

Chapter 10

Control Structures

Overview

This chapter treats the subject of control structures. A control structure is a language feature which defines the order of evaluation of expressions or larger units. Control structures exist in a variety of forms, the lowest level being the primitive control instructions which are defined by the hardware. These include instruction sequencing, conditional and unconditional jumps, and subroutine calls. Control structures in high-level programming languages developed out of and are implemented in terms of these instructions.

Above the expression level, there are four kinds of control structures. These are subroutine call with parameters, statement sequence, conditional execution, and repetition. In order to be useful, programs must be able to perform a section of code for one particular set of inputs and a different set of actions for another. Functional languages have only control expressions: that is, expressions that contain one or more conditions, and for each condition, an expression to evaluate when the condition is true. Procedural languages, on the other hand, have both control expressions and control statements.

Conditional forms are basic and absolutely essential in a programming language because they are inherently outside-in control structures. The conditional lets us test whether it is safe to proceed with a computation before doing so. Various conditional forms are discussed, ranging from very primitive to very general. The generalized conditional is shown to be the most flexible. and the **CASE** statement the most efficient.

The simplest decision structure is the conditional **IF** \langle condition \rangle **GOTO** \langle label \rangle . Unfortunately, its usage often leads to spaghetti code because the **GOTO** may lead anywhere in the program. This problem is largely resolved by using structured conditionals such as

If..Then..Elseif..Else. The **CASE** statement is an efficient structured conditional control statement with multiple branches. There are three ways to implement the **CASE** statement to achieve greater efficiency than a structured conditional. Many languages support the **CASE**; several of these (including the forms in Pascal, C, and early COBOL) have defects which make the **CASE** statement cumbersome to use.

Iterative control structures are used to process homogeneous data structures such as arrays, lists, and files. The most primitive loop is the infinite loop, which is a one-in/zero-out structure. The two simplest finite loops, formed with a conditional **GOTO**, are the repeat loop (with its test at the end of the loop) and the while loop (with a test at the top). These correspond to the structured **REPEAT** and **WHILE** statements found in most languages. **Ada**, **Turing**, and **FORTH** have combined these forms into a more general loop in which the loop termination test may appear anywhere in the loop.

If the length of a data structure is known, automatic processing can be performed by a counted loop. The **FOR** loop has two parts:

- A control element, containing an initial value for the loop variable, a goal, and an increment.
- A scope, containing a sequence of statements to be iterated.

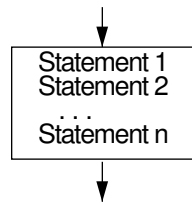
Most languages restrict the expressions that may be used in the control element, and most prohibit changing the loop variable within the scope. The iteration element is a generalization of the counted loop with no restrictions on the expressions within the control element or use of the loop variable. It is not, strictly, a counted loop, and it cannot be implemented as efficiently.

Some languages support implicit iteration. Their basic data structure is a set, array, or list of some sort, and they support a variety of iterative functions to process an entire compound object and return a simple or compound result. Implicitly iterative expressions add great power to a language and permit many complex operations to be expressed briefly and clearly.

10.1 Basic Control Structures

A *control structure* is a language feature defined in the semantics of the language (not in syntax only) that defines the order of evaluation of expressions or larger program units.

The *primitive control instructions* defined by the hardware in all computers are listed below. The control structures in higher languages developed out of these and are implemented in terms of them. Execution starts when the computer is turned on or receives a **RESET** signal. A boot

Exhibit 10.1. Diagram of a sequence of statements.

sequence is built into the computer hardware which causes actual execution to start somewhere, in a location defined by the hardware. In modern computers, the boot sequence initializes whatever is necessary, loads the operating system (the OS), then transfers control to it. The OS then accepts a series of commands and carries them out by loading and transferring control to other system programs or to application programs. The transfer of control to a program is represented graphically by the arrow leading into the top of its control diagram.

10.1.1 Normal Instruction Sequencing

Normally, instructions are executed in the order that they are loaded into memory. This is carried out by the instruction cycle of the machine. A typical version of the cycle is as follows:

1. Instruction fetch: Load the CPU's instruction register from the memory address stored in the instruction counter register (IC).
2. Instruction counter increment: Add 1 to the address stored in the IC, in preparation for fetching the next instruction.
3. Decode the instruction: Connect all the appropriate registers and logic circuits. If the instruction references memory, load the memory address register with the address.
4. Execute: Do the current instruction. If it references memory, do the fetch or store. If it is an arithmetic instruction, perform the operation.

Machine instructions are executed in the order in which they are loaded into memory unless that order is changed by a jump instruction. A sequence of instructions that does not include jumps will be diagrammed as a simple box, as in Exhibit 10.1.

Sequences are a basic control structure in procedural languages. The code section of a program in a procedural language is a sequence of statements; execution starts at the first and progresses to the next until the end is reached. Programmers are trained to analyze their problems as ordered sequences of steps to be performed on a set of objects. The pure functional languages do not support statement sequences. Rather, sequences are eliminated by use of nested function calls and lazy evaluation. Chapter 12 discusses this approach to programming.

10.1.2 Assemblers

A symbolic assembly language is built on the semantic basis provided by the raw machine, machine language, a symbol table, and ways to define variables, labels, and functions. A macro assembler provides source code macros in addition. This semantic basis is completely flexible: using it you can express any action that your machine can do. But it is not a very powerful semantic basis because it lacks any way to convey the programmer's semantic intent to the translator, and the translator has no mechanisms for ensuring that this intent is carried out.

Macros ease the task of writing code but do not extend the semantic basis, since all macros are expanded, that is, replaced by their definitions, *before* translating the program.

Assemblers impose no restrictions, good or bad, on the programmer; they permit the programmer to write good code or totally meaningless code. Higher-level languages impose many kinds of helpful restrictions, including some that can be checked at compile time and others that must be checked at load-link time or at run time. These restrictions help the translator identify meaningless code.

Two of the primitive semantic features supported by assemblers cause a lot of difficulty in debugging and in proving that a program is correct. These are (1) the GOTO, and (2) destructive assignment. The GOTO introduces complex data-dependent flow patterns which cannot be predicted at compile time, and assignment to variables introduces dependency on time and the order of evaluation into the meaning of an expression. Both make proofs of correctness hard to construct.

In the rest of this chapter, we examine aspects of control structures that are supported by higher-level programming languages. Two kinds of control structures above the expression level are basic: subroutine call with parameters and conditional execution (discussed in Section 10.2). Procedural languages have two more basic kinds of control: execution of a sequence of statements and repetition (discussed in Section 10.3). Many languages, especially older ones, provide a fifth kind of control structure: a GOTO [Section 11.1], which is more or less restricted depending on the language. Finally, some languages contain primitives for communication between concurrent, asynchronous processes, such as the task names and `accept` statement in Ada.¹

10.1.3 Sequence, Subroutine Call, IF, and WHILE Suffice

In 1968 Dijkstra² wrote an important and provocative paper that advocated that the GOTO statement should be dropped from programming languages because it was the cause of far too many program errors. This position was based on earlier work by Jacopini³ which showed that any flow diagram can be written, without arbitrary GOTOs, in terms of conditionals, while loops, and sequences [Exhibit 10.2].

Dijkstra did not claim that programs limited to these control structures would be maximally

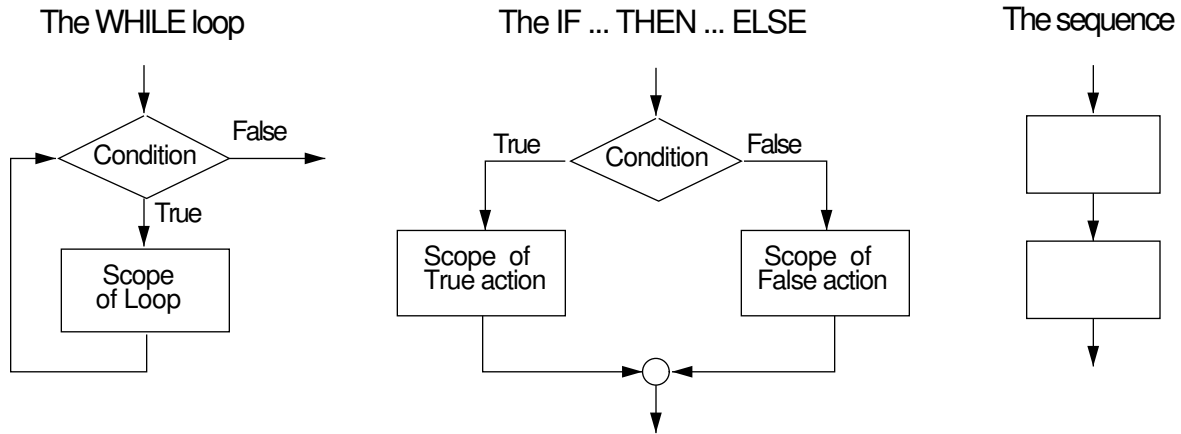
¹Concurrency primitives are beyond the scope of this book.

²Dijkstra's ideas were published first in Europe: Dijkstra [1965]. His ideas became known in the United States after he published a second short paper: Dijkstra [1968].

³Cf. Böhm and Jacopini [1966].

Exhibit 10.2. A sufficient set of basic control structures.

These are the traditional flow diagrams for Dijkstra's three control structures.



efficient, merely that they could be written, and written more rapidly and with less likelihood of errors. His evidence was drawn from real industrial experience and from personal observation.

Each of these control structures is comprised of a group of boxes with connecting arrows. Each group is a *one-in/one-out* structure; that is, on a flowchart, one line comes into the control structure and one line goes out of it. This is an important property: programs limited to these formation patterns are easier to debug because the effects of any transfer of control are confined to a small and defined part of the algorithm.

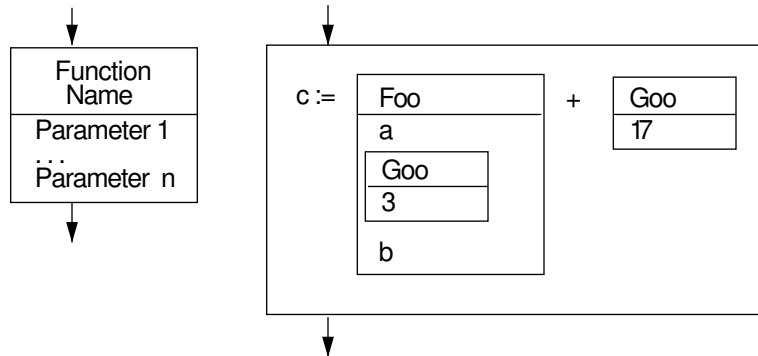
Restricting control to one-in/one-out structures also makes mathematical proofs about the correctness of the program much easier to construct. Such a program can be decomposed into a series of short one-in/one-out sections, and correctness conditions can be written that relate the program state at the top of a section to the state at the bottom. Next, the correctness of each section can be proved independently. Finally, these short proofs can be combined into a whole, yielding a proof of correctness of the entire program whose structure matches the program structure.

The term *structured programming* has no single definition but is generally used to mean a top-down style of program development. The first step is to state the high-level purpose, requirements, and general process of a program. Each process step is then refined by elaborating the definition and adding more detail. In the end, procedures and functions are written to correspond to each portion of the problem definition, and these must be written using only the one-in/one-out control structures provided by the programming language.

Pascal is the "structured" language most widely used now for instruction. It contains recursive functions and procedures, the three control structures just discussed (conditionals, **WHILE** loops, sequences), two additional loop control structures (**FOR** and **REPEAT** loops) [Exhibits 10.27 and 10.23], and a badly designed multibranching conditional structure (**CASE**) [Exhibit 10.18]. These

Exhibit 10.3. Diagram of a subroutine call with parameters.

The left diagram represents a procedure call (a function call statement); the right diagram shows three function calls nested within an expression.



extra control structures are not necessary in a complete language, but they are certainly nice to have. Pascal also contains a GOTO instruction, but students are often not told that it exists and are seldom taught how or why to use it.

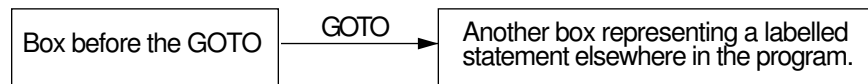
10.1.4 Subroutine Call

This instruction, also called *jump to subroutine*, saves the current value of the IC, then stores the entry address of the subroutine in the IC. *Subroutine return* restores the saved address and brings control back to the instruction after the call. A simple subroutine call box is shown on the left in Exhibit 10.3. This is diagrammed as a two-part box, divided horizontally, with the subroutine name written above the line and the parameters written below [Exhibit 10.3]. Where a subroutine call is nested within another call or within a statement, these boxes can be used to show the nesting structure. For example, the box on the right in Exhibit 10.3 represents the following nested function calls:

```
c := Foo(a, Goo(3), b) + Goo(17);
```

All modern programming languages provide the ability to define and call subroutines with parameters. Without subroutines, it is nearly impossible to write and debug a large program. The programmer must break a large problem into conceptually manageable pieces. Subroutines are the programmer's tool for implementing each part in such a way that the interfaces are clearly defined and the interactions among parts are limited. This methodology, of decomposing a problem into clearly defined parts with limited and clearly defined interactions, then building a program by implementing each, is called "top-down programming". It is now recognized to be the basis of good program design.

To understand why there is no real substitute for subroutines with parameters, let us look at a pair of languages that did not, initially, support them: BASIC and dBASE.

Exhibit 10.4. Diagram of the computer's primitive GOTO instruction.

BASIC was originally defined⁴ lacking a crucial feature. It had a `GOSUB xxx` statement which did a subroutine jump to statement number `xxx`, and a `RETURN` statement which returned control to the calling program, but parameters and results had to be passed through global variables. The programmer could emulate the normal parameter passing mechanism by storing each parameter value in a global variable, calling the subroutine, and using the same global variable name inside the subroutine. There were no “dummy parameter” names. The subroutine would likewise store its results in global variables. Thus what is easy in most languages became cumbersome in BASIC, and the programmer had to be careful not to use the same name twice for two different purposes. Worse, the subroutine's actions and those of the main program were not well isolated from each other.

BASIC was recently revised and updated by the author of the original BASIC language, John Kemeny. The new version is called True BASIC⁵, perhaps to distinguish it both from the original language and from the dozens of nonstandardized extended BASIC implementations that were developed. This new version upgrades the original language extensively, but perhaps the most important improvement is the inclusion of subroutines with parameters.

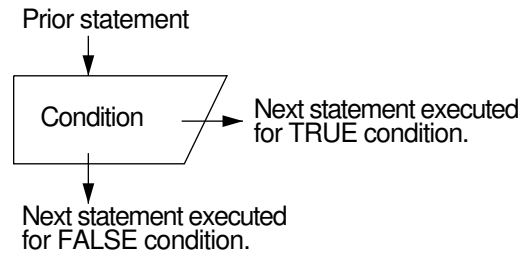
dBASE is a popular data base language for personal computers that includes extensive procedures and functions for forming data bases and handling and extracting data from data base files. In addition, it includes a general programming facility similar to BASIC which can be used to create fancy input and output programs. This general programming facility was added to dBASE in pieces, as it became clear that data base primitives alone were not enough. Flexibility in handling input, output, and some processing and statistics gathering can only be achieved with a general-purpose language. The most recent version of dBASE was extended to include a parameter facility for subroutines. With this extension dBASE has become a general, interpretive, procedural programming language specialized for handling data bases.

10.1.5 Jump and Conditional Jump

Jump and conditional jump instructions change the next instruction to be executed by storing a new address into the IC. They differ from a jump to subroutine instruction in that they do not save or restore the address from which the jump originates. The `GOTO` is an unconstrained transfer of control to some other part of the program. It is diagrammed as an arrow, as in Exhibit 10.4.

⁴Dartmouth [1965].

⁵Kemeny [1985].

Exhibit 10.5. Diagram of the primitive conditional.

A test or condition will be diagrammed as a trapezoid [Exhibit 10.5]. Control enters at the top of this box. Control leaves to the right through the diagonal side (along the horizontal arrow) if the condition is true. It leaves through the bottom (the downward arrow) for a false condition. The arrows are not normally written on the diagram and are shown here and in subsequent diagrams only to emphasize the flow of control.

10.1.6 Control Diagrams

In order to visualize and compare the wide range of control statements in different languages, we need a language-independent representation for each basic kind of control. Flowcharts provide a graphic representation of control flow but cannot represent the difference between GOTO and structured transfers of control. We define a new kind of diagram, called a *control diagram*, that can represent the limited branches inherent in modern control structures.⁶ Simple diagrams are defined for each of the primitive control structures. These diagrams will be elaborated and extended to develop higher-level control structures.

Modern control structures are one-in/one-out units, that is, control can enter the unit at exactly one place and leave the unit at exactly one place. Within a control structure, control may jump around, skip sections, and form repetitive loops.

Our control diagrams represent this control flow implicitly and/or explicitly. Each control diagram has a frame with one entry arrow and one exit arrow. Smaller boxes may be nested within the outer frame. Control flows downward, implicitly, unless an arrow or a branch (indicated by a box section with a diagonal side) directs it elsewhere.

The control diagram for a sequence of statements was shown in Exhibit 10.1. The single entry is indicated by an arrow along the top edge of the sequence box. When control enters this box it progresses downward through any enclosed statements until it reaches the bottom line, then exits on the single exit arrow.

The control diagram for a subroutine call was shown in Exhibit 10.3. When control enters a subroutine call box, the arguments are evaluated and bound to the parameter names; then, control

⁶These diagrams are modeled after, but are not the same as, Nassi and Schneiderman's system.

Exhibit 10.6. Syntax and uses of a conditional expression in C.

Syntax: $\langle \text{condition} \rangle ? \langle \text{expression} \rangle : \langle \text{expression} \rangle$

Two examples of usage:

```
a = b < c ? b : c;           /* Store the smaller value in a. */
printf("%d\n", b < c ? b : c); /* Print whichever is smaller, b or c. */
```

goes to the subroutine code and eventually returns to the bottom of the subroutine call box. Then it leaves by the exit arrow.

Control diagrams for conditionals, loops, and limited control transfers will be presented with these control structures in the remaining sections of this chapter.

10.2 Conditional Control Structures

All programming languages must contain some control structure that permits conditional execution of a section of code or selection among two or more alternative sections of code. This is obviously fundamental to programming: a program that performs the same actions for all inputs is of limited use.

The kinds of control structures included in a language are, in part, determined by whether the language is functional (nested), or a mixture of functional and procedural (sequential). Functional languages contain only expressions—they have no statements, and, therefore, all control structures in these languages are control expressions. Procedural languages all have control statements, and some also include control expressions. Let us explore the difference between control expressions and control statements by looking at conditionals.

10.2.1 Conditional Expressions versus Conditional Statements

The *conditional expression* is the control structure most intimately associated with the functional languages. It is an expression that contains one or more conditions and, for each condition, an expression to evaluate when the condition is true. A final expression is included to evaluate when all the conditions are false. The conditional expression returns, as its result, the value of whichever expression was evaluated. The simplest conditional expression has the general form:

$$\text{if } \langle \text{condition} \rangle \text{ then } \langle \text{expression 1} \rangle \text{ else } \langle \text{expression 2} \rangle$$

For example, the value of the following expression is B or C, whichever is smaller:

$$\text{if } B < C \text{ then } B \text{ else } C;$$

Because a conditional expression returns a result, it may be embedded in a larger expression. Exhibit 10.6 shows the syntax and usage of the conditional expression in C.

Contrast this to the *conditional statement*, which returns no result. A simple conditional statement has the basic form:

Exhibit 10.7. Syntax and uses of a conditional statement in C.

Syntax: `if (<condition>) <statement> else <statement>;`

Usage: these statements are equivalent to the conditional expressions in Exhibit 10.6.

```
if (b<c) a=b; else a=c;          /* Store the smaller value in a. */
if (b<c) printf("%d\n", b); else printf("%d\n", c);
```

`if <condition> then <statement 1> else <statement 2>;`

No result is returned, and the conditional statement is used by including entire statements in its clauses and by placing the entire unit in a sequence of statements. To be useful, the included statements must either do I/O or store their results in a set of variables, as illustrated by Exhibit 10.7.

It is not necessary for a language to provide both a conditional expression and a conditional statement, although some do. Anything that can be written using a conditional expression can also be written (more redundantly) using a conditional statement, statement sequencing, assignment, and a temporary variable, *V*. To do this, the conditional expression is first extracted from its context and its **THEN** and **ELSE** clauses modified to store their results in *V*:

`if <condition> then V=<expression 1> else V=<expression 2>`

Then the outer expression, in which the conditional expression was embedded, is written with *V* substituted for the conditional expression. Exhibit 10.7 shows the result of applying this process to the expressions in Exhibit 10.6.

The opposite transformation is much more complex. Rewriting an entire conditional statement as an expression may not be possible to do mechanically if it contains several statements in the **THEN** and **ELSE** clauses. In the simple case, the transformation is done by “factoring out” the common portion of the **THEN** and **ELSE** clauses. If what remains of the clauses can be written as two expressions, *<Texp>* and *<Fexp>*, and both return the same type result, we can finish the transformation. The common part extracted from the **THEN** and **ELSE** clauses is written once (instead of twice), with a “hole” left where *<Texp>* and *<Fexp>* had been. Then the new conditional expression is formed by writing

`if <original condition> then <Texp> else <Fexp>`

and is embedded in this hole.

The Effect of Conditional Expressions on Programming Style. A Pascal program cannot be written in nested, LISP-like style because Pascal does not support conditional expressions. The **IF** in Pascal is a statement—it does not return a result and therefore may not be used in an argument to a function. The **IF** statement must communicate through the the program environment and not through parameters on the stack. In contrast, the LISP conditional, **COND**, is an *expression* that returns a result which can be used in an enclosing expression. LISP conditionals can, therefore,

communicate with the rest of the program using parameters. It is this difference, not the presence or absence of many parentheses in the syntax, that is the root of the difference between the LISP and Pascal programming styles.

Most languages are not purely sequential or purely nested. LISP is primarily a nested language, but it also contains a `rplaca` (destructive assignment) function and the `progn` control structure. These enable the programmer to use variables and to list a series of expressions to be executed sequentially. Further, all the major sequential languages permit nested function calls, stack allocation, and parameter binding, and some, for example C, contain a conditional expression.

In spite of this extensive overlap in the actual semantic features supported by languages in the two classes, LISP programs tend to be written as nests of function calls, and C programs tend to be written as sequences of statements.

In the late 1970s several people decided that all those parentheses in LISP were “bad”. They wrote *front-end processors* which accepted programs written in a much more Algol-like or C-like syntax and translated them to ordinary LISP. These preprocessors supported arithmetic expressions with infix operators and conditionals that started with IF. But the programs written for these front-ends still used the modular, nested programming style, common with LISP, with little or no use of variables.

On the other hand, consider C. Theoretically a C programmer *could* write using a LISP-like style, but most do not. One can only speculate on the reasons for this. A few possibilities come to mind:

1. The loop control structures in C are useful. C programmers often use arrays and files. Looping statements are a natural way to process these data structures. But loops in C are statements, not functions. If a programmer uses them, he or she cannot write in a purely nested style. In contrast, LISP programmers often use list- and tree-structured data. Recursion, with nesting, is a natural way to process these structures.
2. A programmer using C *expects* to write in a sequential rather than nested style, and a programmer writing in a LISP variant *expects* to write in a nested style. The expected style dominates the program even though either style could be used in either language.
3. Sequential processing may be more natural for humans than highly nested function calls. Breaking the code into statements and storing intermediate results in variables helps the programmer keep things straight in his or her mind.

It is likely that some combination of these reasons works to keep C programmers writing programs that are more heavily procedural than the typical LISP program.

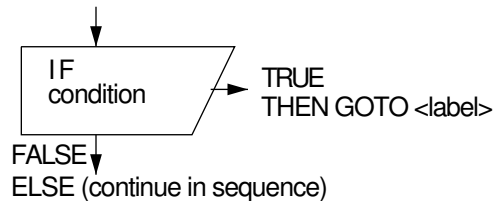
10.2.2 Conditional Branches: Simple Spaghetti

The simplest, most primitive decision structure is the simple **Conditional GOTO**, or IF-GO, diagrammed in Exhibit 10.8. This statement exactly implements the conditional branch instruction

Exhibit 10.8. Syntax and diagram of the simple IF-GO.

Possible source syntax: IF <condition> GOTO <label>; (Keywords vary.)

Diagram :



built into the hardware of most computers, and it was the only conditional control structure in the original BASIC.

The IF-GO is often used to create “spaghetti code”, or code that contains many IF-GO instructions with overlapping scopes that form intertwined threads of code. Spaghetti code is easy to write but hard to debug. It has poor lexical coherence and poor locality of effects. Visually, the spaghetti is the GOTO arrow in the diagram, which may end anywhere in the program.

The IF-GO causes poor lexical coherence because it permits the THEN and ELSE clauses to be placed at widely separated places in the program. Further, if statement labels are distant and not in numeric or alphabetic order, they can be hard to locate in a large program. These problems can, of course, have greater or smaller impact depending on the style of the programmer. The lexical coherence problem is reduced considerably by using the IF-GO to emulate a structured IF...THEN...ELSE [Exhibit 10.9]. The important semantic problems created by use of IF-GO are discussed in Chapter 11, Section 11.1.

10.2.3 Structured Conditionals

Conditional Diagrams. Conditionals will always be diagrammed by boxes (or box sections) with one diagonal side. Normally, control flows through that diagonal side when the condition is true, but this default can be overridden by drawing a horizontal arrow labeled “FALSE” or “F” that crosses the diagonal side. The other condition (normally “FALSE”) causes control to continue flowing downward. Sometimes the diagrams are explicitly labeled for emphasis, even when the default defines the desired direction of control flow.

Structured conditionals are one-in/one-out units, and the corresponding control diagrams have a single entry at the top and a single exit somewhere along the bottom. The conditional is built out of a series of condition clauses, which are rectangles divided in the middle by a diagonal line. The condition is written to the left of the diagonal line, the actions to be performed when the condition

Exhibit 10.9. The IF in BASIC.

Two control skeletons are shown here. On the left side, the **THEN** clause is placed at the end of the program, far distant from the related **IF** and **ELSE** clauses. On the right, the **THEN** clause is placed directly after its related **ELSE** clause. The number of **GOTO** instructions and labels required for both cases is the same, but the right version has shorter spaghetti strands and, therefore, better lexical coherence.

	Poor lexical coherence	Better lexical coherence
	IF <condition> GO 300	IF <condition> GO 100
	<else clause>	<else clause>

200	<rest of program>	GO 200
	...	100 <then clause>
	END	...
300	<then clause>	200 <rest of program>

	GO 200	END

is true are written on the right.

The jump around the **ELSE** clause or clauses is diagrammed by using a *control frame* along the right and bottom edges of the box. Just as control structures replace **GOTO**s, these frames replace long twisting control transfer arrows. Control enters a frame by way of a horizontal arrow after completing a **TRUE** clause, then flows down to the bottom and out by the exit arrow. If control reaches the final **ELSE** clause, it then goes downward into the frame and, again, out at the bottom [Exhibit 10.10].

The Basic Forms. Any of the forms of the structured conditional can be implemented either as a conditional expression (as in **LISP**) or as a conditional statement (as in **Pascal**). They differ from the simple conditional branch statement by being one-in/one-out structures. Contrast the diagram of the conditional branch in Exhibit 10.8 to the diagram of the structured **IF-THEN** conditional on the left in Exhibit 10.10. The difference is that a surrounding frame has become a part of the control structure, delimiting the scope of the **TRUE** clause. The conditional provides two ways to enter this frame, but there is only one way to leave it: downward.

A more general form of this same control structure also includes a scope of actions for the **FALSE** case, giving the full **IF...THEN...ELSE** control structure on the right in Exhibit 10.10. Use of this control diagram is illustrated in Exhibit 10.11.

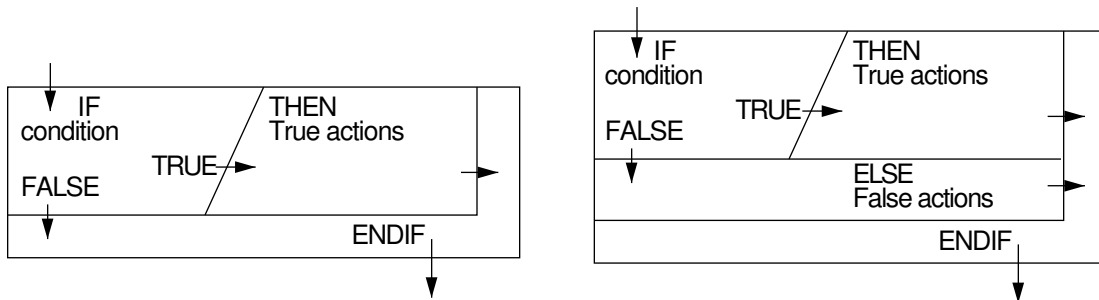
All modern procedural languages provide a conditional control structure with at least two clauses, although the keywords and punctuation vary from language to language. The greatest difference is the way the scopes of the **THEN** and **ELSE** are delimited. There are two possibilities:

Exhibit 10.10. Diagrams of the structured conditional.

In both diagrams, the GOTO of the simple IF-GO has been replaced by a block of statements, to be executed when the condition is true. After executing this block control passes to the right into the frame of the control diagram. If the condition is false, control passes downward into the frame.

In the diagram on the right, a box section is added for the ELSE clause. If the condition is FALSE, control flows into this section, and from there downward into the frame.

This is a one-in/one-out structure. The second exit of the simple conditional branch has been “captured” by the frame of this structure. However control reaches the frame, it leaves the frame by the single downward exit.

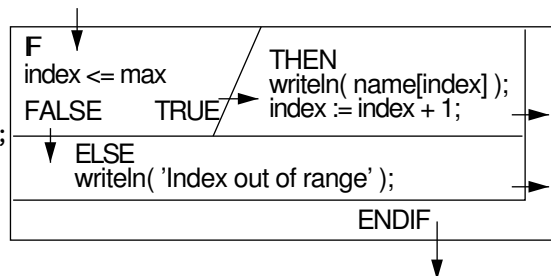
**Exhibit 10.11. A structured conditional statement in Pascal.**

A Pascal conditional statement is shown and diagrammed.

```

if index <= max then begin
    writeln( name[index] );
    index := index + 1 end
else
    writeln('Index out of range');

```



1. The scopes of the `THEN` and `ELSE` are each restricted to a single statement. If more than one action is required, a compound statement may be used. This is a series of statements delimited by `BEGIN...END` or equivalent bracketing symbols. This kind of syntax is used in Pascal and C.
2. An explicit `ENDIF` is provided. The series of statements between the `THEN` and the `ENDIF` (or the optional `ELSE`) is the `TRUE`-scope. The series of statements between the `ELSE` and the `ENDIF` is the `FALSE`-scope. This syntax is used in FORTRAN, Ada, and Turing.

The advantage of choice 1 is that it has fewer keywords and the language syntax is simpler. A disadvantage is that the syntax is ambiguous if one `IF` is nested inside another, and only one has an `ELSE` clause. An extra rule in the grammar forces this lone `ELSE` to be parsed with the nearer `IF`.

The advantages of choice 2 are that the program does not become filled with `BEGIN...END` pairs which can interfere with indenting and can be a source of clutter. Also, the ambiguity problem for nested conditionals is solved by the `ENDIF`. If `ENDIF` is placed before the lone `ELSE`, that `ELSE` parses with the outer `IF`; otherwise, it goes with the inner `IF`.

Finally, this `IF...THEN...ELSE` control structure can be extended again to permit a series of conditions to be tested [Exhibit 10.12]. This produces the most general and flexible possible conditional control structure. Exhibit 10.13 shows a use of the general conditional in LISP. In other languages, conditional clauses after the first are often denoted by the keyword `ELSIF` or `ELSEIF`, as shown in Exhibit 10.14.

Some languages support the `IF...THEN...ELSE` but not the `ELSEIF` clause. To write a generalized conditional in these languages, one writes a series of conditionals, nesting each successive `IF` inside the prior `ELSE` clause. The Pascal code fragment in Exhibit 10.15 does the same thing as the LISP and Ada versions in Exhibits 10.13 and 10.14. Note the similarity to Ada code. The same Pascal statement is shown indented in two ways. The version on the left shows the actual nesting depth of the `IF`s. The version on the right reflects the semantics of the situation, a choice among several parallel cases.

A Short History of Conditionals. At about the same time as the simple `IF...THEN...ELSE` was introduced in ALGOL-60, the generalized conditional came into use in MAD⁷ and in LISP.⁸ The generalized form was later included in PL/1⁹ but then was omitted from many new languages (e.g., C and Pascal) which instead included the simpler `IF...THEN...ELSE` from ALGOL-60.

The original FORTRAN did not include any structured conditionals at all. All forms of `IF` used `GOTO`s to transfer control. The ANSI committee in charge of revising the FORTRAN standard

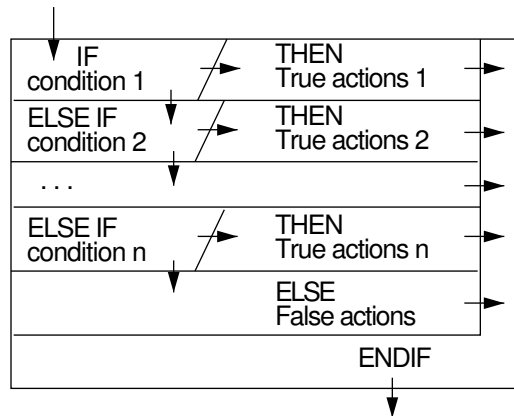
⁷Michigan Algorithm Decoder, Arden, Galler, and Graham [1963]. This language was based on the preliminary ALGOL report (1958) and implemented at the University of Michigan in about 1959.

⁸LISt Processing language, McCarthy et al. [1962]. This language was implemented at M.I.T. in the late 1950s.

⁹Programming Language 1, implemented by IBM in 1967 and intended to be the language that would fill everyone's needs. B. Galler, one of the designers of MAD, was on the committee that designed PL/1.

Exhibit 10.12. Diagram of the generalized structured conditional.

This is a one-in/one-out structure. Any number of conditions may be listed in series, each associated with a scope of actions to be executed if that condition is TRUE. The conditions are tested in the order written, and the actions for the first true condition are executed. If no condition is true, the FALSE scope at the bottom is executed. After executing a scope, control passes into the frame.



resisted all urging to include a structured conditional in FORTRAN II and even in FORTRAN IV. Many nonstandard extensions of FORTRAN IV included IF...THEN...ELSE...ENDIF, though, and finally, with FORTRAN 77, the full structured IF with ELSEIF was included because of popular demand. This illustrates both the trouble with design-by-committee, and the slowness with which good new ideas are accepted by people accustomed to doing things in other ways.

The syntax of a language is certainly simplified (and translation is therefore made easier) by omitting the ELSEIF. The ELSEIF adds no “power” to the language, since the same flow of control can be created using the simple IF...THEN...ELSE control structure where the scope of each ELSE is another IF statement.

Opinions about what is most important in language design vary from community to community

Exhibit 10.13. Use of the general conditional expression cond in LISP.

This call on the print function takes one piece of information, an age, and prints a word describing that age. Its control diagram is the same as the diagram in Exhibit 10.14.

```

(print (cond ((< age 5) ("young child"))
           ((< age 13) ("child"))
           ((< age 18) ("adolescent"))
           (T ("adult"))))
  
```

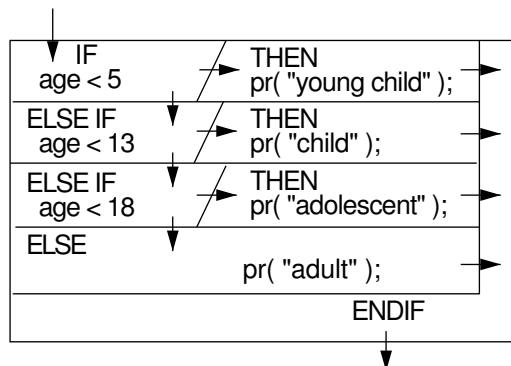
Exhibit 10.14. Use of the general conditional statement in Ada.

I/O in Ada is too complex to be explained here. Please simply assume that the user has defined a procedure named “pr” that takes one string argument, and prints it, followed by a carriage return. This code fragment then does the same thing as the LISP version in Exhibit 10.13. Its diagram is on the right.

```

IF age < 5 THEN
  pr("young child");
ELSIF age < 13 THEN
  pr("child");
ELSIF age < 18 THEN
  pr("adolescent");
ELSE
  pr("adult");
END IF;

```

**Exhibit 10.15. Using Pascal to emulate ELSEIF.**

Indentation shows nesting depth.
Each IF is within the prior ELSE.

```

IF age < 5 THEN
  writeln('young child')
ELSE IF age < 13 THEN
  writeln('child')
  ELSE IF age < 18 THEN
    writeln('adolescent')
  ELSE
    writeln('adult');

```

Left-justified ELSE IF shows semantic structure—a series of equal choices.

```

IF age < 5 THEN
  writeln('young child')
ELSE IF age < 13 THEN
  writeln('child')
ELSE IF age < 18 THEN
  writeln('adolescent')
ELSE
  writeln('adult');

```

and change often. In this case, there is no agreement whether a “good” language should include an `ELSEIF` clause. The `ELSEIF` statement has some nice properties:

- It is a good reflection of the semantics of a series of parallel choices, a commonly occurring control pattern.
- It combines well with the `ENDIF` to delimit a series of conditional clauses without need for `BEGIN...END` bracketing, which clutters the code.
- A series of `IF`s has each `IF` nested within the `ELSE` clause of the preceding `IF`. This causes many programmers to indent each `IF` more than the prior one, as shown in the left column of Exhibit 10.15. In contrast, the `ELSEIF` statement encourages programmers to indent scopes uniformly, in a manner consistent with their status as equally important alternatives.

Thus factors relating to human engineering are the primary reasons for including the more complex `ELSEIF` syntax in a language design. It is not theoretically necessary or even helpful.

10.2.4 The Case Statement

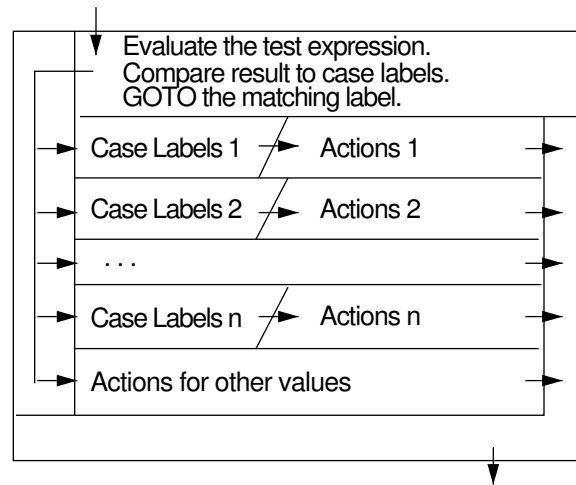
While the generalized conditional is maximally flexible, it is not always maximally efficient or convenient to write. Consider the situation in which one of a set of N actions must be taken depending on the value of a single expression [Exhibit 10.13]. It is redundant to write that expression $N - 1$ times as part of N conditions, and it is inefficient to evaluate the expression $N - 1$ times, especially if it is a long expression. Some efficiency can be gained by evaluating the test expression once and storing the answer in a variable, but that still requires fetching that value $N - 1$ times. In assembly language, a programmer would write only one fetch.

This need was met in nonstructured procedural languages by the *computed GOTO statement*. APL, COBOL, and FORTRAN all contain versions of a computed GOTO. In this control structure, an expression is evaluated and its result is used to select one statement label from a list of labels. Control then goes to that label.

Essentials of the CASE Statement. What is needed here is a structured conditional control statement with multiple branches that implements this common control pattern in an efficient way. The single expression can be evaluated once at the top of the control structure and its value loaded into a machine register where it can then be used to index a transfer vector or be compared efficiently to a series of constants to determine which set of actions to execute. This general **CASE** structure is illustrated in Exhibit 10.16.

This is the control structure implemented by the **CASE** in Ada. It is a complex control structure whose semantics are defined by the following list of rules:

1. Each set of case labels is a series of constants or constant expressions of the same discrete type as the result of the test expression.

Exhibit 10.16. The CASE statement.

2. The actions executed are the ones following the case label that is equal to the value of the test expression. The same constant may not be included in two sets of labels.
3. If no label equals the test value, the “other values” clause is executed.
4. After executing a set of actions, control passes to the right, into the “frame”, and from there to the single exit.

Possible variations of the general **CASE** structure involve the rules for the type of value returned by the test expression and the rules for labeling cases. The value must, in any case, be a discrete value. Real numbers such as 3.91 would not make meaningful case labels because of the approximate nature of real arithmetic.

Some languages require that every possible value of the test expression occur exactly once as a case label. If a type with a large number of possible values, such as integer, is permitted, some way must be provided to avoid listing all possible values. This might be done by using subranges of the type as case labels or by using an **OTHERS** clause to cover nonexceptional cases.

There are three good implementations of the **CASE** structure. In all three, the test expression is evaluated once at the top and loaded into a machine register. An easy implementation is possible if the machine instruction set includes a **computed goto** instruction, and the range of values of the control expression is small and matches the range that the hardware is built to handle.

The second implementation, a transfer vector, is exceptionally fast but possibly requires a lot of space. It uses the register containing the test expression value to index an array of code addresses. The compiler generates this array by first sorting the case labels into ascending order, then storing the address of the object code for a case action in the array position corresponding to the value

of its case label. To execute case N , the translator generates a simple indirect GOTO through the N th position of the label array. With this implementation, every possible value in the type of the test expression must occur as a case label, and the transfer array will be as long as the number of possible case labels.

A third implementation of the case is possible if one of the above conditions does not hold. If a possible case label may be omitted or a very large type (like integer) is permitted for the case labels, the CASE must be translated into a series of conditional GOTOs. A series of case labels can be compared very efficiently to the register containing the test value, and a GOTO taken to the first label matching the value in the register. This is not as fast as use of a transfer vector, but it is a big improvement over the code that would be generated (by a nonoptimizing compiler) for a series of IF statements.

Defective Versions of the CASE Statement.

COBOL's CASE. The CASE statement was introduced into COBOL a long time ago when the programming community did not understand that GOTOs cause unending trouble. Unlike the modern CASE, the COBOL CASE is a multiway conditional GOTO, and a program containing a CASE becomes one massive nest of GOTOs. Procedure calls and GOTOs interact in a curious way in COBOL; the same block of code can be entered either way. If a block is entered by a procedure call, control returns to the caller at block end. But if it is entered by a GOTO, control continues on to the next lexical program element. Using the CASE in such an environment makes a program even harder to debug than it would be in FORTRAN or in BASIC. Control can shoot off in any direction if the programmer is not careful.

A good CASE statement is particularly useful in modern data processing applications where on-line data entry is expedited by menu-driven programs. The general CASE is ideal for menu handling. COBOL 85¹⁰ introduced a new statement type, called EVALUATE which implements the general case structure [Exhibit 10.17]. It is an especially nice version of the general control structure because it allows range expressions to be used as case labels.

The introduction of the EVALUATE statement in COBOL fixes an old defect and illustrates the way languages can grow and adapt to an increasingly sophisticated public. It also illustrates one negative aspect of language growth and restandardization: the proliferation of keywords. Rather than redefine the keyword CASE to have a modern semantics, a new keyword was introduced. The standardization committee had no choice about introducing a new keyword: if they redefined the old one, *all existing programs* that used CASE would become obsolete overnight. In the business world this is certainly unacceptable. Many companies have programs in use that were written ten years ago and are simply modified every time the company's needs change. Reworking all those programs to eliminate the CASE GOTOs is not reasonable. The old CASE statement must now be categorized as an archaic feature that should never be used. Perhaps in twenty years it can be dropped from the language.

¹⁰A good presentation of the changes in the language can be found in Stern and Stern [1988].

Exhibit 10.17. The COBOL EVALUATE statement.

General Syntax	Code Sample
EVALUATE <expression>	EVALUATE EXAM-SCORE
WHEN <condition-1> <action-1>	WHEN 90 THRU 100 PERFORM A-ROUTINE
WHEN <condition-2> <action-2>	WHEN 80 THRU 89 PERFORM B-ROUTINE
...	...
WHEN <condition-n> <action-n>	WHEN 0 THRU 59 PERFORM F-ROUTINE
WHEN OTHER <action-other>	WHEN OTHER PERFORM ERROR-ROUTINE
END EVALUATE	END EVALUATE

Pascal's **CASE**. Perhaps because the **CASE** statement is inherently complex, some languages contain variants of it that can be considered to be faulty. That is, they fail to include some important part of the general structure.

Experience has shown that a **CASE** conditional structure needs to include an **OTHERS** clause which is to be executed if none of the case labels matches the test value. For example, a very common application for a **CASE** statement is to handle character data. In such situations, it is often desirable to provide special handling for a few special characters, and one or two routines to be executed for most of the other possible characters. The potential efficiency of a **CASE** statement makes it attractive in implementing this situation. With an **OTHERS** clause, this structure requires writing out case labels only for cases that require special handling. But without an **OTHERS** clause, case labels would have to be written and tested for all 128 possible ASCII characters. This is messy and cumbersome, and most programmers will end up using a series of half a dozen **IFs** instead of writing a **CASE** statement with 128 case labels.

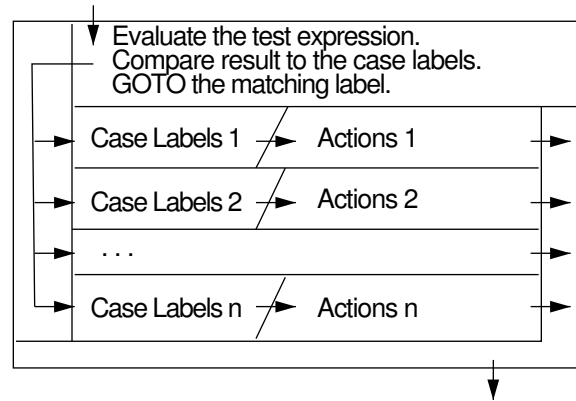
Standard Pascal fails to provide an **OTHERS** clause for its **CASE** statement. It has the logical structure shown in Exhibit 10.18. Because it lacks this clause, the Pascal **CASE** statement is rarely appropriate for real situations. Many Pascal implementations are extended to contain an **OTHERS** clause. Unfortunately, there is no single standard way to add such an extension, and the extensions often have slight syntactic differences, making a program containing a **CASE** nonportable.

If the value of the test expression is *not* included among the case labels, it is an "error" according to the ISO Pascal standard. (That is, it is a violation of strict Pascal semantics.) In this situation the person who builds a translator has several choices:

- Detect the error at compile time and give an error comment.
- Detect the error at run time and halt execution.
- Make clear in the documentation that the error will not be detected.

Handling a case error in the third way makes it legal (by the standard) to implement one sensible default: if no case label matches, do nothing. Occasionally this is even what the programmer wants

Exhibit 10.18. The Pascal CASE with no default clause.



to do. But failing to detect a language error is just not a good way to get around the defects in the language. Ideally, this defect will be corrected in the next Pascal standard.

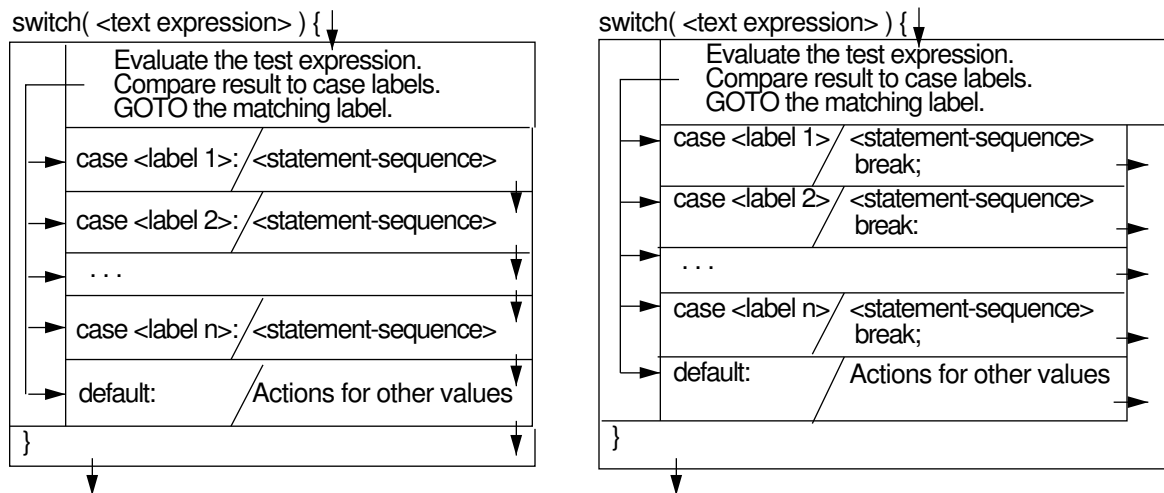
C's switch Statement. The `switch` statement in C does have an `OTHERS` clause, marked by the keyword `default`. However, C omits a different part of the general `CASE` control structure: the exit into the frame after completion of the `CASE` actions. The general `CASE` is like a series of simple conditionals, each with an accompanying scope. The C `switch` is not. It is like a `computed GOTO`, where all target statements are within the scope of the `switch` statement. Restricting the targets to the scope of the `switch` is a great improvement over the semantics of a `computed GOTO`, since it enforces locality of effects and lexical coherence. Nonetheless, the C `switch` is less than a true `CASE` structure.

The `switch` differs from the general `CASE` structure because once C "goes" to a case label and completes its corresponding actions, it keeps on "going", doing, in turn, all the actions for the succeeding cases! [See Exhibit 10.19, first column.] This is rarely, if ever, what the programmer wants. The `switch` lacks the frame on the right which is part of the `CASE` structure.

The C `switch` would be unusable except for the existence of the `break` statement, which can be used to attach a frame to any C control structure. Executing a `break` statement sends control into the frame and thus immediately terminates the `switch` statement in which it is embedded. Thus the `break` is like a structured `GOTO out of` the control structure.

C forces the programmer to explicitly write the `break` which should be an integral part of the `CASE` control structure. This does provide some flexibility. The programmer may choose to leave the control structure (`break`) or continue with the actions for the other cases. But this is a case of too much, error prone flexibility. It is a very rare situation in which one would want to continue with all the rest of the case actions.

One can only guess at the reasoning of the C designers when they defined the `switch` this

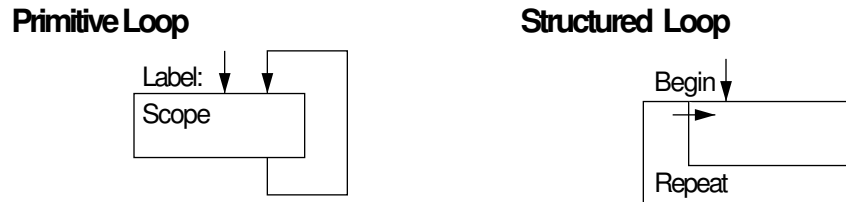
Exhibit 10.19. The C switch statement without and with break.

way. Since the `break` exists, all the internal mechanisms must be included in the translator to implement the general CASE structure. The reasoning that the combination of `switch` and `break` is more flexible than the general CASE is specious, since an added control command “`don't-break`” could just as well be combined with the CASE to produce the same flexibility as `switch` plus `break`, and be less error prone in the common case. Perhaps this strangeness is simply an accident of history, and a reflection of the fact that C is a twenty-year-old language. In any case, by the principle of *Too Much Flexibility*, we can say that it is a design error.

10.3 Iteration

Iterative control structures are used to process homogeneous aggregate data structures such as arrays, lists, and files. In procedural languages, iteration is accomplished through explicit loop statements which do not return values although they may modify variables or arrays in memory. In functional languages, iteration is done through functions in which repetition is implicit, such as the LISP `map` functions. These often return lists that were lengthened by one element on each iteration.

Diagramming Loops. Iterative control structures are one-in/one-out units, and the corresponding control diagrams still must have a single entry at the top and a single exit somewhere along the bottom. The nonsequential transfers of control in a structured loop are diagrammed by using a *control frame* around parts of the box. Control enters the frame by “falling through” from the bottom of the scope, then continues in a clockwise direction, to the left across the bottom, and

Exhibit 10.20. The simplest loop: an infinite loop.


upward on the left. Control reenters the scope along a horizontal arrow, near the upper left, that leads into some enclosed box section. Control leaves the loop structure at the bottom along a vertical exit arrow.

10.3.1 The Infinite Loop

The most primitive loop is formed by placing a statement label at the top of the desired loop scope and writing a branch to that label at the bottom of the scope. Its diagram is, therefore, formed by joining the diagrams of a labeled sequence of statements to the diagram of a `GOTO` [Exhibit 10.20, left]. Far from being useless, infinite loops are now used extensively to implement processes that interact by using the I/O interrupt system. This kind of infinite loop, though, formed by an unconstrained `GOTO`, has been replaced in modern languages by the structured infinite loop [Exhibit 10.20, right].

In the structured form, the loop scope is delimited by keywords (`Begin` and `Repeat` are used here), and the branch statement is generated by the translator. As shown in Exhibit 10.20, this is a one-in/*zero*-out structure! The loop scope is framed on the bottom and left, and the only exit from the frame is back into the scope at the top. When used with an `exit` statement, the exit adds a frame and exit arrow on the right. The result is equivalent to the diagram in Exhibit 10.25.

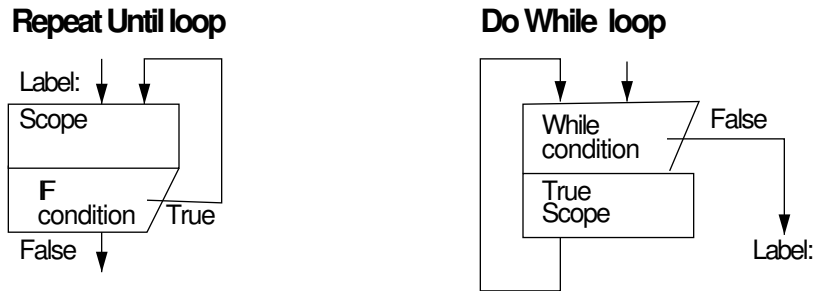
10.3.2 Conditional Loops

Two different simple conditional loops can be formed with a single conditional `GOTO`. The first form is the same as the primitive infinite loop except that the `GOTO` at the bottom of the scope is a conditional `GOTO` [Exhibit 10.21, left]. This is frequently called a `REPEAT LOOP`.

In the second form, usually called a `WHILE LOOP`, the conditional branch is placed at the top of the scope [Exhibit 10.21, right]. This form is more frequently useful. Loops are used to process files, lists, and arrays, and it is usually necessary to prepare for the possibility of empty files and zero-length data structures.

Corresponding to the two kinds of conditional `GOTO` loops are the two structured loops shown in Exhibit 10.22. Their translations into machine code will be the same, but the structured form automatically generates a semantically correct branch instruction. In the structured loop, the

Exhibit 10.21. The simplest finite loops, formed with a GOTO.



programmer marks the beginning and end of the loop scope with the exit test and a keyword or special character. The compiler then generates the GOTO that forms the loop.

A WHILE loop has the exit test at the top of the scope, a REPEAT loop at the bottom. The WHILE loop was one of the control structures recommended by Dijkstra to be used in eliminating GOTOs. Structured loops are easier and faster to translate than the GOTO versions,¹¹ and they are less error prone in the hands of any programmer.

The REPEAT is not a very useful loop form because the scope is always executed at least once. In almost all situations some case can arise in which the scope should be skipped. Often these cases correspond to faulty input data. The WHILE makes it easier to check for these errors. For example, the REPEAT loop in Exhibit 10.23 will malfunction if the input, *n*, is less than 1.

With either of these loop forms, the loop test may have a positive or a negative sense; that is, it may loop on TRUE and exit on FALSE, or loop on FALSE and exit on TRUE. The former type is called a *While test* and the latter type is called an *Until test*. This is a detail that must be checked when

¹¹See Chapter 11, Section 11.1.

Exhibit 10.22. Structured conditional loops.

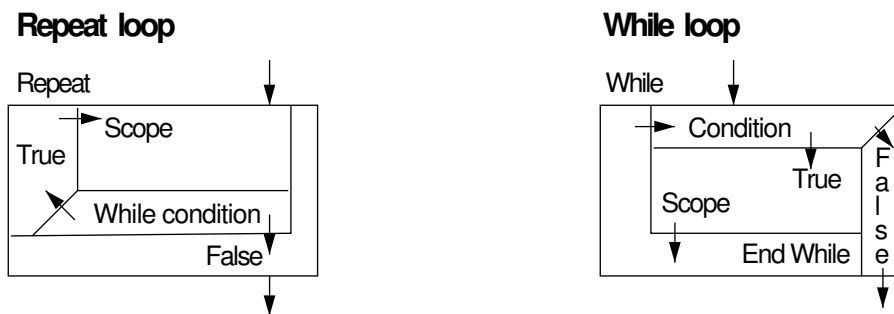


Exhibit 10.23. REPEAT and WHILE loops in Pascal.

These loops both print out the first n elements of the array a . The repeat loop will malfunction unless $n \geq 1$.

Repeat loop:	While loop:
Readln(n);	Readln(n);
i:= 1;	i:= 1;
Repeat	While i<=n Do Begin
writeln(i, a[i]);	writeln(i, a[i]);
i := i + 1	i := i + 1
Until i=n;	End;

learning a new language. Exhibit 10.23 shows samples of the REPEAT and WHILE loops in Pascal. Note that the REPEAT loop ends with an *Until* test.

The WHILE loop is an appropriate basis for an iterative expression as well as for an iterative statement. It is built into LISP's family of `map` functions and into iterative constructs in modern functional languages such as Miranda. This topic is taken up more fully in Section 10.4.

10.3.3 The General Loop

Restricting loops to a single exit at either the top or the bottom of the scope is artificial. It leads to awkward programming practices such as the “priming read”—a read instruction given to bring in the first line of data before entering the main processing loop [Exhibit 11.5]. The main loop then must end with a duplicate of that read instruction. There is no reason why a loop should not have its exit test at any arbitrary place within its scope, as in the FORTH program in Exhibit 10.24.

Let us define the *general loop* [Exhibit 10.25] to be a structured loop that merges the two common loop forms (REPEAT and WHILE) into a single more flexible looping form. The loop termination test in the general loop is not constrained to occur at the beginning or end of the loop. The beginning and end of the scope will be bracketed by the keywords LOOP and END LOOP, and statements of the scope may come both before and after the termination test. The termination test could be either a While test or an Until test, or the language might provide both forms, giving the programmer an option.

This very nice loop form is supported in FORTH, Turing, and Ada [Exhibit 10.26]. It is likely to become more widely used since REPEAT and WHILE loops are simply special cases of this general structure, and language translation is simplified by reducing the number of special cases. Providing the extra flexibility costs little or nothing.

Exhibit 10.24. The general loop in FORTH 83.

Assume `getn` has been defined to be a function that reads an integer from the keyboard and leaves it on the stack.

```
VARIABLE sum
: read&add          ( Print a prompt and read a value. )
                   ( If it is positive, sum it and repeat, else quit. )
  0 sum !          ( Initialize sum to zero. )
  BEGIN
    ." Please enter a number and type a CR."
    getn 0 >      ( True if input is a positive number. )
    WHILE        ( Exit loop if prior expression is not True. )
      sum @ + sum ! ( Fetch sum, add input, store back in sum. )
    REPEAT
      sum @ .     ( Fetch value of sum and print it. )
  ;              ( End of function definition. )
```

10.3.4 Counted Loops

Finally, we can add some mechanism to any of the loop forms (`REPEAT`, `WHILE`, or `LOOP`) to automate sequential processing of a data structure whose length is known. We call the result a *counted loop* because the number of iterations is fixed at the beginning of the loop, and the loop exits when the iteration count reaches that number. Counted loops differ from general loops because the number of repetitions of a general loop may depend on the result of computations within the loop.

There are nearly as many variations in the details of the counted loop as there are languages that support it. In general, though, a variable, called the *loop variable*, must somehow be defined. It is initialized at the top of the loop, incremented at the bottom of the loop, and tested by a loop test. Within the scope of the loop, the value of the loop variable is available and is often used as

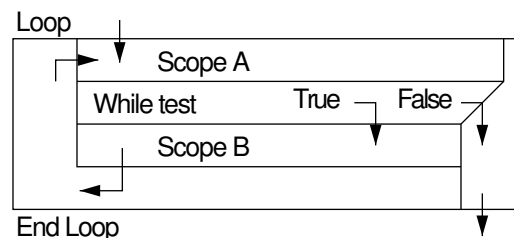
Exhibit 10.25. The general loop.

Exhibit 10.26. The general loop in Ada.

This function does the same thing as the FORTH function in Exhibit 10.24. Assume that `write_prompt` is a programmer-defined procedure that writes a message to the screen, and `get_number` is a function that reads an integer from the keyboard and returns it.

```

FUNCTION read_add RETURN integer IS
  n: integer;
  sum: integer := 0          -- Initialize sum to zero.
  LOOP
    write_prompt("Please enter a number and type a CR.");
    n:= get_number ;
    EXIT WHEN n<=0 ;
    sum := sum + n;
  END LOOP;
  return sum;
END read_add;

```

a subscript. When the value of the loop variable exceeds the preset limit, the loop is terminated. An additional “clean-up” step may or may not be done at loop exit time.

Exhibit 10.27 gives a diagram of a typical counted loop with the loop test at the top, like a WHILE loop. This form is implemented in several languages, including Pascal, COBOL, FORTH, and BASIC.¹²

The nature and implementation of the loop variable has evolved through the years. FORTRAN and COBOL had only static storage, and loop variables were ordinary global variables. PL/1, an old language, permits the loop variable to be declared in any scope surrounding the loop. As the years passed, the special nature of the loop variable was better recognized, and various special restrictions were applied to it.

Pascal, for example, requires that the loop variable be declared locally in the block that contains the counted loop. Some languages restrict the use of the loop variable. Pascal and all versions of FORTRAN say that the programmer must not assign a value to the loop variable within the loop. This is done for efficiency. During execution of the loop, the loop variable can be temporarily implemented by a machine register, instead of or in addition to an ordinary memory location. This makes it very fast to test, increment, and use the loop variable. The typical loop refers to its loop variable many times. Thus many machine cycles are saved by not fetching and storing this value every time it is referenced or updated.

FORTH *predeclares* a local identifier, I, to name the loop variable. (If two loops are nested, I is the index for the inner one and J for the outer one.) The loop variable is kept on top of the return

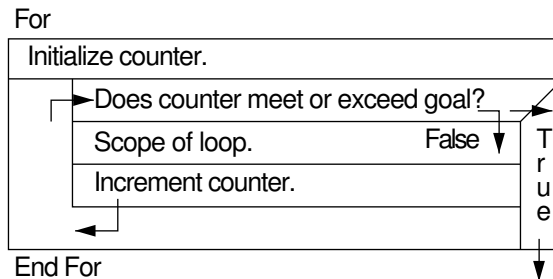
¹²The `for` loop in C is not a counted loop by our definition; it is dealt with in the next section.

Exhibit 10.27. The counted loop.**Syntax in various languages**

```

FORTH:
    10 0 DO <scope> LOOP
Pascal:
    For J:=1 to 10 Do Begin
        <statement sequence>
    End;
BASIC:
    FOR J=1 TO 10
        <sequence of lines>
    NEXT J
COBOL:
    PERFORM <paragraph-name>
        VARYING J FROM 1 BY 1
        UNTIL J > 10.

```



stack, rather than in an ordinary storage object, and it is deallocated automatically at loop exit.

Two newer languages, **Ada** and **Turing**, go further in this direction. They open a local declaration scope at the beginning of the counted loop and automatically declare the loop variable name as a local variable in that scope. The loop variable is used within the loop and deallocated when the loop terminates. This arrangement has one real advantage: a loop variable, by its nature, has only local meaning, and that meaning is obvious from the context. (This is reflected in the fact that programmers still like to use single-letter names for loop variables.) Declaring the loop variable as a local variable within the loop increases the modularity of the program and minimizes chances of accidental error.

A question now arises about the meaning of the loop variable after exit from the loop. If the loop variable was local to the loop, the answer is clear and easy to implement: since the loop variable was deallocated at loop exit time, its value is undefined. That means that it is completely implementation-dependent, and the programmer should never attempt to use the loop variable after a loop exit. This makes it easy for a compiler writer to implement the language—she or he can use a register for the loop variable, to gain execution efficiency, and does not need to store the final value back into a memory location.

In most languages the loop variable is not local to the loop. However, there is still a question about its meaning after loop exit. Does it contain its original value? Its value on the final time through the loop? A value that is one greater than the goal? In **FORTRAN II**, the loop variable was kept in a machine index register and its value was officially undefined after loop exit. The **ISO Pascal** standard also states that the value is *undefined* after loop exit.

This design decision has been called a “victory of religious fervor over common sense”.¹³ It is easy to define in a formal semantic definition and easy to implement efficiently. However, the programmer often needs to know the value of the index after an abortive loop exit. If that information is not available, the programmer is forced to introduce yet another variable and assign the index value to it inside the loop. This is surely neither efficient nor desirable.

FORTRAN 77 illustrates a more practical design. The language standard defines what the index value will be after an exit. The index value may still be stored in a machine register during the loop, but if so, an *epilogue*, or clean-up step, must be executed before leaving the loop normally or by a `GOTO`. The epilogue will copy the current value of the loop variable back into its ordinary memory location. The loop variable may, thus, have either of two values after exit. If the loop exited normally after completing N iterations, the loop variable equals the initial value plus N times the step size. (This is at least one greater than the goal value). If a `GOTO` is used to leave the loop, the loop variable will contain the value current at the time of exit.

The counted loop requires several arguments: the size and sign of the increment, the initial and terminal values for the loop variable, the beginning and end of the scope to be iterated, and the exact termination condition. Different languages provide more or less flexibility in specifying these things and have different rules about the times at which each element is evaluated. Some ways in which counted loop statements differ from language to language are trivial syntactic details; others are more major semantic differences that determine the power and flexibility of the control structure.

FORTRAN II is an example of a language that has differed in a major semantic way from most. It placed the execution of the scope before the loop test so that every loop was executed at least once. That is now recognized to be a mistake, and all modern languages place the test at the top, before the loop scope.

In all languages, control must enter a counted loop through the initialization code (you cannot `GOTO` into a counted loop). But there are differences in the rules about what happens at loop entry and loop exit, and the ways that control is allowed leave a loop. Further, treatment of the loop variable, increment, and goal expressions can be different.

All languages increment or decrement the loop variable after executing the scope and before retesting the exit condition. Most of the time, programmers write loops that increment or decrement the value of a variable by 1, so Pascal and Ada restrict the increment to be either +1 or -1. This shortens the language syntax a little but is an unnecessary and annoying restriction.

Some languages exit when the loop variable meets the goal, some when the goal is exceeded, and some permit the programmer to write his or her own test using relational operators. Among the latter group, some exit when the test expression evaluates to `TRUE` and others when it evaluates to `FALSE`.

If the language allows the programmer to write an expression (rather than a constant) as the goal, it must also specify when that expression will be evaluated. The original FORTRAN, designed for efficiency, permitted the goal to be a constant or simple variable name. The value was fetched

¹³B. Kernighan, informal comment.

Exhibit 10.28. Pascal computes a tripcount.

Assume *i* and *k* have been declared to be local integer variables in the program that contains this code fragment. It is not clear from looking at this loop whether it will produce three lines of output or an infinite number. To know the answer you must know that Pascal *does* compute a tripcount before entering the loop.

```

k := 3;
FOR i := 1 TO k DO BEGIN    { A tripcount of three is computed here. }
    writeln(i, k);
    k := k + 1              { Changing k does not change the tripcount. }
END;
writeln(i);                { Answer is unpredictable; i is undefined. }

```

once and loaded into a register. Changing the value of the goal variable within the loop did not affect loop termination. Modern FORTRAN permits the goal to be specified by an expression, but the meaning remains the same: it is evaluated only once, before entering the loop. At that time, the *tripcount*, or number of loop repetitions, is fixed. Changes to values of variables within the loop cannot affect the tripcount.

This restriction does not apply to the PL/1 FOR loop. Its goal is specified by an arbitrary expression that is reevaluated every time around the loop. If the values of some of its variables change, the loop termination condition changes. This is also true in COBOL.

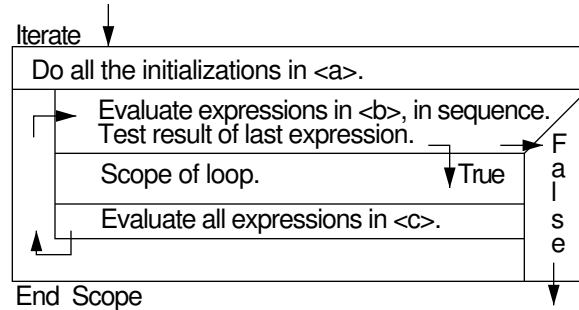
Three things must be fixed before loop entry in order to determine a tripcount: the initial value of the loop variable, the goal value, and the increment value. If any one of these can be changed within the loop, the number of iterations is unpredictable. In Pascal, all three are fixed. The increment value is always +1 or -1. The language standard states that the loop variable may not be modified within the loop. The goal expression is evaluated only once, at the top of the loop. These three together determine the tripcount [Exhibit 10.28].

A small change in the definition of the semantics of a counted loop can radically affect its behavior. If either the goal value or the increment value can be changed inside the loop, the tripcount cannot be determined ahead of time. FORTH has two counted loop forms, DO...LOOP and DO...+LOOP. In both of these, the beginning and ending index values are computed before the loop. The DO...LOOP has a constant increment value of +1, and so has a fixed tripcount. But the DO...+LOOP uses whatever value is left on the top of the stack as the increment value, producing completely unpredictable behavior.

Clearly, in order to understand the semantics of the counted loop in any particular language, all of these matters must be clarified. The counted loop is thus a very interesting statement type. While its general intent is obvious, knowing its actual meaning requires knowledge of many small details. Programmers changing to a language with different details make lots of errors writing counted loops.

Exhibit 10.29. The iteration element.

Syntax: Iterate (<a>;;<c>) <scope>;



10.3.5 The Iteration Element

The most general form of the FOR loop is called the *iteration element*.¹⁴ It was developed in 1963 and was incorporated into the MAD language at that time. Dennis Ritchie, who used MAD at Harvard, later incorporated it into his own language, C. The iteration element generalizes the semantics of each part of the FOR statement and is equally implementable as a loop statement and as a loop expression. It has two sections: a control element and a scope, diagrammed in Exhibit 10.29. The control element contains:

- a. A list of arbitrary expressions to be evaluated before entering the loop (initializations). Control passes to expression (b) after executing these expressions.
- b. An arbitrary expression whose result must be interpretable as a truth value. This is evaluated before executing the loop scope. Iteration continues until this value is **FALSE**. At that point control passes out of the iteration element.
- c. A list of expressions to be executed after the scope of the loop (increments). Control passes to expression (b) after executing these expressions.

The scope begins at the end of the control element. In an iteration statement, the scope contains a single statement or a compound statement, consisting of a sequence of statements delimited by begin-scope and end-scope symbols. Executing the scope results in a side effect such as input, output, or assignment. In an iteration expression, a program object is computed each time the scope is executed. These program objects are collected into an aggregate (a list or an array) that becomes the result of the entire iteration expression.

¹⁴Galler and Fischer [1965].

Exhibit 10.30. Uses of the iteration element in C.

```

/* Read a file of subscripts and print corresponding array elements. */
for( ; scanf("%d",&k)!=EOF; )
    if (k>=0 && k<=max) printf("%d\n", ar[k]);

/* Read N, then read and sum N numbers. */
for (sum=count=0, scanf("%d",&N); count<N; count++)
    { scanf("%d", &k); sum+=k;}

/* Sum the numbers in a linked list. */
for (current=list, sum=0;
     current!=NULL;
     sum+=current->value, current=current->next);

```

The notation `current->value` on the last line means to dereference the pointer `current` and select the `value` field of the resulting cell. It is equivalent, in C, to `(*current).value`.

Iteration in C. The C `for` statement implements the iteration element. The syntax for `for` is:

```
for(<expression>; <expression>; <expression>) <statement>
```

Exhibit 10.30 shows examples of the use of the `for`. For these loops, assume identifiers have been declared as follows:

- A cell has two fields, named `value` and `next`.
- `current` and `list` are pointers to cells.
- `k` and `N` are integer variables.
- `ar` is an array of integers with subscripts `0...max`.
- `scanf` stores the data in the address given as an argument and returns an error/success code which will equal EOF when end-of-file is found.

The biggest difference between the iteration element and ordinary FOR loops is the totally unrestricted nature of the pieces of the control element. This leads to all sorts of unusual constructs, like the last loop in Exhibit 10.30 in which *all* of the work of the loop is done by the control element, and the scope is empty.

When using a normal FOR statement, there is generally one way to set up the initialization, goal, increment, and scope to achieve the desired result. When using the iteration element, there can be several ways to do the same thing [Exhibit 10.31]. This flexibility confuses programmers who are accustomed to restricted forms of the FOR. Many want to be told what the *right* way is, but, of course, there is no single right way. At best we can give rough guidance:

Exhibit 10.31. Many ways to write the same iteration.

These `for` loops do the same thing as the last loop in Exhibit 10.30. (Note: When two C expressions are connected by a “,” they become the syntactic equivalent of a single expression.)

```

sum=0;
for (current=list; current!=NULL; current=current->next)
    sum+=current->value;    /* One action in each slot. */

for (current=list, sum=0; current!=NULL; current=current->next)
    sum+=current->value;    /* The usual way to write it in C */

current=list, sum=0;
for ( ; current!=NULL; )
    sum+=current->value, current=current->next;

current=list, sum=0;    /* Very bad style. */
for ( ; ; ) if (current==NULL) break;
    else sum+=current->value, current=current->next;

```

The following two loops do the same process and work correctly on nonempty lists. Both will cause a run-time error if `current` is the `NULL` list.

```

for (current=list, sum=0;
    (current=current->next)!=NULL;
    sum+=current->value);    /* Ugly but efficient. */

current=list, sum=0;    /* A total misuse of the for loop. */
for ( ; current=current->next; sum+=current->value )
    if (current==NULL) break;

```

-
- Put all the initializations relevant to the loop in part (a). Don't initialize loop variables or accumulators earlier, for instance, in the declarations. (This follows from the principle of Local Effects.) Don't put an action here that is repeated in some other loop part.
 - Part (b) should contain the loop test. If some action, such as reading data, needs to be done before the test on every iteration (including the first), put it there also. If you have no loop test here, use an explicit infinite loop.
 - Part (c) normally contains the statement that changes the loop variable, in preparation for the test. If more than one linear data structure is being processed, increment the “current element” pointers for all of them here.
 - The loop body should contain any actions that are left, such as output, function calls, and

Exhibit 10.32. Equivalent loop forms in C.

```
while ( <condition> ) <scope>;  
for ( ; <condition> ; ) <scope>;
```

control statements.

Because the initialization, test, and increment parts of the iteration element are arbitrary expressions, evaluated every time around the loop, super-efficient implementations are not feasible. The FORTRAN implementation (where the loop variable is kept in an index register) makes both incrementation and use as a subscript very fast. This cannot be done if the increment clause is an arbitrary list of expressions to evaluate.

On the other hand, this iteration element has total flexibility and no other forms of iteration are necessary. It could serve as the only iterative form in the language. C does have a `while` statement, but it is equivalent to a degenerate `for` loop [Exhibit 10.32].

10.4 Implicit Iteration

Some languages support particularly powerful operations on aggregates of data in which iteration is implicit. In the modern functional languages this is the only kind of iteration supported, and it plays an important role. To be useful, an iterative construct in a functional language must return a value, since using destructive assignment to modify the contents of an array is prohibited and there is no other way to communicate the results of the iteration.

10.4.1 Iteration on Coherent Objects

Implicit iteration is one of the most striking features of APL. Most operators in APL are defined for numbers but are applied iteratively to their arguments if the arguments are arrays or matrices. Operators defined for arrays are iterated if the arguments are matrices. How iteration is performed on an APL object depends on its number of dimensions and on the operation being iterated.

Compare the first few examples in Exhibit 10.33 to the LISP code in Exhibit 9.28, and compare the last line to the C code in Exhibit 10.34. (The APL operator “=” performs comparison, not assignment.)

When coupled with dynamic storage allocation, automatic implicit iteration is tremendously powerful. Code that would fill many lines in a language with explicit loops can often be written in one line of APL. Compare the last expression in Exhibit 10.33 to its (partial) C equivalent in Exhibit 10.34. The difference in length illustrates the power of implicit looping when used to process all the elements in a data aggregate.

Exhibit 10.33. Use of implicit looping in APL.

```

▽ ANSWER ← A FUNC B           Ⓜ Define a two-argument function named FUNC.
[1] ANSWER ← (3 × A)- B       Ⓜ Return 3 times A minus B.
▽
  X ← 4 9 16 25               Ⓜ Bind X and Y to integer vectors.
  Y ← 1 2 3 4
  Z ← (3 4) ρ 1 2 3 4 1 4 9 16 1 8 27 64   Ⓜ A 3 by 4 matrix.

```

Expression	Value Computed
X + 1	(5 10 17 26)
X + Y	(5 11 19 29)
X FUNC Y	(11 25 45 71)
(X + 1) = X + Y	(1 0 0 0)
Z = 1	((1 0 0 0)(1 0 0 0)(1 0 0 0))
+/ Z=W	If W is a scalar, the result is a vector containing the number of occurrences of W in each row of Z. If W is the same shape as Z, the result is a vector whose length is the number of rows in Z, containing the number of positions in each row where corresponding elements of W and Z match.

Exhibit 10.34. Explicit looping in ANSI C takes more writing.

```

int j, k, count[3];
int w; /* Assume w is initialized before entering loops. */
int z[3][4] = {{1, 2, 3, 4}, {1, 4, 9, 16}, {1, 8, 27, 64}};

```

Given these declarations, the double loop and `printf` statement below do more or less the same thing as the final line of APL code in Exhibit 10.33.

```

for(j=0; j<3; j++)
{
  for (k=count[j]=0; k<4; k++)
    if ( z[j][k] == w ) count[j]++;
}
printf("%d %d %d\n", count[0], count[1], count[2] );

```

Exhibit 10.35. Implicit looping in dBMAN.

Assume the current record type has fields named `lastname`, `sex`, `hours`, `grosspay`, and `withheld`, and that the current file contains records of this type for each employee. Statistics can be gathered on subsets of the file selected by simple statements which access each record in the file and test or process it. Examples of dBMAN commands follow:

1. `display all for lastname = 'Jones' and sex = 'M'`
2. `copy to lowpay.dat all for grosspay < 10000`
3. `sum grosspay, withheld to grosstot, whelddtot all for hours > 0`

Line 1 finds and displays all male employees named Jones. Line 2 creates a new file that contains copies of some of the records in the current file. Line 3 sums the current gross pay and withholding fields for all employees who have logged some working hours.

A more important benefit of implicit looping is its flexibility. Any size or shape of arrays can be accommodated, so long as every operator is called with arguments whose shapes make sense when combined. APL defines iteration patterns for scalar/scalar, scalar/vector, scalar/matrix, vector/vector, vector/matrix, and so on through higher and higher dimensional objects. The APL code that counts the occurrences of `W` in `Z` (`+ Z=W`) will work on any shape matrix, and it also will provide a meaningful answer if `W` is a simple integer. Different C code would have to be written for those cases.

The primary data aggregate type supported by functional languages is the list, and the implicit iterative functions in these languages process lists. In APL the data aggregate type is the array, and operators are iterated over arrays. A data base language such as dBASE also makes extensive use of implied iteration to process all data elements of its basic aggregate type, which is “file”.

dBMAN, a dBASE look-alike for the Atari computer, supports many operations that test or manipulate each record in a file, in turn, possibly producing a new file as the result. Examples of such operations are `display` and `sum` [Exhibit 10.35].

The term “fourth-generation language”, sometimes applied to data base languages, recognizes the great ease with which large volumes of data can be manipulated using implicitly iterative statements.

10.4.2 Backtracking

A few languages exist that not only support implicit iteration, but combine it with a trial-and-error backtracking algorithm to enable the programmer to write pattern matching expressions. Prolog will search a data base of relations and extract items, one at a time, that match a pattern specified by the programmer. SNOBOL and its successor, ICON, search text strings for the first section that

Exhibit 10.36. A data base of Prolog relationships.

- | | | |
|---------------------|-------------------------|-------------------------|
| 1. likes(Sara,John) | 7. likes(John,Jana) | 13. does(John,skating) |
| 2. likes(Jana,Mike) | 8. likes(Sean,Mary) | 14. does(Sean,swimming) |
| 3. likes(Mary,Dave) | 9. does(Mike,skating) | 15. does(Mary,skating) |
| 4. likes(Beth,Sean) | 10. does(Jana,swimming) | 16. does(Beth,swimming) |
| 5. likes(Mike,Jana) | 11. does(Sara,skating) | |
| 6. likes(Dave,Mary) | 12. does(Dave,skating) | |
-

matches a specified pattern (these patterns can be very complex).

In all three languages the semantics of the search command requires that all possibilities must be examined before returning a FAIL result. In some cases this can be done only by making a trial match, then continuing the matching process. If the process later fails, then the system must back up to the point it made the trial match, discard that possibility, and search for a different trial match. This process is called “backtracking”. The implementation of backtracking uses a stack to store pointers to the positions at which each trial match was made. It is a well-understood recursive process but nontrivial to write and potentially very slow to execute.

We can illustrate this process by a Prolog example. Prolog maintains a data base of facts and relations. It searches this data base in response to queries, looking for a set of facts that match the user’s requirements. Each query is one or a series of patterns to be matched, in order. As each pattern is matched, any variables in the pattern become bound to the corresponding values in the fact, and a marker is left pointing to that fact. If some later pattern cannot be matched, Prolog will come back to this marker and try to find another match. All searches start at the top of the data base and move sequentially through it. If the end of the data base is reached and a matching fact is not found, Prolog returns the answer “no”.

Exhibit 10.36 contains a brief data base about a group of teenagers, their current crushes, and their favored activities. The numbers listed are not part of the data base but are for the purpose of tracing the pattern matching operation. The reader should keep in mind that this data base and the query we will trace are both very simple. Prolog can express far more complex relationships with multiple, nested dependencies.

The owner of this data base is looking for a compatible couple to share an afternoon of sport with himself and his date. Exhibit 10.37 gives his query, which contains four goals to be satisfied together. When and if a match is found, Prolog will tell him the girl, the guy, and the activity he wants. The chart below the query traces the steps Prolog would follow in trying to match all the patterns.

String processing and artificial intelligence application programs commonly need to perform exhaustive searches. Thus the implicit iteration plus backtracking provided in ICON and Prolog make them very appropriate and very powerful languages for these application areas.

Exhibit 10.37. Answering a Prolog query.**Query:** ?- likes(X,Y), likes(Y,X), does(X,Z), does(Y, Z)

Marker for Goal 1	Bind X to	Bind Y to	Marker for Goal 2	Marker for Goal 3	Bind Z to	Marker for Goal 4	Next Step
1	Sara	John	7? no				Try goal 1. Try goal 2. Retry goal 1.
2	Jana	Mike	5? yes	10	swimming	9? no	Try goal 2. Try goal 3. Try goal 4. Retry goal 3.
			no	no			Retry goal 2. Retry goal 1.
3	Mary	Dave	6? yes	15	skating	12? yes	Try goal 2. Try goal 3. Try goal 4. Solution found.

Answer: X=Mary, Y=Dave, Z=skating**Exercises**

1. What is a primitive control instruction?
2. What is the difference between a jump and a conditional jump?
3. What is the purpose of using subroutines in a program?
4. Why is “sequence” considered to be a basic control structure in procedural languages but not in functional ones?
5. What is a one-in-one-out unit? Give a specific example.
6. Of what does a sufficient set of basic control structures consist, according to Dijkstra? Why are these structures easier to debug than programs that use the unconstrained GOTO?
7. What is the difference between a conditional statement and a conditional expression? Give an example of each.

8. How are structured conditionals different from simple conditional branch statements?
9. In what two ways are **THEN** and **ELSE** clauses delimited in modern procedural languages?
10. What are the advantages of each method discussed in question 9?
11. What is a **CASE** statement?
12. Why can a **CASE** statement be implemented more efficiently than a series of conditionals?
13. Why is an infinite loop considered a one-in-zero-out control structure?
14. What are the two simple conditional loops that can be formed? How are they implemented as structured loops?
15. What is a general loop? How is it different from the other structured loops? What are its advantages?
16. What is a counted loop? When is it used?
17. What is the loop variable? How has it evolved through the years?
18. What happens to the loop variable upon exit from the loop?
19. What is a tripcount? How do changes to the loop variable affect the tripcount?
20. What is an iteration element? How does it differ from a counted loop? (Both have been called **FOR** loops.)
21. To understand exactly how a **for** loop operates you must know several details about what happens in what order and under what conditions. Make a list of questions which you must answer to interpret the meaning of a **for**.
22. The **for** loop in Pascal is superficially similar to the **for** loop in C but has very different semantics. List three ways in which the Pascal loop is restricted when compared to the C loop.
23. Diagram all five of the following **FORTH** loops, and write the keywords in appropriate places on each diagram.
 - a. `begin...again`
 - b. `begin...while...repeat`
 - c. `begin...until`
 - d. `do...loop`
 - e. `do...+loop`

24. What is implicit iteration? What is its purpose?
25. Name two programming languages that support iteration without the use of explicit control structures such as FOR, WHILE, GOTO, or recursion. What kind of iteration is supported?
26. Why is implicit iteration often combined with backtracking? How is it used?