INTRODUCTION AND SCOPE

Gerald Weinberg, in his book The Psychology of Computer Programming suggests that we cannot really measure the goodness of programs on an absolute scale, and that we generally cannot even measure them on a relative scale. There is indeed evidence that rapid quantification of software quality is not really feasible, because simple formulas can often be misleading and hence not very credible. One could, for example, measure program complexity in terms of the fraction of program statements which are branch statements. Consider however two programs, one with 10 decision points leading to 10 different tasks, each consisting of one instruction, and one with 3 decision points leading to 3 tasks, each consisting of 30 instructions. The “complexity” rating of the first would be 0.50, while that of the second would be 0.09. Surely two such programs would not differ so greatly in complexity.

Thus, we are led to less quantifiable measures of software quality, and a cursory glance at recent literature indicates that “simplicity” is a most desirable characteristic. The simplification of a complex task can be achieved by modularizing it into separate, smaller tasks. We further require that each task be discrete and visible, that it be self-contained (thus constraining the assumptions it makes regarding the implementation of other tasks), that it have a single entry point and a single exit, and that when invoked by another task or module it returns to a standard point.

One way of enhancing the simplicity of a program is through structured modularity, which is achieved through the separation, within a module, of data, processing code, and control, and through the maintenance of a simple, visible, control structure.

This paper concerns itself with how this can be accomplished using the COBOL language as defined in the 1974 COBOL Standard. We will look at how structured constructs can be simulated using standard COBOL verbs, and discuss the micro-efficiency of using these constructs versus less disciplined techniques. We also will point out certain deficiencies in COBOL 74 which can impair program clarity, and make suggestions on future improvements.

IMPLEMENTATION OF STRUCTURED CONSTRUCTS

Structured programming may be viewed as a set of rules designed to enhance a program’s readability, thereby reducing stylistic differences between programs written by different individuals, and improving a programmer’s ability to understand and modify existing programs. The rules are, by now, well known:

(a) Use of code formatting conventions (e.g., indentation) to typographically represent control logic.
(b) Limit subroutine size (e.g., to the number of lines that can fit on a listing page).
(c) Have but a single entry and a single exit per subprogram.
(d) Limit the control forms to three basic structures,

\[ S = \text{statement}, \ P = \text{predicate} \]

\[ \begin{align*}
&\text{Sequence:} \ S_1 \ S_2 \\
&\text{If-then-else:} \ \text{If } P \ \text{then } S_1 \ \text{else } S_2 \\
&\text{Do-while:} \ \text{While } P \ \text{do } S
\end{align*} \]

The basic tool in COBOL for implementing structured programming is the PERFORM verb. Experience has indicated that it is wise to use the THRU option to define the scope of the PERFORMed code, since there is a tendency to neglect definition of the scope of a SECTION. In the following discussion, some familiarity with structured programming and COBOL is assumed.

If-then-else

The if-then-else construct can be implemented in three different ways. The most common is

IF condition
  statement-1
  statement-2
  \[ \ldots \] 
  statement-n
Indexed looping is implemented by:

```
PERFORM paragraph-name THRU paragraph-name-exit
VARYING iden-1 FROM iden-2
BY iden-3
UNTIL (NOT P).
```

**Do-while**

The do-while construct must be "simulated" in COBOL by the use of PERFORM verb with the UNTIL option. The UNTIL option is similar to the WHILE construct in that testing of the condition predicate occurs prior to any code being executed, but differs from WHILE in that termination occurs when the condition is "true" rather than "false." Thus, the do-while construct is:

```
PERFORM paragraph-name THRU paragraph-name-exit UNTIL (NOT P).
```

The negation of P is necessitated by the aforementioned difference in the termination condition between UNTIL and WHILE.
This implementation is not recommended, for reasons which will be discussed in a later section.

Micro-efficiency of the structured constructs

Many COBOL users have expressed the concern that the extensive use of structured constructs (i.e., avoiding the GO TO) may create inefficiencies in terms of execution time and memory utilization. Our experience indicates that some degradation may indeed occur, but its degree is slight.

Consider for example the use of PERFORM ... THRU versus GO TO. On the IBM 8/360 V4 COBOL Compiler a GO TO translates to a conditional branch. Thus, execution of a paragraph requires 2 instructions (one to go, one to return), there is less flexibility (the paragraph cannot be "executed through"), and the control structure is hidden. Execution of PERFORM ... THRU requires five instructions, the PERFORMed paragraph can be executed in-line, and the control structure is clear.

Looping is accomplished by using the do-until construct. This requires 18 machine instructions for one compiler we have examined. Consider however the alternative (by using GO TO). This would require the programmer to:

- Set lower and upper bounds, and the increment.
- Increment the index.
- Test the condition and branch.
- Reset the index.

These tasks require 14 machine instructions on the same compiler. The resultant saving of 4 machine instructions hardly warrants the loss of control logic visibility.

Finally, consider the use of GO TO ... DEPENDING ON versus ALTER as a means of implementing the case construct. A 4-case example required 15 instructions on an IIS 6050 using the GO TO ... DEPENDING ON instruction. A series of 4 IF ... ALTER ... statements required 32 instructions. Not only does ALTER obscure a program's control structure, but in fact is not a very efficient verb.

We have found that attempts to recode existing programs so as to delete GO TO's can result in confusion. The use of structured programming as a discipline must be approached at the level of program formulation if it is to be successful.

The Federal COBOL Compiler Testing Service recently recoded the executive program for the COBOL Compiler Validation System. This program is about 3000 COBOL instructions in size and is quite complex. By reformulating the system we were able to recode it at the cost of some 5 compilations. Of three severe errors that we encountered, 2 were caused by GO TO's which had been retained in the program.

The inter-program communication module

Modularity is a byproduct of structured programming. Modularity in large programming systems is difficult to achieve without a true subroutine capability. This capability is provided in COBOL 74 through the interprogram communication (IPC) module. It allows the transfer of control from one program to another within the same run unit (through the CALL statement) and for sharing of data, or parameter passing (through the USING option). Additionally, programs whose name is not known at compile time may be invoked at execute time, and the availability of memory for a called program may be determined at execute time. While these features have been present in some COBOL implementations, they now have been placed within the standard for the first time. This is surely one instance in which COBOL 74 is a major improvement over COBOL 68. This capability not only enhances modularity, but allows for truly reusable code by providing the means for easily detaching and reattaching subprograms. The presence of the LINKAGE section allows for the grouping of passed parameters, thereby providing good interface visibility as well as some "locality" of data.

DEFICIENCIES AND FUTURE REQUIREMENTS

The revised COBOL language possesses many features which make a positive contribution to the writing of clear, simple, and therefore (hopefully) reliable programs. Regrettably, the language also has many deficiencies which tend to inhibit good programming.

Perhaps the most serious of these is the sheer size of the language. Ideally, a programmer using COBOL for a particular class of applications should have to understand only those features of the language pertinent to that application; while the programmer using COBOL in a broader way should be able to use and understand all of its features. This requires modularity of design, but also smooth integration of the modules. Yet, someone wishing to use the RELATIVE I/O module of the current standard (Reference 4) needs to know many of the details of the NUCLEUS, SEQUENTIAL I/O, and TABLE HANDLING. It should be noted in passing that the 1974 Standard document is over 500 pages long. Jean Sammet required but 16 pages to describe the basic elements of COBOL 61 (Reference 6). One wonders if the two documents are really describing the same language. Even a simple verb such as DIVIDE has five formats in COBOL 74, with a minimal compiler implementation requiring three of them. Further aggravating the feasibility of a user knowing only part of the language is the large number of reserved words (keywords) in COBOL 74—over 300 of them!

C. A. R. Hoare cites simplicity of design as a necessary condition if a language is to assist the programmer in program design, documentation, and debugging. The designers of COBOL appear to have replaced the objective of simplicity with that of modularity, but as the above example suggests, the result has been a "modular" language which is nevertheless quite complex.

Lack of rigor in the COBOL language specification (i.e., Reference 4) is a particularly troublesome shortcoming. An instance of this is the large number of "undefined" results. The specifications for the ENABLE verb in COBOL 74, for example, are not explicit as to what is to occur if an ENABLE is issued with respect to a device that is already
ENABLEd. Clearly something must take place. This means that the language specified by a given COBOL implementation will differ from that specified by the standard. This same problem is also caused by the many instances in COBOL where the specifications are “implementor defined,” as is the case with the COMPUTE verb.

It has been pointed out in an earlier section that a form of “structured” programming is possible using COBOL. This does however require a “simulation” of the structured constructs (as in the do-until case). Use of the substitutes may indeed enhance control logic visibility, but they are more awkward to use than would be the unenumbered constructs. (It should be noted that there are several commercially available “pre-compilers” which allow the use of the simple structured constructs.) Furthermore, complete “structuredness” is precluded by a lack of vertical parentheses (or block structure) and an absence of “local” data. The former shortcoming can be particularly annoying where IF...THEN...ELSE cases are nested.

Finally, there are features in COBOL 74 whose use tends to obscure a program’s control structure beyond normal bounds of good taste. The possibility of asynchronous exits (e.g., the AT END clause of the READ verb) will obviously alter the control sequence of a program. This “flaw” is not, of course, peculiar to COBOL. One way of alleviating the problem is to test for a “flag” in the program control flow, and to set the flag upon the occurrence of an asynchronous interrupt.

Consider, for example, use of the ADD statement with the ON SIZE ERROR option. If the ADD takes place in the control sequence of statements, the usual GO TO upon an occurrence of the sum interrupt would disrupt this sequence. One could however PERFORM the code which includes the ADD statement. The ON SIZE ERROR option would be used to set a flag upon the occurrence of an overflow. The statement following the PERFORM could then test the flag for a given value, and perform the test appropriate to that value. This would of course be done using the if-then-else construct.

The ALTER verb is a particularly offensive weapon in the hands of anyone wishing to obscure a program’s control logic. Briefly, the ALTER statement causes the modification of the destination paragraph of a given GO TO statement (see Reference 4 for a complete and precise definition). The very fact that the ALTER verb modifies a predetermined sequence of operations inhibits good programming practice.

The control sequence may furthermore be additionally obscured by the fact that GO TO statements which have been modified by an ALTER statement, and which are contained in independent program segments may, in some instances, be returned to their initial states.

CONCLUSION

The modules which have been added to COBOL 68 resulting in COBOL 74 have increased the language’s generality to a level which is at least comparable to PL/I. One may argue that it still falls short of true generality, since its applicability in areas such as simulation, list structure processing, graphics and even numerical computations is less than self-evident. Still, it is as “general” as one may reasonably expect at this time. This trend away from the purely commercial applicability is welcomed. The need to have easily transportable programs is increasingly pervasive, and is more easily accomplished in an environment where one general purpose language is used. Furthermore, today’s programming problems seldom can be neatly classified under a single application field, and this is certainly true in “commercial” environments.

Included in this expanded language are several features which encourage good programming practices. One hopes that further improvements will be made, and that those features detrimental to good programming will be deleted from COBOL.

REFERENCES