STRUCTURED PROGRAMMING CONSIDERED HARMFUL

William Slater
COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF TEXAS AT ARLINGTON

Howard Modell
DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF ARIZONA

It should be obvious that considerable attention has been lavished on improving the Programming Art by attempting to give it some discipline. This attention has produced control structures, like IF-THEN-ELSE, DO-WHILE, DO-UNTIL, SELECT-WHEN, and Coroutines, as well as new languages like PL/1 BLISS, PASCAL, EUCLID, AND SL5. In addition, a set of techniques for the design and coding or programs has been proclaimed, called STRUCTURED PROGRAMMING." (SP)

One of the earliest tenets of SP was that the BRANCH control structure, epitomized by FORTRAN's GOTO statement, is inherently dangerous. This arose from the idea that unrestricted, undisciplined use of it led to programs that were difficult, if not outright impossible, to follow or debug. This led to a deluge of articles expounding on the virtue of removing all GOTO's from one's program, and replacing them with more advanced control structures.

We feel that the advocates of SP have gone a trifle awry in their zeal. By saying that the traditional control structures and programming methods are obsolete or dangerous, and that the newer structures and methods are THE WAY to go, we feel that the advocates of SP are stifling the creative, artistic element in programming.

Therefore, we propose the following additional constructs which we feel will add new life, new gracefulness and new challenge to programming.

First and foremost, we advocate replacing all GOTO's with COMEFROM's.* The semantics of "COMEFROM label" say that control continues at the statement following the COMEFROM immediately after executing the statement with the indicated label. Variations on this statement include: COMEFROM (L1,L2,L3,...) where control passes after executing a statement with one of the listed labels; COMEFROM (L1,L2...), IV where IV is an index into the list of labels, = a Computed COMEFROM; COMEFROM IV, (11,12...) which is an ASSIGNED COMEFROM. In a case of multiple COMEFROM's referring to the same label, the last executed COMEFROM would have preference.

---

* "A Linguistic Contribution to GOTOless Programming"
R. Lawrence Clark, DATAMATION Dec. 1973, pg. 62-63

From STRUCTURED PROGRAMMING IN APL (Geller & Freedman), we steal the construct SKIP n, which says control passes to the nth statement following the SKIP.

To oppose this, we also offer a variety of REPEAT constructs:

- REPEAT FROM label -- which passes control to the specified label,
- REPEAT FROM label1 TO label2 -- which causes reexecution of the specified block of statements, and then control continues following the REPEAT,
- REPEAT FROM l1 TO l2 UNLESS boolean
- REPEAT FROM l1 TO l2 UNTIL boolean -- which are simply conditional forms of the second case,
- REPEAT NEXT n
- REPEAT LAST n -- which causes the previous or next n lines to be repeated. Execution then continues with the statement after the REPEAT.
- REPEAT NEXT n UNTIL boolean
- REPEAT NEXT n UNLESS boolean -- conditional repeats.

Next, we present a variety of flexible IF statements. For example, the construct "IF boolean THEN block UNLESS boolean INWHICHCASE block ELSE block." Then there are the triplets "IF boolean MAYBE block," "IF boolean SOMETIMES block," and "IF boolean PERHAPS block," all of which randomly execute the action-block provided the boolean is .TRUE./'1'B.

For the FORTRAN devotees, we propose

```
IF (boolean) L1,L2,L3
```

which transfers to L1 if the boolean is .TRUE., to L3 if the boolean is .FALSE., and to L2 if it is .MAYBE..

For new block structures, we propose:

```
-DO UNLESS (boolean) ;
-DO MAYBE (boolean) ;
-DO SOMETIMES (boolean) ;
-DONT WHILE (boolean) ;
-DONT UNTIL (boolean) ;
-DONT UNLESS (boolean) ;
-DONT;
```

We strongly suggest replacing the attribute/datatype REAL with the more sensible RATIONAL. Until computers with infinite word lengths become practical, it is confusing, not to mention erroneous, to talk about REAL data.

Further, it is suggested that DECLARE be made into an executable statement, or instead provide a REDECLARE statement. This is advocated in the knowledge that there will be times when a variable's attributes might now be decided upon until certain other data is computed or read-in. Also, there will be occasions when it is desirable to redefine a variable's attributes at runtime.

To enhance the flexibility of the basic assignment statement, we present

```
VAR ~= EXPRESSION /* Unassignment, which states that
whatever value the VAR may have had before, it does
not have the value of the EXPRESSION now.  */

VAR ~=!  EXPRESSION /* Exemption, which says
that VAR cannot ever have had the value of the
expression.  */

VAR ~~  EXPRESSION /*  Approximation, which
says that the value of VAR is approximately the
value of expression.  This is useful in processing
RATIONAL data, where exact values are often
improbable.  */

VAR =!  EXPRESSION /*  Restriction, which states
that VAR can only have the value of EXPRESSION */
```

The last item is a general, all-inclusive EQUATE / DEFINED / EQUIVALENCE statement. We feel it should be permissible to perform the following equates:

```
SQRT = ** (function name equated to special symbol)
ADD = + (Alternate form of symbol or operator)
3 = A (redefinition of constant.  Practical in
FORTRAN.)
2 = 1 (equivalence of constants)
+ = function (user defined operator)
* = X + 1 (expression replace symbol -- MACRO
expansion)
```

It is of interest that several of the above equates are already implemented in one language or another (SNOBOL's OPSYM function, for example).

Just to show that these are indeed useful constructs, we conclude this article with several sample short programs.

```
(1)   BEGIN; SUM = 0; COUNT = 0
           COME FROM BLARG;
           UNTIL (EOF) DONT BEGIN;
                            COUNT ~=! 0;
                            AVERAGE = SUM / COUNT;
                            PUT DATA (AVERAGE);
                            STOP;
                            END;
           GET LIST (X);
           COUNT = COUNT + 1;
      BLARG: SUM = SUM + X;
      END;

(2)   GCD: Procedure (M,N) OPTIONS (RECURSIVE);
      DO UNLESS (M=>N); M <-> N; /* <-> IS EXCHANGE
      OPERATOR */
           N = M - N;
      DO UNLESS (N = 0); M = GCD (M,N);
           RETURN (M);
      END;

(3)   IF BOOL1 THEN DECLARE X RATIONAL;
           UNLESS BOOL2
                INWHICHCASE DECLARE X CHAR(3);
           ELSE DECLARE X FIXED BIN(3);
                UNLESS BOOL3
                   INWHICHCASE DECLARE X BIT(1);

(4)   IF A=B  THEN X = X + 1;
              UNLESS B>C
                 INWHICHCASE X = X - 1;
              ELSE X = 3;
```