longer accessible will eventually be reclaimed by a general purpose *garbage collector* (to be discussed in Section 7.7.3). Restoration of state (e.g., saved registers) from escaped routines is not required either: the continuation closure holds everything required to resume the captured context.

This function takes a single argument, f, which is itself a function. It calls f, passing as argument a continuation c that captures the current program counter and referencing environment. The continuation is represented by a closure, indistinguishable from the closures used to represent subroutines passed as parameters. At any point in the future, f can call c to reestablish the captured context. If nested calls have been made, control pops out of them, as it does with exceptions. More generally, however, c can be saved in variables, returned explicitly by subroutines, or called repeatedly, even after control has returned from f (recall that closures in Scheme have unlimited extent; see Section 3.5). `Call/cc` suffices to build a wide variety of control abstractions, including `goto`s, mid-loop `exit`s, multilevel returns, exceptions, iterators (Section 6.5.3), call-by-name parameters (Section 8.3.1), and coroutines (Section 8.6). It even subsumes the notion of returning from a subroutine, though it seldom replaces it in practice.

First-class continuations are an extremely powerful facility. They can be very useful if applied in well-structured ways (i.e., to define new control-flow constructs). Unfortunately, they also allow the undisciplined programmer to construct completely inscrutable programs.

6.3 Sequencing

Like assignment, sequencing is central to imperative programming. It is the principal means of controlling the order in which side effects (e.g., assignments) occur: when one statement follows another in the program text, the first statement executes before the second. In most imperative languages, lists of statements can be enclosed with `begin... end` or `{...}` delimiters and then used in any context in which a single statement is expected. Such a delimited list is usually called a *compound statement*. A compound statement preceded by a set of declarations is sometimes called a *block*.

In languages like Algol 68 and C, which blur or eliminate the distinction between statements and expressions, the value of a statement (expression) list is the value of its final element. In Common Lisp, the programmer can choose to return the value of the first element, the second, or the last. Of course, sequencing is a useless operation unless the subexpressions that do not play a part in the return value have side effects. The various sequencing constructs in Lisp are used only in program fragments that do not conform to a purely functional programming model.

Even in imperative languages, there is debate as to the value of certain kinds of side effects. In Euclid and Turing, for example, functions (that is, subroutines that return values, and that therefore can appear within expressions) are not permitted to have side effects. Among other things, side-effect freedom ensures that a Euclid or Turing function, like its counterpart in mathematics, is always *idempotent*: if called repeatedly with the same set of arguments, it will always return the same value, and the number of consecutive calls (after the first) will not affect the results of subsequent execution. In addition, side-effect freedom for functions means that the value of a subexpression will never depend on whether that subexpression is evaluated before or after calling a function in some other subexpression. These properties make it easier for a programmer or theorem-proving system to reason about program behavior. They also simplify code improvement, for example by permitting the safe rearrangement of expressions.

EXAMPLE 6.44
Side effects in a random number generator

Unfortunately, there are some situations in which side effects in functions are highly desirable. We saw one example in the gen_new_name function of Figure 3.6 (page 125). Another arises in the typical interface to a pseudo-random number generator.

```
procedure srand(seed : integer)
    –– Initialize internal tables.
    –– The pseudo-random generator will return a different
    –– sequence of values for each different value of seed.

function rand() : integer
    –– No arguments; returns a new "random" number.
```

Obviously rand needs to have a side effect, so that it will return a different value each time it is called. One could always recast it as a procedure with a reference parameter:

```
procedure rand(var n : integer)
```

but most programmers would find this less appealing. Ada strikes a compromise: it allows side effects in functions in the form of changes to static or global variables, but does not allow a function to modify its parameters. ■

6.4 Selection

EXAMPLE 6.45
Selection in Algol 60

Selection statements in most imperative languages employ some variant of the `if... then ... else` notation introduced in Algol 60:

```
if condition then statement
else if condition then statement
else if condition then statement
...
else statement
```
■

As we saw in Section 2.3.2, languages differ in the details of the syntax. In Algol 60 and Pascal both the `then` clause and the `else` clause are defined to contain a single statement (this can of course be a `begin... end` compound statement). To avoid grammatical ambiguity, Algol 60 requires that the statement after the `then` begin with something other than `if` (`begin` is fine). Pascal eliminates this restriction in favor of a "disambiguating rule" that associates an `else` with the closest unmatched `then`. Algol 68, Fortran 77, and more modern languages avoid the ambiguity by allowing a statement *list* to follow either `then` or `else`, with a terminating keyword at the end of the construct.

EXAMPLE 6.46
Elsif/elif

To keep terminators from piling up at the end of nested `if` statements, most languages with terminators provide a special `elsif` or `elif` keyword. In Modula-2, one writes

```
IF a = b THEN ...
ELSIF a = c THEN ...
ELSIF a = d THEN ...
ELSE ...
END
```

■

EXAMPLE 6.47
Cond in Lisp

In Lisp, the equivalent construct is

```
(cond
    ((= A B)
        (...))
    ((= A C)
        (...))
    ((= A D)
        (...))
    (T
        (...)))
```

Here `cond` takes as arguments a sequence of pairs. In each pair the first element is a condition; the second is an expression to be returned as the value of the overall construct if the condition evaluates to `T` (`T` means “true” in most Lisp dialects). ■

6.4.1 Short-Circuited Conditions

While the condition in an `if`... `then` ... `else` statement is a Boolean expression, there is usually no need for evaluation of that expression to result in a Boolean value in a register. Most machines provide conditional branch instructions that capture simple comparisons. Put another way, the purpose of the Boolean expression in a selection statement is not to compute a value to be stored, but to cause control to branch to various locations. This observation allows us to generate particularly efficient code (called *jump code*) for expressions that are amenable to the short-circuit evaluation of Section 6.1.5. Jump code is applicable not only to selection statements such as `if`... `then` ... `else`, but to logically controlled loops as well; we will consider the latter in Section 6.5.5.

In the usual process of code generation, either via an attribute grammar or via ad hoc syntax tree decoration, a synthesized attribute of the root of an expression subtree acquires the name of a register into which the value of the expression will be computed at run time. The surrounding context then uses this register name when generating code that uses the expression. In jump code, *inherited* attributes of the root inform it of the addresses to which control should branch if the expression is true or false respectively. Jump code can be generated quite elegantly by an attribute grammar, particularly one that is *not* L-attributed (Exercise 6.9).

EXAMPLE 6.48
Code generation for a Boolean condition

Suppose, for example, that we are generating code for the following source.

```
if ((A > B) and (C > D)) or (E ≠ F) then
    then_clause
else
    else_clause
```

In Pascal, which does not use short-circuit evaluation, the output code would look something like this.

```
    r1 := A              –– load
    r2 := B
    r1 := r1 > r2
    r2 := C
    r3 := D
    r2 := r2 > r3
    r1 := r1 & r2
    r2 := E
    r3 := F
    r2 := r2 ≠ r3
    r1 := r1 | r2
    if r1 = 0 goto L2
L1: then_clause          –– (label not actually used)
    goto L3
L2: else_clause
L3:
```

The root of the subtree for ((A > B) and (C > D)) or (E ≠ F) would name r1 as the register containing the expression value. ■

EXAMPLE 6.49
Code generation for short-circuiting

In jump code, by contrast, the inherited attributes of the condition's root would indicate that control should "fall through" to L1 if the condition is true, or branch to L2 if the condition is false. Output code would then look something like this:

```
    r1 := A
    r2 := B
    if r1 <= r2 goto L4
    r1 := C
    r2 := D
    if r1 > r2 goto L1
L4: r1 := E
    r2 := F
    if r1 = r2 goto L2
L1: then_clause
    goto L3
L2: else_clause
L3:
```

Here the value of the Boolean condition is never explicitly placed into a register. Rather it is implicit in the flow of control. Moreover for most values of A, B, C, D, and E, the execution path through the jump code is shorter and therefore faster (assuming good branch prediction) than the straight-line code that calculates the value of every subexpression. ■

EXAMPLE 6.50
Short-circuit creation of a Boolean value

If the value of a short-circuited expression is needed explicitly, it can of course be generated, while still using jump code for efficiency. The Ada fragment

```
found_it := p /= null and then p.key = val;
```

is equivalent to

```
if p /= null and then p.key = val then
    found_it := true;
else
    found_it := false;
end if;
```

and can be translated as

```
      r1 := p
      if r1 = 0 goto L1
      r2 := r1→key
      if r2 ≠ val goto L1
      r1 := 1
      goto L2
L1:   r1 := 0
L2:   found_it := r1
```

The astute reader will notice that the first goto L1 can be replaced by goto L2, since r1 already contains a zero in this case. The code improvement phase of the compiler will notice this also, and make the change. It is easier to fix this sort of thing in the code improver than it is to generate the better version of the code in the first place. The code improver has to be able to recognize jumps to redundant instructions for other reasons anyway; there is no point in building special cases into the short-circuit evaluation routines. ■

DESIGN & IMPLEMENTATION

Short-circuit evaluation

Short-circuit evaluation is one of those happy cases in programming language design where a clever language feature yields both more useful semantics *and* a faster implementation than existing alternatives. Other at least arguable examples include `case` statements, local scopes for `for` loop indices (Section 6.5.1), `with` statements in Pascal (Section 7.3.3), and parameter modes in Ada (Section 8.3.1).

6.4.2 `Case`/`Switch` Statements

EXAMPLE 6.51
`Case` statements and nested `if`s

The `case` statements of Algol W and its descendants provide alternative syntax for a special case of nested `if`... `then` ... `else`. When each condition compares the same integer expression to a different compile-time constant, then the following code (written here in Modula-2)

```
i := ... (* potentially complicated expression *)
IF i = 1 THEN
    clause_A
ELSIF i IN 2, 7 THEN
    clause_B
ELSIF i IN 3..5 THEN
    clause_C
ELSIF (i = 10) THEN
    clause_D
ELSE
    clause_E
END
```

can be rewritten as

```
CASE ... (* potentially complicated expression *) OF
    1:      clause_A
|   2, 7:   clause_B
|   3..5:   clause_C
|   10:     clause_D
    ELSE    clause_E
END
```

The elided code fragments (*clause_A*, *clause_B*, etc.) after the colons and the `ELSE` are called the *arms* of the `CASE` statement. The lists of constants in front of the colons are `CASE` statement *labels*. The constants in the label lists must be disjoint, and must be of a type compatible with the tested expression. Most languages allow this type to be anything whose values are discrete: integers, characters, enumerations, and subranges of the same. C# allows strings as well. ■

EXAMPLE 6.52
Translation of nested `if`s

The `CASE` statement version of the code above is certainly less verbose than the `IF`... `THEN` ... `ELSE` version, but syntactic elegance is not the principal motivation for providing a `CASE` statement in a programming language. The principal motivation is to facilitate the generation of efficient target code. The `IF`... `THEN` ... `ELSE` statement is most naturally translated as follows.

```
      r1 := ...              –– calculate tested expression
      if r1 ≠ 1 goto L1
      clause_A
      goto L6
L1:   if r1 = 2 goto L2
      if r1 ≠ 7 goto L3
L2:   clause_B
      goto L6
```

```
     goto L6            –– jump to code to compute address
L1:  clause_A
     goto L7
L2:  clause_B
     goto L7
L3:  clause_C
     goto L7
     . . .
L4:  clause_D
     goto L7
L5:  clause_E
     goto L7

L6:  r1 := . . .        –– computed target of branch
     goto *r1
L7:
```

Figure 6.4 General form of target code generated for a five-arm case statement. One could eliminate the initial goto L6 and the final goto L7 by computing the target of the branch at the top of the generated code, but it may be cumbersome to do so, particularly in a one-pass compiler. The form shown adds only a single jump to the control flow in most cases, and allows the code for all of the arms of the `case` statement to be generated as encountered, before the code to determine the target of the branch can be deduced.

```
L3:  if r1 < 3 goto L4
     if r1 > 5 goto L4
     clause_C
     goto L6
L4:  if r1 ≠ 10 goto L5
     clause_D
     goto L6
L5:  clause_E
L6:
```

EXAMPLE 6.53
Jump tables

Rather than test its expression sequentially against a series of possible values, the `case` statement is meant to *compute* an address to which it jumps in a single instruction. The general form of the target code generated from a `case` statement appears in Figure 6.4. The code at label L6 can take any of several forms. The most common of these simply indexes into an array:

```
T:   &L1                –– tested expression = 1
     &L2
     &L3
     &L3
     &L3
     &L5
     &L2
     &L5
     &L5
     &L4                –– tested expression = 10
```

```
L6:  r1 := . . .              −− calculate tested expression
     if r1 < 1 goto L5
     if r1 > 10 goto L5       −− L5 is the "else" arm
     r1 −:= 1                 −− subtract off lower bound
     r2 := T[r1]
     goto *r2
L7:
```

Here the "code" at label T is actually a table of addresses, known as a *jump table*. It contains one entry for each integer between the lowest and highest values, inclusive, found among the case statement labels. The code at L6 checks to make sure that the tested expression is within the bounds of the array (if not, we should execute the else arm of the case statement). It then fetches the corresponding entry from the table and branches to it. ■

Alternative Implementations

A linear jump table is fast. It is also space-efficient when the overall set of case statement labels is dense and does not contain large ranges. It can consume an extraordinarily large amount of space, however, if the set of labels is nondense or includes large value ranges. Alternative methods to compute the address to which to branch include sequential testing, hashing, and binary search. Sequential testing (as in an if... then ... else statement) is the method of choice if the total number of case statement labels is small. It runs in time $O(n)$, where n is the number of labels. A hash table is attractive if the range of label values is large but has many missing values and no large ranges. With an appropriate hash function it will run in time $O(1)$. Unfortunately, a hash table requires a separate entry for each possible value of the tested expression, making it unsuitable for statements with large value ranges. Binary search can accommodate ranges easily. It runs in time $O(\log n)$, with a relatively low constant factor.

To generate good code for all possible case statements, a compiler needs to be prepared to use a variety of strategies. During compilation it can generate code for the various arms of the case statement as it finds them, while simultaneously building up an internal data structure to describe the label set. Once it has seen all the arms, it can decide which form of target code to generate. For the sake of simplicity, most compilers employ only some of the possible implementations. Many use binary search in lieu of hashing. Some generate only indexed jump tables; others only that plus sequential testing. Users of less sophisticated compilers may need to restructure their case statements if the generated code turns out to be unexpectedly large or slow.

Syntax and Label Semantics

As with if... then ... else statements, the syntactic details of case statements vary from language to language. In keeping with the style of its other structured statements, Pascal defines each arm of a case statement to contain a single statement; begin... end delimiters are required to bracket statement lists. Modula, Ada, Fortran 90, and many other languages expect arms to contain statement lists by default. Modula uses `|` to separate an arm from the following label. Ada brackets labels with `when` and `=>`.

Standard Pascal does not include a default clause: all values on which to take action must appear explicitly in label lists. It is a dynamic semantic error for the expression to evaluate to a value that does not appear. Most Pascal compilers permit the programmer to add a default clause, labeled either `else` or `otherwise`, as a language extension. Modula allows an optional `else` clause. If one does not appear in a given `case` statement, then it is a dynamic semantic error for the tested expression to evaluate to a missing value. Ada requires arm labels to cover *all* possible values in the domain of the type of the tested expression. If the type of tested expression has a very large number of values, then this coverage must be accomplished using ranges or an `others` clause. In some languages, notably C and Fortran 90, it is *not* an error for the tested expression to evaluate to a missing value. Rather, the entire construct has no effect when the value is missing.

The C `switch` *Statement*

C's syntax for `case` (`switch`) statements (retained by C++ and Java) is unusual in other respects.

```
switch (... /* tested expression */) {
    case 1:  clause_A
             break;
    case 2:
    case 7:  clause_B
             break;
    case 3:

    case 4:
    case 5:  clause_C
             break;
    case 10: clause_D
             break;
    default: clause_E
             break;
}
```

DESIGN & IMPLEMENTATION

`Case` statements

`Case` statements are one of the clearest examples of language design driven by implementation. Their primary reason for existence is to facilitate the generation of jump tables. Ranges in label lists (not permitted in Pascal or C) may reduce efficiency slightly, but binary search is still dramatically faster than the equivalent series of `if`s.

Here each possible value for the tested expression must have its own label within the switch; ranges are not allowed. In fact, lists of labels are not allowed, but the effect of lists can be achieved by allowing a label (such as 2, 3, and 4 above) to have an *empty* arm that simply "falls through" into the code for the subsequent label. Because of the provision for fall-through, an explicit break statement must be used to get out of the switch at the end of an arm, rather than falling through into the next. There are rare circumstances in which the ability to fall through is convenient:

EXAMPLE 6.54
Fall-through in C switch statements

```
letter_case = lower;
switch (c) {
    ...
    case 'A' :
        letter_case = upper;
        /* FALL THROUGH! */
    case 'a' :
        ...
        break;
    ...
}
```

Most of the time, however, the need to insert a break at the end of each arm—and the compiler's willingness to accept arms without breaks, silently—is a recipe for unexpected and difficult-to-diagnose bugs. C# retains the familiar C syntax, including multiple consecutive labels, but requires every nonempty arm to end with a break, goto, continue, or return.

Historical Origins

EXAMPLE 6.55
Fortran computed goto

Modern case statements are a descendant of the computed goto statement of Fortran and the switch construct of Algol 60. In early versions of Fortran, one could specify multiway branching based on an integer value as follows.

```
goto (15, 100, 150, 200), I
```

If I is one, control jumps to the statement labeled 15. If I is two, control jumps to the statement labeled 100. If I is outside the range 1...4, the statement has no effect. Any integer-valued expression could be used in place of I. Computed gotos are still allowed in Fortran 90 but are identified by the language manual as a *deprecated* feature, retained to facilitate compilation of old programs.

EXAMPLE 6.56
Algol 60 switch

In Algol 60, a switch is essentially an array of labels:

```
switch S := L15, L100, L150, L200;
...
goto S[I];
```

Algol 68 eliminates the gotos by, in essence, indexing into an array of statements, but the syntax is rather cumbersome.

✓ **CHECK YOUR UNDERSTANDING**

19. List the principal uses of `goto`, and the structured alternatives to each.
20. Explain the distinction between exceptions and multilevel returns.
21. What are *continuations*? What other language features do they subsume?
22. Why is sequencing a comparatively unimportant form of control flow in Lisp?
23. Explain why it may sometimes be useful for a function to have side effects.
24. Describe the *jump code* implementation of short-circuit Boolean evaluation.
25. Why do imperative languages commonly provide a `case` statement in addition to `if... then ... else`?
26. Describe three different search strategies that might be employed in the implementation of a `case` statement, and the circumstances in which each would be desirable.

6.5 Iteration

Iteration and recursion are the two mechanisms that allow a computer to perform similar operations repeatedly. Without at least one of these mechanisms, the running time of a program (and hence the amount of work it can do and the amount of space it can use) is a linear function of the size of the program text, and the computational power of the language is no greater than that of a finite automaton. In a very real sense, it is iteration and recursion that make computers useful. In this section we focus on iteration. Recursion is the subject of Section 6.6.

Programmers in imperative languages tend to use iteration more than they use recursion (recursion is more common in functional languages). In most languages, iteration takes the form of *loops*. Like the statements in a sequence, the iterations of a loop are generally executed for their side effects: their modifications of variables. Loops come in two principal varieties; these differ in the mechanisms used to determine how many times they iterate. An *enumeration-controlled* loop is executed once for every value in a given finite set. The number of iterations is therefore known before the first iteration begins. A *logically controlled* loop is executed until some Boolean condition (which must necessarily depend on values altered in the loop) changes value. The two forms of loops share a single construct in Algol 60. They are distinct in most later languages, with the notable exception of Common Lisp, whose `loop` macro provides an astonishing array of options for initialization, index modification, termination detection, conditional execution, and value accumulation.

6.5.1 Enumeration-Controlled Loops

EXAMPLE 6.57
Early Fortran `do` loop

Enumeration-controlled loops are as old as Fortran. The Fortran syntax and semantics have evolved considerably over time. In Fortran I, II, and IV a loop looks something like this:

```
       do 10 i = 1, 10, 2
           ...
   10  continue
```

The number after the `do` is a label that must appear on some statement later in the current subroutine; the statement it labels is the last one in the *body* of the loop: the code that is to be executed multiple times. `Continue` is a "no-op": a statement that has no effect. Using a `continue` for the final statement of the loop makes it easier to modify code later: additional "real" statements can be added to the bottom of the loop without moving the label.[5] ■

The variable name after the label is the *index* of the loop. The comma-separated values after the equals sign indicate the initial value of the index, the maximum value it is permitted to take, and the amount by which it is to increase in each iteration (this is called the *step size*). A bit more precisely, the loop above is equivalent to

EXAMPLE 6.58
Meaning of a `do` loop

```
       i = 1
   10      ...
       i = i + 2
       if i <= 10 goto 10
```

Index variable `i` in this example will take on the values 1, 3, 5, 7, and 9 in successive loop iterations. Compilers can translate this loop into very simple, fast code for most machines. ■

In practice, unfortunately, this early form of loop proved to have several problems. Some of these problems were comparatively minor. The loop bounds and step size (1, 10, and 2 in our example) were required to be positive integer constants or variables: no expressions were allowed. Fortran 77 removed this restriction, allowing arbitrary positive and negative integer and real expressions. Also, as we saw in Section 2.16 (page 57), trivial lexical errors can cause a Fortran IV compiler to misinterpret the code as an ordinary sequence of statements beginning with an assignment. Fortran 77 makes such misinterpretation less likely by allowing an extra comma after the label in the `do` loop header. Fortran 90 takes back (makes "obsolescent") the ability to use real numbers for loop bounds and step sizes. The problem with reals is that limited precision can cause comparisons (e.g., between the index and the upper bound) to produce unexpected or even implementation-dependent results when the values are close to one another.

5 The `continue` statement of C probably takes its name from this typical use of the no-op in Fortran, but its semantics are very different: the C `continue` starts the next iteration of the loop even when the current one has not finished.

The more serious problems with the Fortran IV `do` loop are a bit more subtle:

- If statements in the body of the loop (or in subroutines called from the body of the loop) change the value of `i`, then the loop may execute a different number of times than one would assume based on the bounds in its header. If the effect is accidental, the bug is hard to find. If the effect is intentional, the code is hard to read.
- `Goto` statements may jump into or out of the loop. Code that jumps out and (optionally) back in again is expressly allowed (if difficult to understand). On the other hand, code that simply jumps in, without properly initializing `i`, almost certainly represents a programming error, but will not be caught by the compiler.
- If control leaves a `do` loop via a `goto`, the value of `i` is the one most recently assigned. If the loop terminates normally, however, the value of `i` is implementation-dependent. Based on Example 6.58, one might expect the final value to be the first one outside the loop bounds: $L + (\lfloor (U - L)/S \rfloor + 1) \times S$, where L, U, and S are the lower and upper bounds of the loop and the step size, respectively. Unfortunately, if the upper bound is close to the largest value that can be represented given the precision of integers on the target machine, then the increment at the bottom of the final iteration of the loop may cause arithmetic overflow. On most machines this overflow will result in an apparently negative value, which will prevent the loop from terminating correctly. On some it will cause a run-time exception that requires the intervention of the operating system in order to continue execution. To ensure correct termination and/or avoid the cost of an exception, a compiler must generate more complex (and slower) code when it is unable to rule out overflow at compile time. In this event, the index may contain its final value (not the "next" value) after normal termination of the loop.
- Because the test against the upper bound appears at the bottom of the loop, the body will always be executed at least once, even if the "low" bound is larger than the "high" bound.

DESIGN & IMPLEMENTATION

Numerical imprecision

The writers of numerical software know that the results of arithmetic computations are often approximations. A comparison between values that are approximately equal "may go either way." The Fortran 90 designers appear to have decided that such comparisons should be explicit. Fortran 90 `do` loops, like the `for` loops of most other languages, reflect the precision of discrete types. The programmer who wants to control iteration with floating-point values must use an explicit comparison in a pre-test or post-test loop (Section 6.5.5).

These problems arise in a larger context than merely Fortran IV. They must be addressed in the design of enumeration-controlled loops in any language. Consider the arguably more friendly syntax of Modula-2:

EXAMPLE 6.59
Modula-2 `for` loop

```
FOR i := first TO last BY step DO
    ...
END
```

where `first`, `last`, and `step` can be arbitrarily complex expressions of an integer, enumeration, or subrange type. Based on the preceding discussion, one might ask several questions.

1. Can `i`, `first`, and/or `last` be modified in the loop? If so, what is the effect on control?
2. What happens if `first` is larger than `last` (or smaller, in the case of a negative `step`)?
3. What is the value of `i` when the loop is finished?
4. Can control jump into the loop from outside?

We address these questions in the paragraphs below. ■

Changes to Loop Indices or Bounds

Most languages, including Algol 68, Pascal, Ada, Fortran 77 and 90, and Modula-3, prohibit changes to the loop index within the body of an enumeration-controlled loop. They also guarantee to evaluate the bounds of the loop exactly once, before the first iteration, so any changes to variables on which those bounds depend will not have any effect on the number of iterations executed. Modula-2 is vague; the manual says that the index "should not be changed" by the body of the loop [Wir85b, Sec. 9.8]. ISO Pascal goes to considerable lengths to prohibit modification. Paraphrasing slightly, it says [Int90, Sec. 6.8.3.9] that the index variable must be declared in the closest enclosing block, and that neither the body of the `for` statement itself nor any statement contained in a subroutine local to the block can "threaten" the index variable. A statement is said to threaten a variable if it

- Assigns to it
- Passes it to a subroutine by reference
- Reads it from a file
- Is a structured statement containing a simpler statement that threatens it

The prohibition against threats in local subroutines is made because a local variable will be accessible to those subroutines, and one of them, if called from within the loop, might change the value of the variable even if it is not passed to it by reference.

Empty Bounds

Modern languages refrain from executing an enumeration-controlled loop if the bounds are empty. In other words, they test the terminating condition *before* the first iteration. The initial test requires a few extra instructions but leads to much more intuitive behavior. The loop

EXAMPLE 6.60
Obvious translation of a `for` loop

```
FOR i := first TO last BY step DO
    ...
END
```

can be translated as

```
     r1 := first
     r2 := step
     r3 := last
L1:  if r1 > r3 goto L2
     ...                   -- loop body; use r1 for i
     r1 := r1 + r2
     goto L1
L2:
```

■

EXAMPLE 6.61
`For` loop translation with test at the bottom

A slightly better if less straightforward translation is

```
     r1 := first
     r2 := step
     r3 := last
     goto L2
L1:  ...                   -- loop body; use r1 for i
     r1 := r1 + r2
L2:  if r1 ≤ r3 goto L1
```

The advantage of this second version is that each iteration of the loop contains a single conditional branch, rather than a conditional branch at the top and an unconditional branch at the bottom. (We will consider yet another version in Exercise ◎ 15.4.) ■

The translations shown above work only if $\mathsf{first} + (\lfloor(\mathsf{last} - \mathsf{first})/\mathsf{step}\rfloor + 1) \times \mathsf{step}$ does not exceed the largest representable integer. If the compiler cannot verify this property at compile time, then it will have to generate more cautious code (to be discussed in Example 6.63).

Loop Direction The astute reader may also have noticed that the code shown here implicitly assumes that `step` is positive. If `step` is negative, the test for termination must "go the other direction." If `step` is not a compile-time constant, then the compiler cannot tell which form of test to use. Some languages, including Pascal and Ada, require the programmer to predict the sign of the step. In Pascal, one must say

EXAMPLE 6.62
Reverse direction `for` loop

```
for i := 10 downto 1 do ...
```

In Ada, one must say

```
for i in reverse 1..10 do ...
```

Modula-2 and Modula-3 do not require special syntax for "backward" loops, but insist that `step` be a compile-time constant so the compiler can tell the difference (Modula (1) has no `for` loop). ■

EXAMPLE 6.63

`For` loop translation with iteration count

In Fortran 77 and Fortran 90, which have neither a special "backward" syntax nor a requirement for compile-time constant steps, the compiler can use an "iteration count" variable to control the loop:

```
     r1 := first
     r2 := step
     r3 := max(⌊(last − first + step)/step⌋, 0)          -- iteration count
             -- NB: this calculation may require several instructions.
             -- It is guaranteed to result in a value within the precision
                of the machine,
             -- but we have to be careful to avoid overflow during its calculation.
     if r3 ≤ 0 goto L2
L1:  ...                         -- loop body; use r1 for i
     r1 := r1 + r2
     r3 := r3 − 1
     if r3 > 0 goto L1
     i := r1
L2:
```

The use of the iteration count avoids the need to test the sign of `step` within the loop. It also avoids problems with overflow when testing the terminating condition (assuming that we have been suitably careful in calculating the iteration count). Some processors, including the PowerPC, PA-RISC, and most CISC machines, can decrement the iteration count, test it against zero, and conditionally branch, all in a single instruction. In simple cases, the code improvement phase of the compiler may be able to use a technique known as *induction variable elimination* to eliminate the need to maintain both r1 and r3. ■

Access to the Index Outside the Loop

Several languages, including Fortran IV and Pascal, leave the value of the loop index undefined after termination of the loop. Others, such as Fortran 77 and Algol 60, guarantee that the value is the one "most recently assigned." For "normal" termination of the loop, this is the first value that exceeds the upper bound. It is not clear what happens if this value exceeds the largest value representable on the machine (or the smallest value in the case of a negative step size). A similar question arises in Pascal, in which the type of an index can be a subrange or enumeration. In this case the first value "after" the upper bound can often be invalid.

EXAMPLE 6.64

Index value after loop

```
var c : 'a'..'z';
...
for c := 'a' to 'z' do begin
    ...
end;
(* what comes after 'z'? *)
```

■

EXAMPLE 6.65

Preserving the final index value

Examples like this illustrate the rationale for leaving the final value of the index undefined in Pascal. The alternative—defining the value to be the last one that was valid—would force the compiler to generate slower code for every loop, with two branches in each iteration instead of one:

```
     r1 := 'a'
     r2 := 'z'
     if r1 > r2 goto L3      –– Code improver may remove this test,
                             –– since 'a' and 'z' are constants.
L1:  . . .                   –– loop body; use r1 for i
     if r1 = r2 goto L2
     r1 := r1 + 1
     –– NB: Pascal step size is always 1 (or −1 if downto)
     goto L1
L2:  i := r1
L3:
```

Note that the compiler must generate this sort of code in any event (or use an iteration count) if arithmetic overflow may interfere with testing the terminating condition. ■

Several languages, including Algol W, Algol 68, Ada, Modula-3, and C++, avoid the issue of the value held by the index outside the loop by making the index a local variable *of* the loop. The header of the loop is considered to contain a *declaration* of the index. Its type is inferred from the bounds of the loop, and its scope is the loop's body. Because the index is not visible outside the loop, its value is not an issue. Since it is not visible even to local subroutines, much of the concept of "threatening" in Pascal becomes unnecessary. Finally, there is no chance that a value held in the index variable before the loop, and needed after, will inadvertently be destroyed. (Of course, the programmer must not give the index the same name as any variable that must be accessed within the loop, but this is a strictly local issue: it has no ramifications outside the loop.)

DESIGN & IMPLEMENTATION

`For` loops

Modern `for` loops reflect the impact of both semantic and implementation challenges. As suggested by the subheadings of Section 6.5.1, the semantic challenges include changes to loop indices or bounds from within the loop, the scope of the index variable (and its value, if any, outside the loop), and `goto`s that enter or leave the loop. Implementation challenges include the imprecision of floating-point values (discussed in the sidebar on page 272), the direction of the bottom-of-loop test, and overflow at the end of the iteration range. The "combination loops" of C (to be discussed in Section 6.5.2) move responsibility for these challenges out of the compiler and into the application program.

Jumps

Algol 60, Fortran 77, and most of their successors place restrictions on the use of the `goto` statement that prevent it from entering a loop from outside. `Goto`s can be used to *exit* a loop prematurely, but this is a comparatively clean operation; questions of uninitialized indices and bounds do not arise. As we shall see in Section 6.5.5, many languages provide an `exit` statement as a semistructured alternative to a loop-escaping `goto`.

6.5.2 Combination Loops

EXAMPLE 6.66
Algol 60 `for` loop

Algol 60, as mentioned above, provides a single loop construct that subsumes the properties of more modern enumeration- and logically controlled loops. The general form is given by

$$\begin{aligned}
\textit{for\_stmt} &\longrightarrow \texttt{for}\ \textit{id}\ \texttt{:=}\ \textit{for\_list}\ \texttt{do}\ \textit{stmt} \\
\textit{for\_list} &\longrightarrow \textit{enumerator}\ (\ \texttt{,}\ \textit{enumerator}\ )^{*} \\
\textit{enumerator} &\longrightarrow \textit{expr} \\
&\longrightarrow \textit{expr}\ \texttt{step}\ \textit{expr}\ \texttt{until}\ \textit{expr} \\
&\longrightarrow \textit{expr}\ \texttt{while}\ \textit{condition}
\end{aligned}$$

Here the index variable takes on values specified by a sequence of enumerators, each of which can be a single value, a range of values similar to that of modern enumeration-controlled loops, or an expression with a terminating condition. Each expression in the current enumerator is reevaluated at the top of the loop. This reevaluation is what makes the `while` form of enumerator useful: its condition typically depends on the current value of the index variable. All of the following are equivalent.

```
for i := 1, 3, 5, 7, 9 do ...
for i := 1 step 2 until 10 do ...
for i := 1, i + 2 while i < 10 do ...
```

In practice the generality of the Algol 60 `for` loop turns out to be overkill. The repeated reevaluation of bounds, in particular, can lead to loops that are very hard to understand. Some of the power of the Algol 60 loop is retained in a cleaner form in the `for` loop of C. A substantially *more* powerful version (not described here) is found in Common Lisp.

C's `for` loop is, strictly speaking, logically controlled. Any enumeration-controlled loop, however, can be rewritten in a logically controlled form (this is of course what the compiler does when it translates into assembler), and C's `for` loop is deliberately designed to facilitate writing the logically controlled equivalent of a Pascal or Algol-style `for` loop. Our Modula-2 example

EXAMPLE 6.67
Combination (`for`) loop in C

```
FOR i := first TO last BY step DO
    ...
END
```

would usually be written in C as

```
for (i = first; i <= last; i += step) {
    ...
}
```

C defines this to be roughly equivalent to

```
i = first;
while (i <= last) {
    ...
    i += step;
}
```

This definition means that it is the programmer's responsibility to worry about the effect of overflow on testing of the terminating condition. It also means that both the index and any variables contained in the terminating condition can be modified by the body of the loop, or by subroutines it calls, and these changes *will* affect the loop control. This, too, is the programmer's responsibility.

Any of the three substatements in the `for` loop header can be null (the condition is considered true if missing). Alternatively, a substatement can consist of a sequence of comma-separated expressions. The advantage of the C `for` loop over its `while` loop equivalent is compactness and clarity. In particular, all of the code affecting the flow of control is localized within the header. In the `while` loop, one must read both the top and the bottom of the loop to know what is going on.

6.5.3 Iterators

In all of the examples we have seen so far (with the possible exception of the combination loops of Algol 60, Common Lisp, or C), a `for` loop iterates over the elements of an arithmetic sequence. In general, however, we may wish to iterate over the elements of any well-defined set (what are often called *containers* or *collections* in object-oriented code). Clu introduced an elegant *iterator* mechanism (also found in Python, Ruby, and C#) to do precisely that. Euclid and several more recent languages, notably C++ and Java, define a standard interface for *iterator objects* (sometimes called *enumerators*) that are equally easy to use but not as easy to write. Icon, conversely, provides a generalization of iterators, known as *generators*, that combines enumeration with backtracking search.[6]

True Iterators

Clu, Python, Ruby, and C# allow any container abstraction to provide an *iterator* that enumerates its items. The iterator resembles a subroutine that is permitted to

6 Unfortunately, terminology is not consistent across languages. Euclid uses the term "generator" for what are called "iterator objects" here. Python uses it for what are called "true iterators" here.

contain `yield` statements, each of which produces a loop index value. `For` loops are then designed to incorporate a call to an iterator. The Modula-2 fragment

EXAMPLE 6.68
Simple iterator in Clu

```
FOR i := first TO last BY step DO
    ...
END
```

would be written as follows in Clu.

```
for i in int$from_to_by(first, last, step) do
    ...
end
```

Here `from_to_by` is a built-in iterator that yields the integers from `first` to $\texttt{first} + \lfloor(\texttt{last} - \texttt{first})/\texttt{step}\rfloor \times \texttt{step}$ in increments of `step`. ■

When called, the iterator calculates the first index value of the loop, which it returns to the main program by executing a `yield` statement. The `yield` behaves like `return`, except that when control transfers back to the iterator after completion of the first iteration of the loop, the iterator continues where it last left off—*not* at the beginning of its code. When the iterator has no more elements to yield it simply returns (without a value), thereby terminating the loop.

In effect, an iterator is a separate thread of control, with its own program counter, whose execution is interleaved with that of the `for` loop to which it supplies index values.[7] The iteration mechanism serves to "decouple" the algorithm required to enumerate elements from the code that uses those elements.

EXAMPLE 6.69
Clu iterator for tree enumeration

As an illustrative example, consider the pre-order enumeration of nodes from a binary tree. A Clu iterator for this task appears in Figure 6.5. Invoked from the header of a `for` loop, it takes the root of a tree as argument. It yields the root node for the first iteration and then calls itself recursively, twice, to enumerate the nodes of the left and right subtrees. ■

Iterator Objects

As realized in most imperative languages, iteration involves both a special form of `for` loop and a mechanism to enumerate values for the loop. These concepts can be separated. Euclid, C++, and Java all provide enumeration-controlled loops reminiscent of those of Clu. They have no `yield` statement, however, and no separate thread-like context to enumerate values; rather, an iterator is an ordinary object (in the object-oriented sense of the word) that provides methods for initialization, generation of the next index value, and testing for completion. Between calls, the state of the iterator must be kept in the object's data members.

EXAMPLE 6.70
Java iterator for tree enumeration

Figure 6.6 contains the Java equivalent of the code in Figure 6.5. The `for` loop at the bottom is syntactic sugar for

7 Because iterators are interleaved with loops in a very regular way, they can be implemented more easily (and cheaply) than fully general threads. We will consider implementation options further in Section Ⓒ 8.6.3.

```
bin_tree = cluster is ..., pre_order, ...           % export list
    node = record [left, right: bin_tree, val: int]
    rep = variant [some: node, empty: null]
    ...
    pre_order = iter(t: cvt) yields(bin_tree)
        tagcase t
            tag empty: return
            tag some(n: node):
                yield(n.val)
                for i: int in pre_order(n.left) do
                    yield(i)
                end
                for i: int in pre_order(n.right) do
                    yield(i)
                end
        end
    end pre_order
    ...
end bin_tree
...
for i: int in bin_tree$pre_order(e) do
    stream$putl(output, int$unparse(i))
end
```

Figure 6.5 Clu iterator for pre-order enumeration of the nodes of a binary tree. In this (simplistic) example we have assumed that the datum in a tree node is simply an `int`. Within the `bin_tree` cluster, the `rep` (representation) declaration indicates that a binary tree is either a `node` or empty. The `cvt` (convert) in the header of `pre_order` indicates that parameter `t` is a `bin_tree` whose internal structure (`rep`) should be visible to the code of `pre_order` itself but not to the caller. In the `for` loop at the bottom, `int$unparse` produces the character string equivalent of a given `int`, and `stream$putl` prints a line to the specified stream.

```
for (Iterator<Integer> it = myTree.iterator(); it.hasNext();) {
    Integer i = it.next();
    System.out.println(i);
}
```

DESIGN & IMPLEMENTATION

"True" iterators and iterator objects

While the `iterator` library mechanisms of C++ and Java are highly useful, it is worth emphasizing that they are *not* the functional equivalents of "true" iterators, as found in Clu, Python, Ruby, and C#. Their key limitation is the need to maintain all intermediate state in the form of explicit data structures, rather than in the program counter and local variables of a resumable execution context.

```
class TreeNode<T> implements Iterable<T> {
    TreeNode<T> left;
    TreeNode<T> right;
    T val;
    ...
    public Iterator<T> iterator() {
        return new TreeIterator(this);
    }
    private class TreeIterator implements Iterator<T> {
        private Stack<TreeNode<T>> s = new Stack<TreeNode<T>>();
        TreeIterator(TreeNode<T> n) {
            s.push(n);
        }
        public boolean hasNext() {
            return !s.empty();
        }
        public T next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            TreeNode<T> n = s.pop();
            if (n.right != null) {
                s.push(n.right);
            }
            if (n.left != null) {
                s.push(n.left);
            }
            return n.val;
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
    ...
}
...
TreeNode<Integer> myTree = ...
...
for (Integer i : myTree) {
    System.out.println(i);
}
```

Figure 6.6 Java code for pre-order enumeration of the nodes of a binary tree. The nested `TreeIterator` class uses an explicit `Stack` object (borrowed from the standard library) to keep track of subtrees whose nodes have yet to be enumerated. Java generics, specified as `<T>` type arguments for `TreeNode`, `Stack`, `Iterator`, and `Iterable`, allow `next` to return an object of the appropriate type (here `Integer`), rather than the undifferentiated `Object`. The `remove` method is part of the `Iterator` interface and must therefore be provided, if only as a placeholder.

The expression following the colon in the concise version of the loop header must support the standard `Iterable` interface, which includes an `iterator()` method that returns an `Iterator` object. ■

EXAMPLE 6.71
Iterator objects in C++

C++ takes a different tack. Rather than propose a special version of the `for` loop that would interface with iterator objects, the designers of the C++ standard library used the language's unusually flexible overloading and reference mechanisms (Sections 3.6.2 and 8.3.1) to redefine comparison (`!=`), increment (`++`), dereference (`*`), and so on, in a way that makes iterating over the elements of a set look very much like using pointer arithmetic (Section 7.7.1) to traverse a conventional array:

```
tree_node<int> *my_tree = ...
...
for (tree_node<int>::iterator n = my_tree->begin();
                              n != my_tree->end(); ++n) {
    cout << *n << "\n";
}
```

C++ encourages programmers to think of iterators as if they were pointers. Iterator `n` in this example encapsulates all the state encapsulated by iterator `it` in the (no syntactic sugar) Java code of Example 6.70. To obtain the next element of the set, however, the C++ programmer "dereferences" `n`, using the `*` or `->` operators. To advance to the following element, the programmer uses the increment (`++`) operator. The `end` method returns a reference to a special iterator that "points beyond the end" of the set. The increment (`++`) operator must return a reference that tests equal to this special iterator when the set has been exhausted. ■

We leave the code of the C++ tree iterator to Exercise 6.15. The details are somewhat messier than Figure 6.6, due to operator overloading, the value model of variables (which requires explicit references and pointers), and the lack of garbage collection. Also, because C++ lacks a common `Object` base class, its container classes are always type-specific. Where generics can minimize the need for type casts in Java and C#, they serve a more fundamental role in C++: without them one cannot write safe, general purpose container code.

Iterating with First-Class Functions

In functional languages, the ability to specify a function "inline" facilitates a programming idiom in which the body of a loop is written as a function, with the loop index as an argument. This function is then passed as the final argument to an iterator. In Scheme we might write

EXAMPLE 6.72
Passing the "loop body" to an iterator in Scheme

```
(define uptoby
  (lambda (low high step f)
    (if (<= low high)
        (begin
         (f low)
         (uptoby (+ low step) high step f))
      '())))
```

We could then sum the first 50 odd numbers as follows.

```
(let ((sum 0))
  (uptoby 1 100 2
          (lambda (i)
            (set! sum (+ sum i))))
  sum)                                  ⟹ 2500
```

Here the body of the loop, `(set! sum (+ sum i))`, is an assignment. The ⟹ symbol (not a part of Scheme) is used here to mean "evaluates to." ■

EXAMPLE 6.73
Iteration with blocks in Smalltalk

Smalltalk, which we consider in Section Ⓒ 9.6.1, provides mechanisms that support a similar idiom:

```
sum <- 0.
1 to: 100 by: 2 do:
    [:i | sum <- sum + i]
```

Like a `lambda` expression in Scheme, a square-bracketed *block* in Smalltalk creates a first-class function, which we then pass as argument to the `to: by: do:` iterator. The iterator calls the function repeatedly, passing successive values of the index variable `i` as argument. Iterators in Ruby employ a similar but somewhat less general mechanism: where a Smalltalk method can take an arbitrary number of blocks as argument, a Ruby method can take only one. Continuations (Section 6.2.2) and lazy evaluation (Section 6.6.2) also allow the Scheme/Lisp programmer to create iterator objects and more traditional style true iterators; we consider these options in Exercises 6.30 and 6.31. ■

Iterating without Iterators

EXAMPLE 6.74
Imitating iterators in C

In a language with neither true iterators nor iterator objects, one can still decouple set enumeration from element use through programming conventions. In C, for example, one might define a `tree_iter` type and associated functions that could be used in a loop as follows.

```
tree_node *my_tree;
tree_iter ti;
...
for (ti_create(my_tree, &ti); !ti_done(ti); ti_next(&ti)) {
    tree_node *n = ti_val(ti);
    ...
}
ti_delete(&ti);
```

There are two principal differences between this code and the more structured alternatives: (1) the syntax of the loop is a good bit less elegant (and arguably more prone to accidental errors), and (2) the code for the iterator is simply a type and some associated functions; C provides no abstraction mechanism to group them together as a module or a class. By providing a standard interface for iterator abstractions, object-oriented languages like C++, Python, Ruby, Java, and C# facilitate the design of higher-order mechanisms that manipulate whole containers: sorting them, merging them, finding their intersection or difference, and so on. We leave the C code for `tree_iter` and the various `ti_` functions to Exercise 6.16. ■

6.5.4 Generators in Icon

Icon generalizes the concept of iterators, providing a *generator* mechanism that causes any expression in which it is embedded to enumerate multiple values on demand.

IN MORE DEPTH

Icon's enumeration-controlled loop, the `every` loop, can contain not only a generator, but any expression that *contains* a generator. Generators can also be used in constructs like `if` statements, which will execute their nested code if *any* generated value makes the condition true, automatically searching through all the possibilities. When generators are nested, Icon explores all possible combinations of generated values, and will even *backtrack* where necessary to undo unsuccessful control-flow branches or assignments.

6.5.5 Logically Controlled Loops

In comparison to enumeration-controlled loops, logically controlled loops have many fewer semantic subtleties. The only real question to be answered is where within the body of the loop the terminating condition is tested. By far the most common approach is to test the condition before each iteration. The familiar `while` loop syntax to do this was introduced in Algol-W and retained in Pascal:

EXAMPLE 6.75
`While` loop in Pascal

```
while condition do statement
```

As with selection statements, most Pascal successors use an explicit terminating keyword, so that the body of the loop can be a statement list. ■

EXAMPLE 6.76
Imitating `while` loops in Fortran 77

Neither (pre-90) Fortran nor Algol 60 really provides a `while` loop construct; their loops were designed to be controlled by enumeration. To obtain the effect of a `while` loop in Fortran 77, one must resort to `goto`s:

```
10  if negated_condition goto 20
    ...
    goto 10
20
```
■

Post-test Loops

EXAMPLE 6.77
Post-test loop in Pascal and Modula

Occasionally it is handy to be able to test the terminating condition at the bottom of a loop. Pascal introduced special syntax for this case, which was retained in Modula but dropped in Ada. A *post-test loop* allows us, for example, to write

```
repeat
    readln(line)
until line[1] = '$';
```

instead of

```
readln(line);
while line[1] <> '$' do
    readln(line);
```

The difference between these constructs is particularly important when the body of the loop is longer. Note that the body of a post-test loop is always executed at least once. ■

EXAMPLE 6.78
Post-test loop in C

C provides a post-test loop whose condition works "the other direction" (i.e., "while" instead of "until"):

```
do {
    line = read_line(stdin);
} while line[0] != '$';
```
■

Midtest Loops

Finally, as we saw in Section 6.2, it is sometimes appropriate to test the terminating condition in the middle of a loop. This "midtest" can be accomplished with an `if` and a `goto` in most languages, but a more structured alternative is preferable. Modula (1) introduced a *midtest*, or *one-and-a-half loop* that allows a terminating condition to be tested as many times as desired within the loop:

EXAMPLE 6.79
Midtest loop in Modula

```
loop
    statement_list
when condition exit
    statement_list
when condition exit
    ...
end
```

Using this notation, the Pascal construct

```
    while true do begin
        readln(line);
        if all_blanks(line) then goto 100;
        consume_line(line)
    end;
 100:
```

can be written as follows in Modula (1).

```
loop
    line := ReadLine;
when AllBlanks(line) exit;
    ConsumeLine(line)
end;
```

The `when` clause here is syntactically part of the `loop` construct. The syntax ensures that an `exit` can occur only within a `loop`, but it has the unfortunate side effect of preventing an exit from within a nested construct. ■

EXAMPLE 6.80
Exit as a separate statement

Modula-2 abandoned the `when` clause in favor of a simpler `EXIT` statement, which is typically placed inside an `IF` statement:

```
LOOP
    line := ReadLine;
    IF AllBlanks(line) THEN EXIT END;
    ConsumeLine(line)
END;
```

Because `EXIT` is no longer part of the `LOOP` construct syntax, the semantic analysis phase of compilation must ensure that `EXIT`s appear only inside `LOOP`s. There may still be an arbitrary number of them inside a given `LOOP`. Modula-3 allows an `EXIT` to leave a `WHILE`, `REPEAT`, or `FOR` loop, as well as a plain `LOOP`. ■

EXAMPLE 6.81
`Break` statement in C

The C `break` statement, which we have already seen in the context of `switch` statements, can be used in a similar manner:

```
for (;;) {
    line = read_line(stdin);
    if (all_blanks(line)) break;
    consume_line(line);
}
```

Here the missing condition in the `for` loop header is assumed to always be true; for some reason, C programmers have traditionally considered this syntax to be stylistically preferable to the equivalent `while (1)`. ■

EXAMPLE 6.82
Exiting a nested loop

In Ada an `exit` statement takes an optional loop-name argument that allows control to escape a nested loop:

```
outer: loop
    get_line(line, length);
    for i in 1..length loop
        exit outer when line(i) = '$';
        consume_char(line(i));
    end loop;
end loop outer;
```

Java extends the C/C++ `break` statement in a similar fashion: Java loops can be labeled as in Ada, and the `break` statement takes an optional loop name as parameter. ■

✓ CHECK YOUR UNDERSTANDING

27. Describe three subtleties in the implementation of enumeration-controlled loops.

28. Why do most languages not allow the bounds or increment of an enumeration-controlled loop to be floating-point numbers?

29. Why do many languages require the step size of an enumeration-controlled loop to be a compile-time constant?
30. Describe the “iteration count” loop implementation. What problem(s) does it solve?
31. What are the advantages of making an index variable local to the loop it controls?
32. What is a *container* (a *collection*)?
33. Explain the difference between true iterators and iterator objects.
34. Cite two advantages of iterator objects over the use of programming conventions in a language like C.
35. Describe the approach to iteration typically employed in languages with first-class functions.
36. Give an example in which a *midtest* loop results in more elegant code than does a pretest or post-test loop.
37. Does C have enumeration-controlled loops? Explain.

6.6 Recursion

Unlike the control-flow mechanisms discussed so far, recursion requires no special syntax. In any language that provides subroutines (particularly functions), all that is required is to permit functions to call themselves, or to call other functions that then call them back in turn. Most programmers learn in a data structures class that recursion and (logically controlled) iteration provide equally powerful means of computing functions: any iterative algorithm can be rewritten, automatically, as a recursive algorithm, and vice versa. We will compare iteration and recursion in more detail in the first subsection below. In the subsection after that we will consider the possibility of passing *unevaluated* expressions into a function. While usually inadvisable, due to implementation cost, this technique will sometimes allow us to write elegant code for functions that are only defined on a subset of the possible inputs, or that explore logically infinite data structures.

6.6.1 Iteration and Recursion

As we noted in Section 3.2, Fortran 77 and certain other languages do not permit recursion. A few functional languages do not permit iteration. Most modern languages, however, provide both mechanisms. Iteration is in some sense the more “natural” of the two in imperative languages, because it is based on the repeated modification of variables. Recursion is the more natural of the two in functional languages, because it does *not* change variables. In the final analysis, which to use in which circumstance is mainly a matter of taste. To compute a sum,

EXAMPLE 6.83
A "naturally iterative" problem

$$\sum_{1 \le i \le 10} f(i)$$

it seems natural to use iteration. In C one would say

```
typedef int (*int_func) (int);
int summation(int_func f, int low, int high) {
    /* assume low <= high */
    int total = 0;
    int i;
    for (i = low; i <= high; i++) {
        total += f(i);
    }
    return total;
}
```

EXAMPLE 6.84
A "naturally recursive" problem

To compute a value defined by a recurrence,

$$\begin{array}{c}\gcd(a, b) \\ \text{(positive integers } a, b\text{)}\end{array} \equiv \begin{cases} a & \text{if } a = b \\ \gcd(a-b, b) & \text{if } a > b \\ \gcd(a, b-a) & \text{if } b > a \end{cases}$$

recursion may seem more natural:

```
int gcd(int a, int b) {
    /* assume a, b > 0 */
    if (a == b) return a;
    else if (a > b) return gcd(a-b, b);
    else return gcd(a, b-a);
}
```

EXAMPLE 6.85
Implementing problems "the other way"

In both these cases, the choice could go the other way:

```
typedef int (*int_func) (int);
int summation(int_func f, int low, int high) {
    /* assume low <= high */
    if (low == high) return f(low);
    else return f(low) + summation(f, low+1, high);
}

int gcd(int a, int b) {
    /* assume a, b > 0 */
    while (a != b) {
        if (a > b) a = a-b;
        else b = b-a;
    }
    return a;
}
```

Tail Recursion

It is often argued that iteration is more efficient than recursion. It is more accurate to say that *naive implementation* of iteration is usually more efficient than naive implementation of recursion. In the preceding examples, the iterative implementations of summation and greatest divisors will be more efficient than the recursive implementations if the latter make real subroutine calls that allocate space on a run-time stack for local variables and bookkeeping information. An "optimizing" compiler, however, particularly one designed for a functional language, will often be able to generate excellent code for recursive functions. It is particularly likely to do so for *tail-recursive* functions such as gcd above. A tail-recursive function is one in which additional computation never follows a recursive call: the return value is simply whatever the recursive call returns. For such functions, dynamically allocated stack space is unnecessary: the compiler can *reuse* the space belonging to the current iteration when it makes the recursive call. In effect, a good compiler will recast our recursive gcd function as

EXAMPLE 6.86
Implementation of tail recursion

```
int gcd(int a, int b) {
    /* assume a, b > 0 */
start:
    if (a == b) return a;
    else if (a > b) {
        a = a-b; goto start;
    } else {
        b = b-a; goto start;
    }
}
```

Even for functions that are not tail-recursive, automatic, often simple transformations can produce tail-recursive code. The general case of the transformation employs conversion to what is known as *continuation-passing style* [FWH01, Chaps. 7–8]. In effect, a recursive function can always avoid doing any work after returning from a recursive call by passing that work into the recursive call, in the form of a continuation.

EXAMPLE 6.87
By-hand creation of tail-recursive code

Some specific transformations (not based on continuation-passing) are often employed by skilled users of functional languages. Consider, for example, the recursive summation function of Example 6.85, written here in Scheme:

```
(define summation (lambda (f low high)
    (if (= low high)
        (f low)                                      ; then part
        (+ (f low) (summation f (+ low 1) high)))))   ; else part
```

Recall that Scheme, like all Lisp dialects, uses Cambridge Polish notation for expressions. The lambda keyword is used to introduce a function. As recursive calls return, our code calculates the sum from "right to left": from high down to low. If the programmer (or compiler) recognizes that addition is associative, we can rewrite the code in a tail-recursive form:

```
(define summation (lambda (f low high subtotal)
    (if (= low high)
        (+ subtotal (f low))
        (summation f (+ low 1) high (+ subtotal (f low))))))
```

Here the `subtotal` parameter accumulates the sum from left to right, passing it into the recursive calls. Because it is tail-recursive, this function can be translated into machine code that does not allocate stack space for recursive calls. Of course, the programmer won't want to pass an explicit `subtotal` parameter to the initial call, so we hide it (the parameter) in an auxiliary, "helper" function:

```
(define summation (lambda (f low high)
  (letrec ((sum-helper (lambda (low subtotal)
             (let ((new_subtotal (+ subtotal (f low))))
               (if (= low high)
                   new_subtotal
                   (sum-helper (+ low 1) new_subtotal))))))
    (sum-helper low 0))))
```

The `let` construct in Scheme serves to introduce a nested scope in which local names (e.g., `new_subtotal`) can be defined. The `letrec` construct permits the definition of recursive functions (e.g., `sum-helper`). ■

Thinking Recursively

EXAMPLE 6.88

Naive recursive Fibonacci function

Detractors of functional programming sometimes argue, incorrectly, that recursion leads to *algorithmically inferior* programs. Fibonacci numbers, for example, are defined by the mathematical recurrence

$$\underset{\text{(nonnegative integer } n)}{F_n} \equiv \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

The naive way to implement this recurrence in Scheme is

```
(define fib (lambda (n)
    (cond ((= n 0) 1)
          ((= n 1) 1)
          (#t (+ (fib (- n 1)) (fib (- n 2)))))))
          ; #t means 'true' in Scheme
```
■

EXAMPLE 6.89

Efficient iterative Fibonacci function

Unfortunately, this algorithm takes exponential time, when linear time is possible. In C, one might write

```
int fib(int n) {
    int f1 = 1; int f2 = 1;
    int i;
    for (i = 2; i <= n; i++) {
        int temp = f1 + f2;
        f1 = f2; f2 = temp;
    }
    return f2;
}
```
■

EXAMPLE 6.90
Efficient tail-recursive Fibonacci function

One can write this iterative algorithm in Scheme: Scheme includes (nonfunctional) iterative features. It is probably better, however, to draw inspiration from the tail-recursive summation function of Example 6.87 and write the following $O(n)$ recursive function.

```
(define fib (lambda (n)
    (letrec ((fib-helper (lambda (f1 f2 i)
               (if (= i n)
                   f2
                   (fib-helper f2 (+ f1 f2) (+ i 1))))))
        (fib-helper 0 1 0))))
```

For a programmer accustomed to writing in a functional style, this code is perfectly natural. One might argue that it isn't "really" recursive; it simply casts an iterative algorithm in a tail-recursive form, and this argument has some merit. Despite the algorithmic similarity, however, there is an important difference between the iterative algorithm in C and the tail-recursive algorithm in Scheme: the latter has no side effects. Each recursive call of the `fib-helper` function creates a new scope, containing new variables. The language implementation may be able to reuse the space occupied by previous instances of the same scope, but it guarantees that this optimization will never introduce bugs. ■

We have already noted that many primarily functional languages, including Common Lisp, Scheme, and ML, provide certain nonfunctional features, including iterative constructs that are executed for their side effects. It is also possible to define an iterative construct as syntactic sugar for tail recursion, by arranging for successive iterations of a loop to introduce new scopes. The only tricky part is to make values from a previous iteration available in the next, when all local names have been reused for different variables. The dataflow language Val [McG82] and its successor, Sisal, provide this capability through a special keyword, `old`. The newer pH language, a parallel dialect of Haskell, provides the inverse keyword, `next`. Figure 6.7 contains side-effect-free iterative code for our Fibonacci function in Sisal. We will mention Sisal and pH again in Sections 10.7 and 12.3.6. ■

EXAMPLE 6.91
Tail-recursive Fibonacci function in Sisal

6.6.2 Applicative- and Normal-Order Evaluation

Throughout the discussion so far we have assumed implicitly that arguments are evaluated before passing them to a subroutine. This need not be the case. It is possible to pass a representation of the *unevaluated* arguments to the subroutine instead, and to evaluate them only when (if) the value is actually needed. The former option (evaluating before the call) is known as *applicative-order evaluation*; the latter (evaluating only when the value is actually needed) is known as *normal-order evaluation*. Normal-order evaluation is what naturally occurs in macros. It also occurs in short-circuit Boolean evaluation, *call-by-name* parameters (to be discussed in Section 8.3.1), and certain functional languages (to be discussed in Section 10.4).

```
function fib(n : integer returns integer)
for initial
    f1 := 0;
    f2 := 1;
    i := 0;
while i < n repeat
    i := old i + 1;
    f1 := old f2;
    f2 := old f1 + old f2;
returns value of f2
end for
end function
```

Figure 6.7 **Fibonacci function in Sisal.** Each iteration of the `while` loop defines a new scope, with new variables named `i`, `f1`, and `f2`. The previous instances of these variables are available in each iteration as `old i`, `old f1`, and `old f2`. The entire `for` construct is an *expression*; it can appear in any context in which a value is expected.

EXAMPLE 6.92
Divisibility macro in C

Historically, C has relied heavily on macros for small, nonrecursive "functions" that need to execute quickly. To determine whether one integer divides another evenly, the C programmer might write

```
#define DIVIDES(a,n) (!((n) % (a)))
/* true iff n has zero remainder modulo a */
```

In every location in which the programmer uses `DIVIDES`, the compiler (actually a preprocessor that runs before the compiler) will substitute the right-hand side of the macro definition, textually, with parameters substituted as appropriate: `DIVIDES(y + z, x)` becomes `(!((x) % (y+z)))`. ■

DESIGN & IMPLEMENTATION

`Inline` as a hint

Formally, the `inline` keyword is a *hint* in C++ and C99, rather than a *directive*: it suggests but does not require that the compiler actually expand the subroutine inline. The compiler is free to use a conventional implementation when `inline` has been specified, or to use an in-line implementation when `inline` has *not* specified, if it has reason to believe that this will result in better code. In effect, the inclusion of the `inline` keyword in the language is an acknowledgment on the part of the language designers that compiler technology is not (yet) at the point where it can always make a better decision with respect to inlining than can an expert programmer. The choice to make `inline` a hint is an acknowledgment that compilers sometimes *are* able to make a better decision, and that their ability to do so is likely to improve over time.

EXAMPLE 6.93
"Gotchas" in C macros

Macros suffer from several limitations. In the code above, for example, the parentheses around `a` and `n` in the right-hand side of the definition are essential. Without them, `DIVIDES(y + z, x)` would be replaced by `(!(x % y + z))`, which is the same as `(!((x % y) + z))`, according to the rules of precedence. More importantly, in a definition like

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

the expression `MAX(x++, y++)` may behave unexpectedly, since the increment side effects will happen more than once. In general, normal-order evaluation is safe only if arguments cause no side effects when evaluated. Finally, because macros are purely textual abbreviations, they cannot be incorporated naturally into high-level naming and scope rules. Given the following definition, for example,

```
#define SWAP(a,b) {int t = (a); (a) = (b); (b) = t;}
```

problems will arise if the programmer writes `SWAP(x, t)`. In C, a macro that "returns" a value must be an expression. Since C is not a completely expression-oriented language like Algol 68, many constructs (e.g., loops) cannot occur within an expression (see Exercise 6.28). ■

All of these problems can be avoided in C by using real functions instead of macros. In most C implementations, however, the macros are much more efficient. They avoid the overhead of the subroutine call mechanism (including register saves and restores), and the code they generate can be integrated into any code improvements that the compiler is able to effect in the code surrounding the call. In C++ and C99, the programmer can obtain the best of both worlds by prefacing a function definition with a special `inline` keyword. This keyword instructs the compiler to expand the definition of the function at the point of call, if possible. The resulting code is then generally as efficient as a macro, but has the semantics of a function call.

Algol 60 uses normal-order evaluation by default (applicative order is also available). This choice was presumably made to mimic the behavior of macros. Most programmers in 1960 wrote mainly in assembler, and were accustomed to macro facilities. Because the parameter-passing mechanisms of Algol 60 are part of the language, rather than textual abbreviations, problems like misinterpreted precedence or naming conflicts do not arise. Side effects, however, are still very much an issue. We will discuss Algol 60 parameters in more detail in Section 8.3.1.

Lazy Evaluation

From the points of view of clarity and efficiency, applicative-order evaluation is generally preferable to normal-order evaluation. It is therefore natural for it to be employed in most languages. In some circumstances, however, normal-order evaluation can actually lead to faster code, or to code that works when applicative-order evaluation would lead to a run-time error. In both cases, what matters is that normal-order evaluation will sometimes not evaluate an argument at all, if its value is never actually needed. Scheme provides for optional normal-order evaluation in the form of built-in functions called `delay` and `force`.[8] These functions provide an implementation of *lazy evaluation*. In the absence of side effects, lazy evaluation has the same semantics as normal-order evaluation, but the implementation keeps track of which expressions have already been evaluated, so it can reuse their values if they are needed more than once in a given referencing environment. A `delay`ed expression is sometimes called a *promise*. The mechanism used to keep track of which promises have already been evaluated is sometimes called *memoization*.[9] Because applicative-order evaluation is the default in Scheme, the programmer must use special syntax not only to pass an unevaluated argument, but also to use it. In Algol 60, subroutine headers indicate which arguments are to be passed which way; the point of call and the uses of parameters within subroutines look the same in either case.

EXAMPLE 6.94
Lazy evaluation of an infinite data structure

A common use of lazy evaluation is to create so-called *infinite* or *lazy data structures* that are "fleshed out" on demand. The following example, adapted from the Scheme manual [ADH$^+$98, p. 28], creates a "list" of all the natural numbers.

```
(define naturals
  (letrec ((next (lambda (n) (cons n (delay (next (+ n 1)))))))
    (next 1)))
(define head car)
(define tail (lambda (stream) (force (cdr stream))))
```

DESIGN & IMPLEMENTATION

Normal-order evaluation

Normal-order evaluation is one of many examples we have seen where arguably desirable semantics have been dismissed by language designers because of fear of implementation cost. Other examples in this chapter include side-effect freedom (which allows normal order to be implemented via lazy evaluation), iterators (Section 6.5.3), sidebar and nondeterminacy (Section 6.7). As noted in the sidebar on page 248, however, there has been a tendency over time to trade a bit of speed for cleaner semantics and increased reliability. Within the functional programming community, Miranda and its successor Haskell are entirely side-effect free, and use normal-order (lazy) evaluation for all parameters.

8 More precisely, `delay` is a *special form*, rather than a function. Its argument is passed to it unevaluated.

9 Within the functional programming community, the term *lazy evaluation* is often used for any implementation that declines to evaluate unneeded function parameters; this includes both naive implementations of normal-order evaluation and the memoizing mechanism described here.

Here cons can be thought of, roughly, as a concatenation operator. Car returns the head of a list; cdr returns everything but the head. Given these definitions, we can access as many natural numbers as we want:

```
(head naturals)                ⟹ 1
(head (tail naturals))         ⟹ 2
(head (tail (tail naturals)))  ⟹ 3
```

The list will occupy only as much space as we have actually explored. More elaborate lazy data structures (e.g., trees) can be valuable in combinatorial search problems, in which a clever algorithm may explore only the "interesting" parts of a potentially enormous search space. ■

6.7 Nondeterminacy

Our final category of control flow is nondeterminacy. A nondeterministic construct is one in which the choice between alternatives (i.e., between control paths) is deliberately unspecified. We have already seen examples of nondeterminacy in the evaluation of expressions (Section 6.1.4): in most languages, operator or subroutine arguments may be evaluated in any order. Some languages, notably Algol 68 and various concurrent languages, provide more extensive nondeterministic mechanisms, which cover statements as well.

IN MORE DEPTH

Absent a nondeterministic construct, the author of a code fragment in which order does not matter must choose some arbitrary (artificial) order. Such a choice can make it more difficult to construct a formal correctness proof. Some language designers have also argued that it is inelegant. The most compelling uses for nondeterminacy arise in concurrent programs, where imposing an arbitrary choice on the order in which a thread interacts with its peers may cause the system as a whole to deadlock. For such programs one may need to ensure that the choice among nondeterministic alternatives is *fair* in some formal sense.

✓ CHECK YOUR UNDERSTANDING

38. What is a *tail-recursive* function? Why is tail recursion important?

39. Explain the difference between *applicative* and *normal-order* evaluation of expressions. Under what circumstances is each desirable?

40. Describe three common pitfalls associated with the use of macros.

41. What is *lazy evaluation*? What are *promises*? What is *memoization*?

42. Give two reasons why lazy evaluation may be desirable.

43. Name a language in which parameters are always evaluated lazily.
44. Give two reasons why a programmer might sometimes want control flow to be *nondeterministic*.

6.8 Summary and Concluding Remarks

In this chapter we introduced the principal forms of control flow found in programming languages: sequencing, selection, iteration, procedural abstraction, recursion, concurrency, and nondeterminacy. Sequencing specifies that certain operations are to occur in order, one after the other. Selection expresses a choice among two or more control-flow alternatives. Iteration and recursion are the two ways to execute operations repeatedly. Recursion defines an operation in terms of simpler instances of itself; it depends on procedural abstraction. Iteration repeats an operation for its side effect(s). Sequencing and iteration are fundamental to imperative (especially von Neumann) programming. Recursion is fundamental to functional programming. Nondeterminacy allows the programmer to leave certain aspects of control flow deliberately unspecified. We touched on concurrency only briefly; it will be the subject of Chapter 12. Procedural abstractions (subroutines) are the subject of Chapter 8.

Our survey of control-flow mechanisms was preceded by a discussion of expression evaluation. We considered the distinction between l-values and r-values, and between the value model of variables, in which a variable is a named container for data, and the reference model of variables, in which a variable is a reference to a data object. We considered issues of precedence, associativity, and ordering within expressions. We examined short-circuit Boolean evaluation and its implementation via jump code, both as a semantic issue that affects the correctness of expressions whose subparts are not always well defined, and as an implementation issue that affects the time required to evaluate complex Boolean expressions.

In our survey we encountered many examples of control-flow constructs whose syntax and semantics have evolved considerably over time. Particularly noteworthy has been the phasing out of `goto`-based control flow and the emergence of a consensus on structured alternatives. While convenience and readability are difficult to quantify, most programmers would agree that the control-flow constructs of a language like Ada are a dramatic improvement over those of, say, Fortran IV. Examples of features in Ada that are specifically designed to rectify control-flow problems in earlier languages include explicit terminators (`end if`, `end loop`, etc.) for structured constructs; `elsif` clauses; label ranges and `others` clauses in `case` statements; implicit declaration of `for` loop indices as read-only local variables; explicit `return` statements; multi-level loop `exit` statements; and exceptions.

The evolution of constructs has been driven by many goals, including ease of programming, semantic elegance, ease of implementation, and run-time efficiency. In some cases these goals have proven complementary. We have seen for example that short-circuit evaluation leads both to faster code and (in many cases) to cleaner semantics. In a similar vein, the introduction of a new local scope for the index variable of an enumeration-controlled loop avoids both the semantic problem of the value of the index after the loop and (to some extent) the implementation problem of potential overflow.

In other cases improvements in language semantics have been considered worth a small cost in run-time efficiency. We saw this in the addition of a pretest to the Fortran `do` loop and in the introduction of midtest loops (which almost always require at least two branch instructions). Iterators provide another example: like many forms of abstraction, they add a modest amount of run-time cost in many cases (e.g., in comparison to explicitly embedding the implementation of the enumerated set in the control flow of the loop), but with a large pay-back in modularity, clarity, and opportunities for code reuse. Sisal's developers would argue that even if Fortran does enjoy a performance edge in some cases, functional programming provides a more important benefit: facilitating the construction of correct, maintainable code. The developers of Java would argue that for many applications the portability and safety provided by extensive semantic checking, standard-format numeric types, and so on are far more important than speed.

The ability of Sisal to compete with Fortran (it does very well with numeric code) is due to advances in compiler technology, and to advances in automatic code improvement in particular. We have seen several other examples of cases in which advances in compiler technology or in the simple willingness of designers to build more complex compilers have made it possible to incorporate features once considered too expensive. Label ranges in Ada `case` statements require that the compiler be prepared to generate code employing binary search. In-line functions in C++ eliminate the need to choose between the inefficiency of tiny functions and the messy semantics of macros. Exceptions (as we shall see in Section 8.5.4) can be implemented in such a way that they incur no cost in the common case (when they do not occur), but the implementation is quite tricky. Iterators, boxing, generics (Section 8.4), and first-class functions are likewise rather tricky, but are increasingly found in mainstream imperative languages.

Some implementation techniques (e.g., rearranging expressions to uncover common subexpressions, or avoiding the evaluation of guards in a nondeterministic construct once an acceptable choice has been found) are sufficiently important to justify a modest burden on the programmer (e.g., adding parentheses where necessary to avoid overflow or ensure numeric stability, or ensuring that expressions in guards are side-effect-free). Other semantically useful mechanisms (e.g., lazy evaluation, continuations, or truly random nondeterminacy) are usually considered complex or expensive enough to be worthwhile only in special circumstances (if at all).

In comparatively primitive languages, we can often obtain some of the benefits of missing features through programming conventions. In early dialects of

Fortran, for example, we can limit the use of `goto`s to patterns that mimic the control flow of more modern languages. In languages without short-circuit evaluation, we can write nested selection statements. In languages without iterators, we can write sets of subroutines that provide equivalent functionality.

6.9 Exercises

6.1 We noted in Section 6.1.1 that most binary arithmetic operators are left-associative in most programming languages. In Section 6.1.4, however, we also noted that most compilers are free to evaluate the operands of a binary operator in either order. Are these statements contradictory? Why or why not?

6.2 As noted in Figure 6.1, Fortran and Pascal give unary and binary minus the same level of precedence. Is this likely to lead to nonintuitive evaluations of certain expressions? Why or why not?

6.3 Translate the following expression into postfix and prefix notation:

$$[-b + \text{sqrt}(b \times b - 4 \times a \times c)]/(2 \times a)$$

Do you need a special symbol for unary negation?

6.4 In Lisp, most of the arithmetic operators are defined to take two or more arguments, rather than strictly two. Thus `(* 2 3 4 5)` evaluates to 120, and `(- 16 9 4)` evaluates to 3. Show that parentheses are necessary to disambiguate arithmetic expressions in Lisp (in other words, give an example of an expression whose meaning is unclear when parentheses are removed).

In Section 6.1.1 we claimed that issues of precedence and associativity do not arise with prefix or postfix notation. Reword this claim to make explicit the hidden assumption.

6.5 Example 6.31 claims that "For certain values of `x`, `(0.1 + x) * 10.0` and `1.0 + (x * 10.0)` can differ by as much as 25%, even when `0.1` and `x` are of the same magnitude." Verify this claim. (*Warning*: If you're using an x86 processor, be aware that floating-point calculations [even on single precision variables] are performed internally with 80 bits of precision. Roundoff errors will appear only when intermediate results are stored out to memory [with limited precision] and read back in again.)

6.6 Languages that employ a reference model of variables also tend to employ automatic garbage collection. Is this more than a coincidence? Explain.

6.7 In Section 6.1.2 we noted that C uses = for assignment and == for equality testing. The language designers state "Since assignment is about twice as frequent as equality testing in typical C programs, it's appropriate that the operator be half as long" [KR88, p. 17]. What do you think of this rationale?

6.8 Consider a language implementation in which we wish to catch every use of an uninitialized variable. In Section 6.1.3 we noted that for types in which every possible bit pattern represents a valid value, extra space must be used to hold an initialized/uninitialized flag. Dynamic checks in such a system can be expensive, largely because of the address calculations needed to access the flags. We can reduce the cost in the common case by having the compiler generate code to automatically initialize every variable with a distinguished *sentinel* value. If at some point we find that a variable's value is different from the sentinel, then that variable must have been initialized. If its value *is* the sentinel, we must double-check the flag. Describe a plausible allocation strategy for initialization flags, and show the assembly language sequences that would be required for dynamic checks, with and without the use of sentinels.

6.9 Write an attribute grammar, based on the following context-free grammar, that accumulates jump code for Boolean expressions (with short-circuiting) into a synthesized attribute of *condition*, and then uses this attribute to generate code for `if` statements.

stmt ⟶ `if` *condition* `then` *stmt* `else` *stmt*
⟶ *other_stmt*
condition ⟶ *c_term* | *condition* `or` *c_term*
c_term ⟶ *relation* | *c_term* `and` *relation*
relation ⟶ *c_fact* | *c_fact comparator c_fact*
c_fact ⟶ `identifier` | `not` *c_fact* | `(` *condition* `)`
comparator ⟶ `<` | `<=` | `=` | `<>` | `>` | `>=`

(*Hint*: Your task will be easier if you do *not* attempt to make the grammar L-attributed. For further details see Fischer and LeBlanc's compiler book [FL88, Sec. 14.1.4].)

6.10 Neither Algol 60 nor Algol 68 employs short-circuit evaluation for Boolean expressions. In both languages, however, an `if`...`then`...`else` construct can be used as an expression. Show how to use `if`...`then`...`else` to achieve the effect of short-circuit evaluation.

6.11 Consider the following expression in C: `a/b > 0 && b/a > 0`. What will be the result of evaluating this expression when `a` is zero? What will be the result when `b` is zero? Would it make sense to try to design a language in which this expression is guaranteed to evaluate to `false` when either `a` or `b` (but not both) is zero? Explain your answer.

6.12 As noted in Section 6.4.2, languages vary in how they handle the situation in which the tested expression in a `case` statement does not appear among the labels on the arms. C and Fortran 90 say the statement has no effect. Pascal and Modula say it results in a dynamic semantic error. Ada says that the labels must *cover* all possible values for the type of the expression, so the question of a missing value can never arise at run time. What are the tradeoffs among these alternatives? Which do you prefer? Why?

6.13 Write the equivalent of Figure 6.5 in C# 2.0, Python, or Ruby. Write a second version that performs an in-order enumeration rather than pre-order.

6.14 Revise the algorithm of Figure 6.6 so that it performs an in-order enumeration, rather than pre-order.

6.15 Write a C++ pre-order iterator to supply tree nodes to the loop in Example 6.71. You will need to know (or learn) how to use pointers, references, inner classes, and operator overloading in C++. For the sake of (relative) simplicity, you may assume that the datum in a tree node is always an `int`; this will save you the need to use generics. You may want to use the `stack` abstraction from the C++ standard library.

6.16 Write code for the `tree_iter` type (`struct`) and the `ti_create`, `ti_done`, `ti_next`, `ti_val`, and `ti_delete` functions employed in Example 6.74.

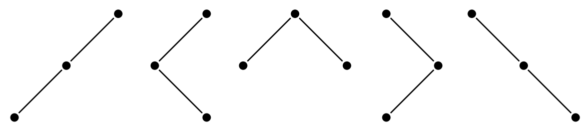
6.17 Write, in C#, Python, or Ruby, an iterator that yields

(a) all permutations of the integers $1..n$

(b) all combinations of k integers from the range $1..n$ ($0 \le k \le n$).

You may represent your permutations and combinations using either a list or an array.

6.18 Use iterators to construct a program that outputs (in some order) all *structurally distinct* binary trees of n nodes. Two trees are considered structurally distinct if they have different numbers of nodes or if their left or right subtrees are structurally distinct. There are, for example, 5 structurally distinct trees of 3 nodes:



These are most easily output in "dotted parenthesized form":

```
(((.).).)
((.(.)).)
((.).(.))
(.((.).))
(.(.(.)))
```

(*Hint*: Think recursively! If you need help, see Section 2.2 of the text by Finkel [Fin96].)

6.19 Build true iterators in Java using threads. (This requires knowledge of material in Chapter 12.) Make your solution as clean and as general as possible. In particular, you should provide the standard `Iterator` or `IEnumerable` interface for use with extended `for` or `foreach` loops, but the programmer should not have to write these. Instead, he or she should write a class with an `Iterate` method, which should in turn be able to call a `Yield` method, which you should also provide. Evaluate the cost of your solution. How much more expensive is it than standard Java iterator objects?

6.20 In an expression-oriented language such as Algol 68 or Lisp, a `while` loop (a `do` loop in Lisp) has a value as an expression. How do you think this value should be determined? (How is it determined in Algol 68 and Lisp?) Is the value a useless artifact of expression orientation, or are there reasonable programs in which it might actually be used? What do you think should happen if the condition on the loop is such that the body is never executed?

6.21 Recall the "blank line" loop of Example 6.80, here written in Modula-2.

```
LOOP
    line := ReadLine;
    IF AllBlanks(line) THEN EXIT END;
    ConsumeLine(line)
END;
```

Show how you might accomplish the same task using a `while` or `repeat` loop, if midtest loops were not available. (*Hint*: One alternative duplicates part of the code; another introduces a Boolean flag variable.) How do these alternatives compare to the midtest version?

6.22 Rubin [Rub87] used the following example (rewritten here in C) to argue in favor of a `goto` statement.

```
int first_zero_row = -1;        /* none */
int i, j;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        if (A[i][j]) goto next;
    }
    first_zero_row = i;
    break;
next: ;
}
```

The intent of the code is to find the first all-zero row, if any, of an $n \times n$ matrix. Do you find the example convincing? Is there a good structured alternative in C? In any language?

6.23 Bentley [Ben86, Chap. 4] provides the following informal description of binary search.

> We are to determine whether the sorted array `X[1..N]` contains the element `T`.... Binary search solves the problem by keeping track of a range within the array in which `T` must be if it is anywhere in the array. Initially, the range is the entire array. The range is shrunk by comparing its middle

element to `T` and discarding half the range. The process continues until `T` is discovered in the array or until the range in which it must lie is known to be empty.

Write code for binary search in your favorite imperative programming language. What loop construct(s) did you find to be most useful? (NB: When he asked more than a hundred professional programmers to solve this problem, Bentley found that only about 10% got it right the first time, without testing.)

6.24 A *loop invariant* is a condition that is guaranteed to be true at a given point within the body of a loop on every iteration. Loop invariants play a major role in *axiomatic semantics*, a formal reasoning system used to prove properties of programs. In a less formal way, programmers who identify (and write down!) the invariants for their loops are more likely to write correct code. Show the loop invariant(s) for your solution to the preceding exercise. (*Hint*: You will find the distinction between $<$ and $\leq$ [or between $>$ and $\geq$] to be crucial.)

6.25 If you have taken a course in automata theory or recursive function theory, explain why `while` loops are strictly more powerful than `for` loops. (If you haven't had such a course, skip this question!) Note that we're referring here to Pascal-style `for` loops, not C-style.

6.26 Show how to calculate the number of iterations of a general Fortran 90-style `do` loop. Your code should be written in an assembler-like notation, and should be guaranteed to work for all valid bounds and step sizes. Be careful of overflow! (*Hint*: While the bounds and step size of the loop can be either positive or negative, you can safely use an unsigned integer for the iteration count.)

6.27 Write a tail-recursive function in Scheme or ML to compute n factorial ($n! = \prod_{1\leq i\leq n} i = 1 \times 2 \times \cdots \times n$). (*Hint*: You will probably want to define a "helper" function, as discussed in Section 6.6.1.)

6.28 Can you write a macro in standard C that "returns" the greatest common divisor of a pair of arguments, without calling a subroutine? Why or why not?

6.29 Give an example in C in which an in-line subroutine may be significantly faster than a functionally equivalent macro. Give another example in which the macro is likely to be faster. (*Hint*: Think about applicative versus normal-order evaluation of arguments.)

6.30 Use lazy evaluation (`delay` and `force`) to implement iterator objects in Scheme. More specifically, let an iterator be either the null list or a pair consisting of an element and a promise that when `force`d will return an iterator. Give code for an `uptoby` function that returns an iterator, and a `for-iter` function that accepts as arguments a one-argument function and an iterator. These should allow you to evaluate such expressions as

```
(for-iter (lambda (e) (display e) (newline)) (uptoby 10 50 3))
```

Note that unlike the standard Scheme `for-each`, `for-iter` should not require the existence of a list containing the elements over which to iterate; the intrinsic space required for `(for-iter f (uptoby 1 n 1))` should be only $O(1)$, rather than $O(n)$.

6.31 (Difficult) Use `call-with-current-continuation` (`call/cc`) to implement the following structured nonlocal control transfers in Scheme. (This requires knowledge of material in Chapter 10.) You will probably want to consult a Scheme manual for documentation not only on `call/cc`, but on `define-syntax` and `dynamic-wind` as well.

(a) Multilevel returns. Model your syntax after the `catch` and `throw` of Common Lisp.

(b) True iterators. In a style reminiscent of Exercise 6.30, let an iterator be a function which when `call/cc`-ed will return either a null list or a pair consisting of an element and an iterator. As in that previous exercise, your implementation should support expressions like

```
(for-iter (lambda (e) (display e) (newline)) (uptoby 10 50 3))
```

Where the implementation of `uptoby` in Exercise 6.30 required the use of `delay` and `force`, however, you should provide an `iterator` macro (a Scheme *special form*) and a `yield` function that allows `uptoby` to look like an ordinary tail-recursive function with an embedded `yield`:

```
(define uptoby
  (iterator (low high step)
    (letrec ((helper
               (lambda (next)
                 (if (> next high) '()
                   (begin                ; else clause
                    (yield next)
                    (helper (+ next step)))))))
             (helper low))))
```

6.32 Explain why the following guarded commands in SR are *not* equivalent.

```
if a < b -> c := a              if a < b -> c := a
[] b < c -> c := b              [] b < c -> c := b
[] else -> c := d               [] true -> c := d
fi                              fi
```

© **6.33–6.35** In More Depth.

6.10 Explorations

6.36 Consider again the idea of *loop unrolling*, introduced in Exercise 5.15. Loop unrolling is traditionally implemented by the code improvement phase of a compiler. It can be implemented at source level, however, if we are faced with the prospect of "hand optimizing" time-critical code on a system whose compiler is not up to the task. Unfortunately, if we replicate the body of a loop k times, we must deal with the possibility that the original number of loop iterations, n, may not be a multiple of k. Writing in C, and letting $k = 4$, we might transform the main loop of Exercise 5.15 from

```
i = 0;
do {
    sum += A[i]; squares += A[i] * A[i]; i++;
} while (i < N);
```

to

```
i = 0;  j = N/4;
do {
    sum += A[i]; squares += A[i] * A[i]; i++;
    sum += A[i]; squares += A[i] * A[i]; i++;
    sum += A[i]; squares += A[i] * A[i]; i++;
    sum += A[i]; squares += A[i] * A[i]; i++;
} while (--j > 0);
do {
    sum += A[i]; squares += A[i] * A[i]; i++;
} while (i < N);
```

In 1983, Tom Duff of Lucasfilm realized that code of this sort can be "simplified" in C by interleaving a `switch` statement and a loop. The result is rather startling, but perfectly valid C. It's known in programming folklore as "Duff's device."

```
i = 0; j = (N+3)/4;
switch (N%4)
    case 0: do{ sum += A[i]; squares += A[i] * A[i]; i++;
    case 3:     sum += A[i]; squares += A[i] * A[i]; i++;
    case 2:     sum += A[i]; squares += A[i] * A[i]; i++;
    case 1:     sum += A[i]; squares += A[i] * A[i]; i++;
            } while (--j > 0);
}
```

Duff announced his discovery with "a combination of pride and revulsion." He noted that "Many people ... have said that the worst feature of C is that `switch`es don't `break` automatically before each `case` label. This code forms some sort of argument in that debate, but I'm not sure whether it's for or against." What do you think? Is it reasonable to interleave a loop and a `switch` in this way? Should a programming language permit it? Is automatic fall-through ever a good idea?

6.37 Using your favorite language and compiler, investigate the order of evaluation of subroutine parameters. Are they usually evaluated left to right or right to left? Are they ever evaluated in the other order? (Can you be sure?) Write a program in which the order makes a difference in the results of the computation.

6.38 Consider the different approaches to arithmetic overflow adopted by Pascal, C, Java, C#, and Common Lisp, as described in Section 6.1.4. Speculate as to the differences in language design goals that might have caused the designers to adopt the approaches they did.

6.39 Learn more about container classes and the *design patterns* (structured programming idioms) they support. Explore the similarities and differences among the standard container libraries of C++, Java, and C#. Which of these libraries do you find the most appealing? Why?

6.40–6.43 In More Depth.

6.11 Bibliographic Notes

Many of the issues discussed in this chapter feature prominently in papers on the history of programming languages. Pointers to several such papers can be found in the Bibliographic Notes for Chapter 1. Fifteen papers comparing Ada, C, and Pascal can be found in the collection edited by Feuer and Gehani [FG84]. References for individual languages can be found in Appendix A.

Niklaus Wirth has been responsible for a series of influential languages over a 30-year period, including Pascal [Wir71], its predecessor Algol W [WH66], and the successors Modula [Wir77b], Modula-2 [Wir85b], and Oberon [Wir88b]. The `case` statement of Algol W is due to Hoare [Hoa81]. Bernstein [Ber85] considers a variety of alternative implementations for `case`, including multilevel versions appropriate for label sets consisting of several dense "clusters" of values. Guarded commands are due to Dijkstra [Dij75]. Duff's device was originally posted to netnews, the predecessor of Usenet news, in May 1984. The original posting appears to have been lost, but Duff's commentary on it can be found at many Internet sites, including *www.lysator.liu.se/c/duffs-device.html*.

Debate over the supposed merits or evils of the `goto` statement dates from at least the early 1960s, but became a good bit more heated in the wake of a 1968 article by Dijkstra ("Go To Statement Considered Harmful" [Dij68b]). The structured programming movement of the 1970s took its name from the text of Dahl, Dijkstra, and Hoare [DDH72]. A dissenting letter by Rubin in 1987 ("'GOTO Considered Harmful' Considered Harmful" [Rub87]; Exercise 6.22) elicited a flurry of responses.

What has been called the "reference model of variables" in this chapter is called the "object model" in Clu; Liskov and Guttag describe it in Sections 2.3 and 2.4.2 of their text on abstraction and specification [LG86]. Clu iterators are described in an article by Liskov et al. [LSAS77] and in Chapter 6 of the Liskov and Guttag text. Icon generators are discussed in Chapters 11 and 14 of the text by Griswold and Griswold [GG96]. The tree-enumeration algorithm of Exercise 6.18 was originally presented (without iterators) by Solomon and Finkel [SF80].

Several texts discuss the use of invariants (Exercise 6.24) as a tool for writing correct programs. Particularly noteworthy are the works of Dijkstra [Dij76] and Gries [Gri81]. Kernighan and Plauger provide a more informal discussion of the art of writing good programs [KP78].

The Blizzard [SFL$^+$94] and Shasta [SG96] systems for software distributed shared memory (S-DSM) make use of sentinels (Exercise 6.8). We will discuss S-DSM in Section 12.2.1.

Michaelson [Mic89, Chap. 8] provides an accessible formal treatment of applicative-order, normal-order, and lazy evaluation. Friedman, Wand, and Haynes provide an excellent discussion of continuation-passing style [FWH01, Chaps. 7–8].