ENHANCING COBOL PROGRAM STRUCTURE: SECTIONS VS. PARAGRAPHS

R. M. Richards
Business Computer Information Systems
North Texas State University
Denton, TX 76203

Abstract

COBOL is sometimes criticized for its lack of structurability. This is due primarily to the common but outdated use of paragraphs to achieve structure in COBOL programming.

In fact. COBOL was designed to be highly structurable. The language itself is based on a hierarachical structure consisting of DIVISIONS, SECTIONS, paragraphs, sentences, and statements. The task is to train COBOL programmers to take maximum advantage of the structures built into the language.

One way to do this is to use SECTION structure in the procedure division rather than paragraph structure. SECTION structure has several distinct advantages over paragraph structure and allows for maximum utilization of the structured approach to COBOL programming.

Introduction

Critics of COBOL argue that the language is not easily structurable (1). Since structure is the watchword in programming today, this is seen by many as being a limiting factor in the use of COBOL in the future.

In fact, COBOL is highly structurable. However, since COBOL preceded the structured approach to programming, the task is to retrofit COBOL programming and COBOL programmers to the structured programming approach. This is not the case with a language designed for structure, or designed after the concept of structure became popular.

COBOL As A Structured Language

The designers of COBOL used much foresight in developing the language (2). Even before the concept of structure became popular, they designed a language that is built upon a hierarchy of structures with the DIVISION the highest structure and progressing through the SECTION, paragraph, sentence, and statement, in decreasing order. Their work, at least indirectly, lead to the structured programming concept.

There are four goals or rules of structured programming:

1. eliminate the use of conditional transfers of control;
2. provide only one entry point and only one exit point for each structure;
3. utilize only one process per structure; and
4. use procedure names to document the function of a procedure and its level in the program hierarchy.

The first and second goals are the toughest to achieve in COBOL. In order to eliminate GO TO´s and provide one entry point and one exit point, the PERFORM verb is available. As is discussed later, how it is used is critical.

The third goal is a matter of programming logic, no matter what language is being used. The fourth is actually facilitated by the self-documenting nature of the COBOL language. But what about the first two goals?

The Perils of Paragraph Structure

COBOL programmers achieve varying results when striving for well-structured programs. Most COBOL texts, especially the earlier ones, typically use paragraph structure. This leads to many of the problems with COBOL structure 'since paragraph structure is relatively cumbersome. A typical COBOL routine structured around the paragraph is presented below:

```
 .
 .
   PERFORM PARA-1 UNTIL
         EOF-INDICATOR =
         'DONE'.
 .
 .
PARA-1.
      (process)
      (process)
      READ (filename) INTO
            (working-storage)
            AT END MOVE 'DONE'
            TO EOF-INDICATOR.

PARA-2.
 .
 .
```

A problem is created by this approach since the exit point is not specifically identified. Both the programmer and the computer must look for the place exit will occur. If a command is issued to exit PARA-1 before the process is complete, a basic principle of structure is violated, namely the restriction on unconditional transfers of control.

Having no definite exit point causes some computers to get lost in processing--actually lose their place. Many compilers generate a WARNING message such as follows to notify the programmer of a possible problem:

```
EXIT FROM PERFORMED PROCEDURE
ASSUMED BEFORE PROCEDURE-NAME.
```

Although this problem has been noted on an NAS 8040 in IBM 370 mode, the problem does not seem to be machine-specific. It has also been observed on a microcomputer system running an entirely different version of COBOL.

The immediate solution to the above problem was to put an EXIT statement at the end of the structure (in its own paragraph, of course). With no other change to the program, the bug was eliminated, at least in that case.

This problem was very perplexing. The IBM compiler treats the EXIT statement as a "no-op" statement. Apparently, however, the compiler "likes" the inclusion of an EXIT paragraph at the end of pereformed structures. Some compilers treat EXIT as an operable statement. But even when it is not operational, the definition of the point of structure exit is important to the compiler.

Using THRU and EXIT

Avoiding a situation such as this can be accomplished by placing an EXIT at the end of the procedure and by using the THRU clause in the PERFORM verb. The THRU clause is frequently used when several paragraphs are to be executed in sequence within a given structure. These techniques are, however. frowned upon by those who set the standards for COBOL programming (3).

An example of this technique is shown below:

```
        .
        .
        PERFORM PARA-1 THRU
             PARA-1-X
             UNTIL EOF-INDICATOR
             = 'DONE'.

        .
        .
PARA-1.
     (process)
     (process)
     READ (filename) INTO
          (working-storage)
          AT END MOVE 'DONE'
          TO INDICATOR.
PARA-1-X.
     EXIT.
```

The THRU option causes problems with COBOL programming standards because it leads to less efficient program coding, compilation, and execution. On the other hand, the EXIT has some merit. Overall internal documentation is enhanced because the EXIT verb provides definite exit points for all procedures. This type of structure exit cannot be mistaken, either by the programmer or by the computer.

This type of exit is effected in other languages by the use of delimiters called "capstone" statements such as the END statement. Capstone statements are used to signify the one and only exit from a given procedure. Usage of the EXIT verb in this manner certainly has precedent.

The THRU Statement Is Not Needed In SECTION Structure

The criticism of the THRU is well-taken. If the function of THRU can be implemented without actually coding the THRU clause, why use it? In fact, the THRU function can be implemented in another way--a way that makes a great deal of sense when striving for structure in COBOL programming.

The convention that allows this is using SECTIONS rather than paragraphs as the basis of COBOL structure. By using SECTIONS, the advantages of using EXIT can be realized while, at the same time, eliminating the need for THRU. An example follows:

```
          .
          .
          PERFORM SECTION-1 UNTIL
               EOF-INDICATOR =
               'DONE'.
          .
          .
    SECTION-1 SECTION.
    SECTION-1-PARA.
          (process)
          (process)
          (process)
          READ (filename) INTO
               (working-storage)
               AT END MOVE 'DONE'
               TO EOF-INDICATOR.
    SECTION-1-X.
          EXIT.
    .
    .
```

Structuring the COBOL program in this manner achieves virtually all of the goals of structured programming. There are no unconditional transfers of control. Each procedure has one entrance and one exit, both of which are well-defined. Plus, the problem of premature exit is eliminated since EXIT must occur in a paragraph by itself, according to the syntax of COBOL.

****

COBOL PROGRAM STRUCTURE - continued on page 55

```
Assign to HIGHEST VOLUME the value of 0.00.
For CITY COUNTER going from 1 to 10 do
   If VOLUME [CITY COUNTER] > HIGHEST VOLUME then
      Assign to HIGHEST VOLUME the value of VOLUME [CITY COUNTER].
      Assign to CITY WITH HIGHEST VOLUME the value of CITY [CITY COUNTER].

Write out CITY WITH HIGHEST VOLUME, HIGHEST VOLUME.
```

Conclusion

Since the essential feature of stepwise refinement is its hierarchical structure, this is best illustrated with a tree. A solution tree is a tool that assists in the gradual development of an algorithm. A solution tree provides an advantage over a linear implementation of stepwise refinement in that a tree shows the entire development process at a glance, with the algorithm clearly distinguished as the leaves of the tree. For a complex problem, there would be a main solution tree and one or more subtrees. The leaves of each subtree constitute a subalgorithm, which is routinely translated into a subprogram in the target language.

Bibliography

1. Collins, William, An Introduction to Programming and Pascal, Macmillan, 1984.

2. Shelly, Gary and T. Cashman, Advanced Structured COBOL, Anaheim Publishsing Company, 1978.

****

COBOL PROGRAM STRUCTURE - continued from page 50

Summary

Structuring COBOL programs around SECTIONS rather than paragraphs allows the programmer to achieve the maximum structurability in COBOL. In fact, except for the EXIT paragraph and the paragraph immediately following the SECTION name (as required by many compilers), there is very little need for paragraphs in this scenario. Since there are no unconditional transfers of control in totally structured programming, why use paragraphs?

The technique described above is rapidly becoming a standard practice with many COBOL experts and authors. Because of this, COBOL is attaining a new reputation for structurability.

It is important to note that COBOL ´81 will provide even better programming techniques to achieve structure. The inclusion of an "inline" PERFORM, an actual CASE structure. and capstone statements such as END PERFORM will aid immensely in structuring COBOL programs.

COBOL is a highly structurable programming language. Because the language is evolving to meet future programming needs, it will maintain its dominance in the business world for many years to come. Because of the ability of the language to perform well in the structured programming environment of today, its popularity as a general use language should increase as well.

Footnotes

1. Feingold, Carl, **Fundamentals of COBOL Programming**, Dubuque: Wm. C. Brown Company Publishing, 1983, p. 131.

2. Hopper, G. M., **Automatic Coding for Digital Computers** (Talk presented at The High Speed Computer Conference. Louisiana State University, Feb. 16, 1955), Remington Rand Corp., ECD-1 (1955).

3. Spence, J. Wayne. **COBOL for the 80´s**, St. Paul: West Publishing Co., 1982, p. 158.