Language Summary 8.3: COBOL

Features: COBOL (COmmon Business Oriented Language) has been widely used since the early 1960s for business applications of computers.

History: COBOL has evolved through a sequence of design revisions, beginning with the first version in 1960 and later revisions in 1974 and 1984. COBOL development was organized by the U.S. Defense Department under the direction of Grace Hopper. Some of the ideas within COBOL developed from Univac's FLOWMATIC, including the use of nouns and verbs to describe actions and the separation of data descriptions with commands — two essential attributes of COBOL. A unique goal of COBOL was to develop a language programmed in "natural English." While the resulting language is somewhat readable, it does have a formal syntax, and without adequate training, cannot be programmed easily.

Translation of COBOL into efficient executable code is complex because of the number of different data representations and the large number of options for most statements. Most of the early COBOL compilers were extremely slow, but more recently improvements in compilation techniques have led to relatively fast COBOL compilers, producing fairly efficient executable code.

Example: COBOL programs are organized into four *divisions*. This organization is a result of two design goals: separating *machine-dependent* from *machine-independent* program elements and that of separating data descriptions from algorithm descriptions. The result is a tripartite program organization: The PROCEDURE division contains the algorithms, the DATA division contains data descriptions, and the ENVIRONMENT division contains machine-dependent program specifications such as the connections between the program and external data files. A fourth *IDENTIFICATION* division serves to name the program and its author and to provide other commentary as program documentation.

The COBOL design is based on a static run-time structure. No run-time storage management is required, and many aspects of the language are designed to allow relatively efficient run-time structures to be used (although this goal is less important than that of hardware independence and program transportability).

The language uses an English-like syntax, which makes most programs relatively easy to read. The language provides numerous optional *noise words* that may be used to improve readability. The syntax makes COBOL programs easy but relatively tedious to write because even the simplest program becomes fairly lengthy. Figure 10.1 gives a brief overview of COBOL syntax.

Reference: J. E. Sammet, "The early history of COBOL," *ACM History of Programming Languages Conference*, Los Angeles, CA (June 1978) *(SIGPLAN Notices (13)8 [August 1978])*, 121–161.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. SUM-OF-PRICES.
 AUTHOR. T-PRATT.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. SUN.
 OBJECT-COMPUTER. SUN.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
      SELECT INP-DATA ASSIGN TO INPUT.
      SELECT RESULT-FILE ASSIGN TO OUTPUT.
 DATA DIVISION.
 FILE SECTION.
 FD INP-DATA LABEL RECORD IS OMITTED.
 01 ITEM-PRICE.
      02 ITEM PICTURE X(30).
      02 PRICE PICTURE 9999V99.
 WORKING-STORAGE SECTION.
 77 TOT PICTURE 9999V99, VALUE 0, USAGE IS COMPUTATIONAL.
 01 SUM-LINE.
      02 FILLER VALUE ' SUM ='PICTURE X(12).
      02 SUM-OUT PICTURE $$,$$$,$$9.99.
      02 COUNT-OUT PICTURE ZZZ9.
      ... More data
 PROCEDURE DIVISION.
 START.
      OPEN INPUT INP-DATA AND OUTPUT RESULT-FILE.
 READ-DATA.
      READ INP-DATA AT END GO TO PRINT-LINE.
      ADD PRICE TO TOT.
      ADD 1 TO COUNT.
      MOVE PRICE TO PRICE-OUT.
      MOVE ITEM TO ITEM-OUT.
      WRITE RESULT-LINE FROM ITEM-LINE.
      GO TO READ-DATA.
PRINT-LINE.
     MOVE TOT TO SUM-OUT.
     ... More statements
     CLOSE INP-DATA AND RESULT-FILE.
     STOP RUN.
```

Figure 8.6. Sample COBOL text.

in memory. The order in which they appear in memory determines the order in which they are executed.

Conditional Statements

A *conditional statement* is one that expresses alternation of two or more statements, or optional execution of a single statement—where *statement* means either a single basic statement, a compound statement, or another control statement. The choice of alternative is controlled by a test on some condition, usually written as an expression involving relational and Boolean operations. The most common forms of conditional statement are the **if** and **case** statements.

If statements. The optional execution of a statement is expressed as a *single-branch* **if**, viz.,

```
if condition then statement endif
```

whereas a choice between two alternatives uses a *two-branch* **if**, namely,

```
if condition then $statement_1$ else $statement_2$ endif
```

In the first case, a condition evaluating to true causes the *statement* to be executed, whereas a false condition causes the statement to be skipped. In the two-branch **if**, $statement_1$ or $statement_2$ is executed depending on whether *condition* is true or false.

A choice among many alternatives may be expressed by nesting additional **if** statements within the alternative statements of a single **if** or by a *multibranch* **if**:

```
if $condition_1$ then $statement_1$
      elsif $condition_2$ then $statement_2$
      ...
      elsif $condition_n$ then $statement_n$
      else $statement_{n+1}$ endif
```

Case statements. The conditions in a multibranch **if** often take the form of repeated testing of the value of a variable, as in the following code:

```
if Tag = 0 then $statement_0$
     elsif Tag = 1 then $statement_1$
     elsif Tag = 2 then $statement_2$
     else $statement_3$
     endif
```

This common structure is expressed more concisely as a **case** statement, such as in Ada:

```
case Tag is
    when 0 => begin
        statement_0
        end;
    when 1 => begin
        statement_1
        end;
    when 2 => begin
        statement_2
        end;
when others => begin
        statement_3
        end;
end case
```

In general, the variable *Tag* may be replaced by any expression that evaluates to a single value, and then the actions for each of the possible values are represented by a compound statement preceded by the value for the expression that would cause that compound statement to be executed. Enumeration types and integer subranges are particularly useful in setting up the possible values that the expression in a **case** statement may return. For example, if variable *Tag* is defined as having the subrange 0..5 as its type, then during execution of the **case** statement, values for *Tag* of 0, 1, or 2 will cause $statement_0$, $statement_1$, or $statement_2$, respectively, to be executed, and any other value will cause $statement_3$ to be executed.

Implementation. **If** statements are readily implemented using the usual hardware-supported branch and jump instructions (the hardware form of conditional and unconditional **goto**). **Case** statements are commonly implemented using a jump table to avoid repeated testing of the value of the same variable. A *jump table* is a vector stored sequentially in memory, each of whose components is an unconditional jump instruction. The expression forming the condition of the **case** statement is evaluated, and the result is transformed into a small integer representing the offset into the jump table from its base address. The jump instruction at that offset, when executed, leads to the start of the code block representing the code to be executed if that alternative is chosen. The resulting implementation structure for the **case** statement is shown in Figure 8.7.

Iteration Statements

Iteration provides the basic mechanism for repeated calculations in most programs. (Recursive subprograms of the next chapter form the other.) The basic structure of an iteration statement consists of a *head* and a *body*. The head controls the number of times that the body will be executed, whereas the body is usually a (compound) statement that provides the action of the statement. Although the

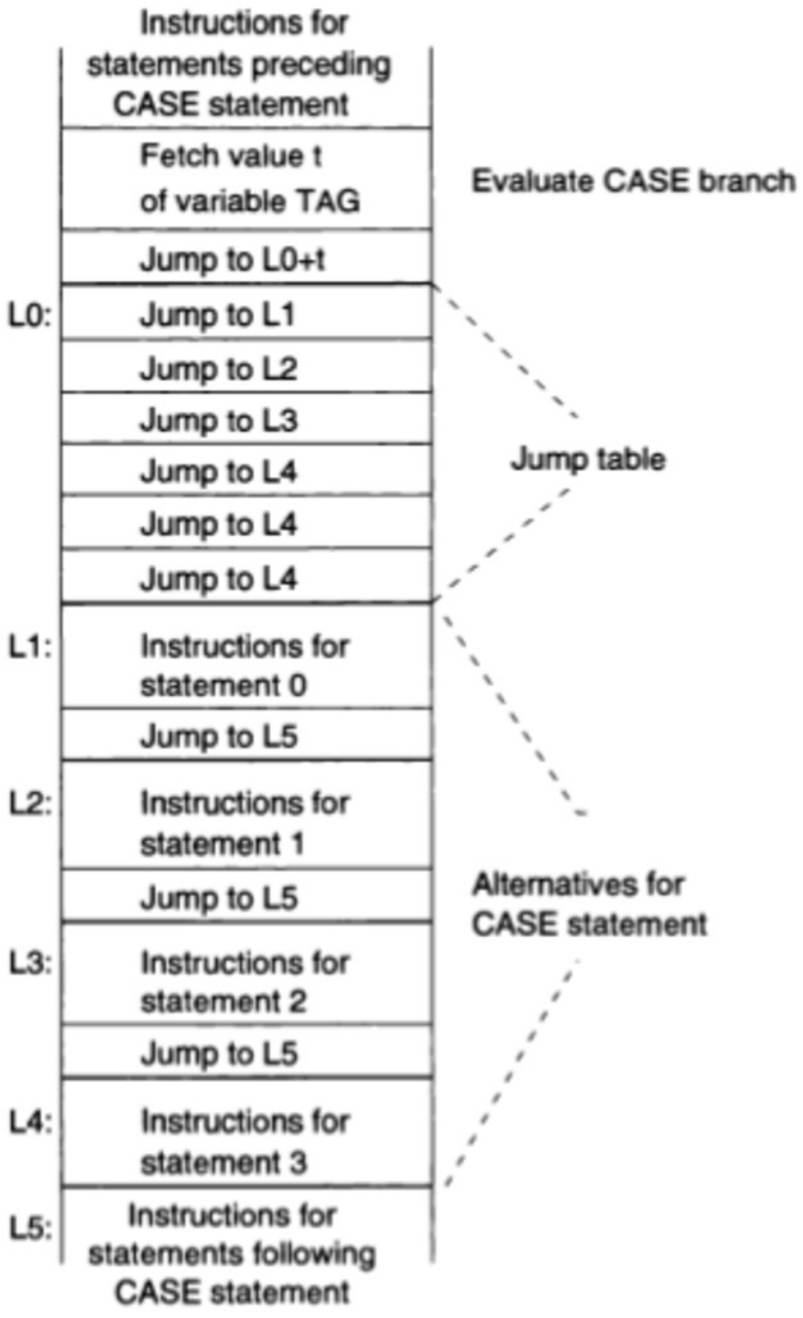

Instructions for statements preceding CASE statement
Fetch value t of variable TAG
Evaluate CASE branch
Jump to L0+t
L0:
Jump to L1
Jump to L2
Jump to L3
Jump to L4
Jump to L4
Jump to L4
Jump table
L1:
Instructions for statement 0
Jump to L5
L2:
Instructions for statement 1
Jump to L5
Alternatives for CASE statement
L3:
Instructions for statement 2
Jump to L5
L4:
Instructions for statement 3
L5:
Instructions for statements following CASE statement

Figure 8.7. Jump table implementation of **case** statement.

bodies of iteration statements are fairly unrestricted, only a few variants of head structure are usually used. Let us look at some typical ones.

Simple repetition. The simplest type of iteration statement head specifies that the body is to be executed some fixed number of times. The COBOL PERFORM is typical of this construct:

perform *body* K **times**

The statement causes K to be evaluated and then the body of the statement to be executed that many times.

However, even this simple statement has some subtle issues. Can K be reevaluated in *body* and change the number of iterations of the loop? What if K is 0 or negative? How does this affect execution?

Although these questions may seem like hairsplitting for this simple iteration statement, the same questions arise in each form of the statement, and thus it is important to look at them in their simplest form here. In each case, it is important to ask: (1) When is the termination test made? and (2) When are the variables used in the statement head evaluated?

Repetition while condition holds. A somewhat more complex iteration may be constructed using a *repeat while* head. A typical form is

while *test* **do** *body*

In this form of iteration statement, the test expression is reevaluated each time after the body has been executed. Note that execution of the body will change some of the values of variables appearing in the test expression; otherwise the iteration, once begun, would never terminate.

Repetition while incrementing a counter. The third alternative form of iteration statement is the statement whose head specifies a variable that serves as a counter or index during the iteration. An initial value, final value, and increment are specified in the head, and the body is executed repeatedly using first the initial value as the value of the index variable, then the initial value plus the increment, then the initial value plus twice the increment, and so on, until the final value is reached. In FORTRAN-77, this is the only form of iteration statement available. The ALGOL **for** statement illustrates the typical structure

for I := 1 **step** 2 **until** 30 **do** *body*

In its general form, the initial value, final value, and increment may be given by arbitrary expressions as in

for K := N-1 **step** 2× (W-1) **until** M× N **do** *body*

Again the question arises as to when the termination test is made and when and how often the various expressions are evaluated. Here the question is of central importance additionally for the language implementor because such iteration statements are prime candidates for optimization, and the answers may greatly affect the sorts of optimizations that can be performed.

Data-based repetition. Sometimes the format of the data determines the repetition counter. Perl, for example, has a **foreach** construct:

foreach $X (@arrayitem) { ... }

For each pass through the loop, scalar variable *$X* will have a value equal to the next element of array *@arrayitem*. The size of the array determines how many times the program loops.

Indefinite repetition. Where the conditions for loop exit are complex and not easily expressible in the usual loop head, a loop with no explicit termination test in the head is often used—for example, in the Ada construct

```
loop
    ...
    exit when condition;
    ...
end loop;
```

or in Pascal, using a **while** loop with a condition that is always true:

```
while true do begin ... end
```

The C **for** statement permits all of these concepts in one construct:

for(*expression*$_1$; *expression*$_2$; *expression*$_3$){*body*}

Here, *expression*$_1$ is the initial value, *expression*$_2$ is the repeat condition, and *expression*$_3$ is the increment. All of these expressions are optional, allowing for much flexibility in C iterations. Some C sample iteration loops can be specified as follows:

Simple counter from 1 to 10:	**for**(i=1; i<=10; i++){*body*}
Infinite loop:	**for**(;;){*body*}
Counter with exit condition:	**for**(i=1;i<=100 && NotEndfile; i++){*body*}

Implementation of loop statements. Implementation of loop-control statements using the hardware branch/jump instruction is straightforward. To implement a **for** loop, the expressions in the loop head defining the final value and increment must be evaluated on initial entry to the loop and saved in special temporary storage areas, where they may be retrieved at the beginning of each iteration for use in testing and incrementing the controlled variable.

Problems in Structured Sequence Control

A **goto** statement is often viewed as a last resort when the structured control statements described earlier prove inadequate for the expression of a difficult sequence-control structure. Although in theory it is always possible to express any sequence-control structure using only the structured statement forms, in practice a difficult form may not have any natural expression directly using only those statements. Several such problem areas are known, and often special control constructs are provided for these cases that make use of a **goto** statement unnecessary. The most common are the following.

Multiple exit loops. Often several conditions may require termination of a loop. The search loop is a common example: A vector of K elements is to be searched for the first element that meets some condition. The loop terminates if either the end of the vector is reached or an appropriate element is found. Iteration through the elements of a vector is naturally expressed using a **for** loop:

```
for I := 1 to K do
    if VECT[I] = 0 then goto α {α outside the loop}
```

In a language such as Pascal, however, either a **goto** statement must be used to escape from the middle of the loop or the **for** loop must be replaced by a **while** loop, which obscures the information contained in the **for** loop head about the existence and range of the index variable I.

The **exit** statement in Ada and the **break** statement of C provide an alternative construct for expressing such loop exits without use of a **goto** statement:

```
for I in 1 .. K loop
    exit when VECT(I) = 0;
end loop;
```

do-while-do. Often the most natural place to test whether to exit a loop comes not at the beginning or end of the loop, but in the middle, after some processing has been done, as in

```
loop
    read(X)
    if end_of_file then goto α {outside the loop}
    process(X)
end loop;
```

This form is sometimes called the do-while-do because a midpoint **while** can handle this sequence:

```
dowhiledo
    read(X)
    while (not end_of_file)
    process(X)
end dowhiledo
```

Unfortunately, no common language implements this structure, although in C, **if** (*condition*) **break** comes close and Ada's **exit when** is similar.

Exceptional conditions. Exceptions may represent various error conditions, such as unexpected end-of-file conditions, subscript range errors, or bad data to be processed. The statements that handle the processing of these exceptional conditions are often grouped at a special place in the program, such as at the end of the subprogram in which the exception might be detected, or possibly in another subprogram used only to handle exceptions. Transfer from the point where the exceptional condition is detected to the exception handler (group of statements) often is best represented using a **goto** statement in languages that do not implement special exception handling statements. However, Ada and ML provide special language mechanisms for defining exception handlers and for specifying the control transfer needed when an exception is detected. The Ada **raise** statement is typical of this exception handling:

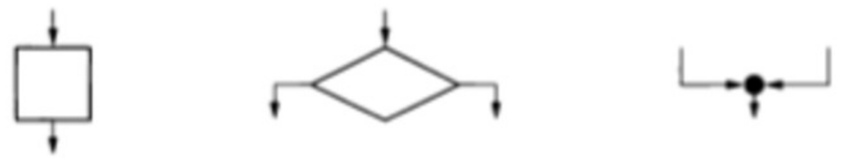
raise BAD_CHAR_VALUE

This statement transfers control to the exception-handling statements that are associated with the exception name BAD_CHAR_VALUE. Exceptions and exception handlers are discussed further in Section 11.1.1.

8.3.3 Prime Programs

Although the collection of control structures presented in this chapter, at first glance, seems like a random assortment of statements, the theory of *prime programs* can be used to describe a consistent theory of control structures. The prime program was developed by Maddux [MADDUX 1975] as a generalization of structured programming to define the unique hierarchical decomposition of a flowchart.

We assume program graphs contain three classes of nodes:



Function nodes represent computations by a program and are pictured as boxes with a single arc entering such a node and a single arc leaving such a node. Intuitively, a function node represents an assignment statement, which causes a change in the state of the virtual machine after its execution.

Decision nodes are represented as diamond-shaped boxes with one input and two output arcs labeled *true* and *false*. These represent predicates, and control flows out of a decision box on either the true or false branch.

A join node is represented as a point where two arcs flow together to form a single output arc.

Every flowchart consists of these three components. We define a *proper program*, which is our formal model of a control structure, as a flowchart that:

1. has a single entry arc,
2. has a single exit arc, and