PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 3/72

EXTENDING THE CONTROL STRUCTURE OF PL/I

by

J. C. P. Bauer
and
A. L. Furtado

Computer Science Department - Rio Datacenter

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

EXTENDING THE CONTROL STRUCTURE OF PL/I

J. C. P. Bauer
Research Assistant
Computer Science Department
PUC/RJ

and

A. L. Furtado
Associate Professor
Computer Science Department
PUC/RJ

This paper was submitted for publication elsewhere

Series Editor: Prof. A. L. Furtado - March/1972

ABSTRACT

A number of control statements have been suggested as extensions to ALGOL 60. Several languages such as ALGOL W, EULER, BLISS and PASCAL [1] have implemented one or more of them.

Besides contributing to a more natural style in programming, they can be used in situations where a go to statement would seem to be unavoidable. Some authors now tend to believe that the go to statement should not be present in well structured programs [2].

The implementation described here uses the preprocessor facility of PL/I.

This approach has the following disadvantages:

a. the preprocessor phase consumes additional computer time;

b. some characteristics of the new statements were imposed for the convenient use of the preprocessor.

and as advantages:

a. being written in the high level preprocessor language the program to implement the statements is easy to understand and modify;

b. since the PL/I compiler itself is not changed the regular PL/I programs are not affected.

It may be expected that if the new statements prove their value through actual use, which is made possible by this implementation , their addition to the PL/I language could then be considered.

1. The REPEAT - UNTIL group

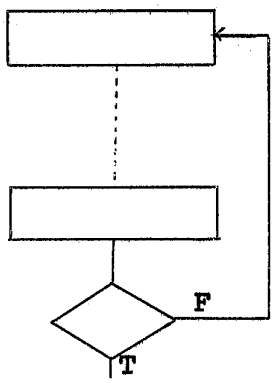
Syntax:

```
REPEAT;

UNTIL (bit - expression);
```

Function: The statements in the group beginning with REPEAT and terminating with UNTIL will be repeatedly executed until the bit-expression is "true" (in the sense of PL/I).

The diagram below shows the situation where the REPEAT-UNTIL group applies:



REPEAT - UNTIL groups can be nested.

Example 1:

```
/*  EXAMPLE OF REPAT - UNTIL
    SOLUTION OF AN EQUATION BY THE NEWTON - RAPHSON METHOD  */

          . . . . . .

          X = 1;
          TOL = 0.0001;
          K = 0;
          MX = 20;
          REPEAT;
               T = FP(X);
               IF T ¬= 0
               THEN D = F(X)/T;
               ELSE DO;
                    PUT LIST ('ZERO DERIVATIVE');
                    STOP;
                    END;
               X = X - D;
               K = K + 1;
          UNTIL(ABS(D/X) < TOL  |  K = MX);
          PUT LIST (X, D, K);

          . . . . . .
```

2. The LEAVE statement [3]

Syntax:

LEAVE (paragraph - label);

where paragraph - label is a constant - label appearing at the head of a block or group, under the form

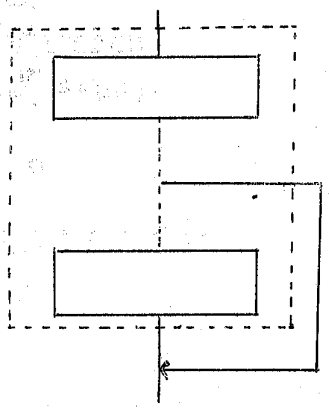
$ (paragraph - label): head of the block or group;

and at the end of the block or group, as:

END $$(paragraph - label);

Function: control is tranferred to the statement following the END statement of the block or group having the paragraph - label.

The diagram shows the situation where LEAVE would be used.



The LEAVE statement itself may appear in a block or group that is nested, to any depth, in the block or group whose execution LEAVE is required to terminate.

Example 2:

```
/*   EXAMPLE OF LEAVE
     THE PRIME NUMBERS FROM  I  TO  J  */

          . . . . . . . . . . .

     I = 10;
     J = 100;
     DO N = I TO J;
          $(L) : DO;
                 DO M = 2 BY 1 WHILE (M ** 2 < = N);
                        IF MOD(N,M) = 0;
                        THEN LEAVE L;
                 END;
                 PUT LIST(N);
                 END  $$(L);
     END;

          . . . . . . . . . . . . . .
```

3. The DO FOREVER group

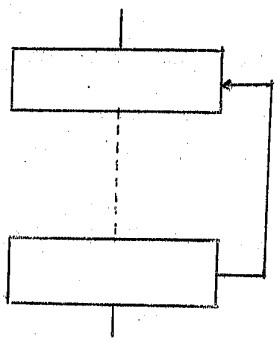
Syntax:

DO FOREVER;

Function: the statements in the group beginning with DO FOREVER and ending with its respective END statement will be repeatedly executed until some statement inside the group causes its termination or termination of the program is caused elsewhere.

DO FOREVER is specifically designed to be used in connection to multitasking.

The diagram is quite simple:



Example 3:

```
/*   EXAMPLE OF DO FOREVER
     SIMULATION OF A PRINTER  */

          . . . . . .

               CHARACTER_PRINTER: PROCEDURE;
               . . . . . .
                    DO FOREVER;
                         WAIT(PRINT_EVENT);
                         COMPLETION (PRINT_EVENT) = '0'B;
                         CHARACTER = MEMORY (ADDRESS);
                         PUT SKIP EDIT (CHARACTER) (A(1));
                    END;
               END CHARACTER_PRINTER;
          . . . . . .

          CALL CHARACTER_PRINTER TASK;  /*  START  I/O  */

          . . . . . .

          COMPLETION (PRINT_EVENT) = '1'B; /* PRINT A CHARACTER  */

          . . . . . .
```

DO FOREVER groups can be nested.

4. The CASE - ENDCASE group

Syntax:

```
CASE (element - expression);

case-label option 1:
      OF (element-expression): simple statement or block or
                                        group;

case-label option 2:
      OF (( element-expression [ , element-expression...] )):
            simple statement or block or group;

ENDCASE;
```

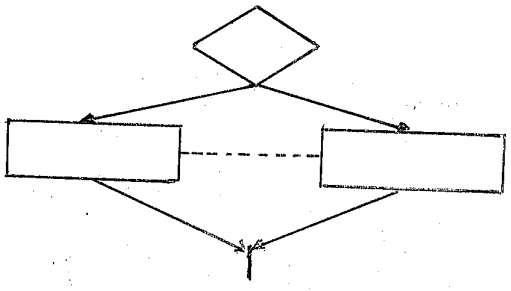
Function: the value of the element expression in CASE is tested for equality with the value (option 1) or values (option 2) of the element-expression(s) in OF; if the test succeeds (option 1) or if it succeeds at least for one member of the list of element - expressions (option 2) the simple or compound (block or group) statement corresponding to OF is executed , and then control is transferred to the ENDCASE statement. Otherwise, the next statement (which must be another OF, an ELSE clause or an ENDCASE) will be executed.

The rules for comparing the element-expressions are the usual PL/I rules for the = operator.

Whenever the statement identified by a case - label is an IF statement it must include an ELSE clause (possibly dummy).

CASE itself admits an ELSE clause. If it is included it must come after all statements with case - labels.

The diagram below shows where CASE is applicable



CASE-ENDCASE groups can be nested, in the sense that the group identified by a case-label or in the ELSE clause can be a CASE-ENDCASE group.

Example 4:

```
/*   EXAMPLE OF CASE
     INTERPRETIVE EXECUTION OF AN ARITHMETIC OPERATION  */

     . . . . . .

     X = 0;
     A = 12;
     B = 3;
     OPR = '.';
     CASE(OPR);
          OF('+'):          X = A + B;
          OF('-'):          X = A - B;
          OF(('*','.')):    X = A * B;
          OF('**'):         X = A ** B;
          ELSE PUT LIST ('ILLEGAL OPERATOR');
     ENDCASE;
     PUT LIST (X) SKIP;
     . . . . . .
```

5. Adding the preprocessor program

a. creation of a data set

```
//NEW  DD  DSN=INFSRC,
//         UNIT=2314,
//         VOL=SER=DC0002,
//         SPACE=(TRK,(100,,10)),
//         DISP=(NEW,CATLG),
//         DCB=(RECFM=F,LRECL=80,BLKSIZE=80),
//         LABEL=EXPDT=99365
```

b. making the preprocessor program a member of the data set

```
//          EXEC   PGM=IEBUPDTE,PARM=NEW
//SYSPRINT    DD  SYSOUT=A
//SYSUT2      DD  DSNAME=INFSRC,DISP=OLD
//SYSIN       DD  *
./       ADD  NAME=PL1H,LEVEL=00,SOURCE=0,LIST=ALL
          . . . . . .
          the preprocessor program
          . . . . . .
./          ENDUP
/*
```

6. Using the new statements

After inserting the preprocessor program into the system, it is left to the user to submit his decks as follows:

```
//    EXEC  PL1LFCLG,PARM.PL1L='MACRO,SOURCE2'
//PL1L.SYSLIB  DD DSN=INFSRC,DISP=SHR
//PL1L.SYSIN DD  *
  ...:PROCEDURE OPTIONS (MAIN);
       % INCLUDE PL1H:
  . . . . . .
```

This will cause the print out of:

a. the user's original program;

b. the preprocessor program;

c. the user's program compiled into standard PL/I by the preprocessor program.

It is clearly necessary to print a. Printing c will sometimes be useful in the debugging phase, as we shall see; anyway it is easy to avoid this print out by changing the EXEC card.

It is clearly unnecessary and wasteful to print b, but the preprocessor itself will have to be changed as to the automatic listing resulting from the % INCLUDE statement.

In order to keeps preprocessor program as simple and short as possible we did not add error messages (But this can be done, for exam - ple, under the form of comments generated together with or instead of PL/I statements). So the user should be aware of error messages given by the pre-processor, the PL/I compiler, and run-time error messages.

This is why we mentioned that print out c could be useful in the debugging phase. Also, the interested reader may wish to see how the new statements are compiled into legal PL/I.

Our experience shows that REPEAT - UNTIL, LEAVE and DO FOREVER are relatively simple to assimilate and errors very seldom occur.

On the contrary, CASE - ENDCASE is guilty of some typical er- rors:

a. the user writes:

```
. . . . . .
OF (element-expression): IF ....
                         THEN ....;
```

the correct sequence being:

```
. . . . . .
OF(element - expression): IF ....
                          THEN ....;
                          ELSE;
```

since where there would not be an ELSE clause an ELSE with a NULL statement has to be included.

b. the user writes:

```
. . . . . .
OF (element-expression): statement;
                         statement;
                         . . . . .
```

the correct sequence being:

```
. . . . . .
OF (element-expression): DO;
                         statement;
                         statement;
                         . . . . .
                         END;
```

since a compound statement is allowed but not a sequence of simple statements.

c. the user writes:

```
. . . . . .
OF (element-expression, element-expression, ....): ....
```

the correct form being:

```
. . . . . .
OF ((element-expression, element-expression,...)):...
```

since option 2 of the case-label requires an extra pair of brackets to denote a list of element-expressions rather than a single one.

d. the user writes:

```
. . . . . .
END CASE;
```

the correct form being:

```
. . . . . .
ENDCASE;
```

no intervening blanks being allowed.

Certain words become reserved words and must not be used. These are, first, the keywords of the new statements, the word OF, the single and the double dollar-sign. Secondly, words beginning with REP and CASE followed by an integer (ex: REP0015).

One of the inconveniences resulting from the fact that such words become reserved words is that the built-in function REPEAT - dealing with strings - cannot be invoked explicitly wherever the REPEAT statement is also used.

7. Listing of the pre-processor program

The sections of the pre-processor program to deal with each of the new statements are indicated by comments.

An interesting feature is the use of character strings to provide stacks (variables PILHA in the REPEAT - UNTIL section and STACK in the CASE - ENDCASE section). The push-down and pop-up operations are respectively done by concatenation and by the SUBSTR built - in function.

For CASE - ENDCASE the lengths of the "cells" in the stack are not necessarily the same. A simple solution was adopted: a delimiter is entered together with each new cell.

We mention this detail to suggest that the PL/I pre-processor facility is not so weak as it may appear from its severe restrictions (such as not allowing the use of arrays).

```
/*   REPEAT - UNTIL   */
      %DCL REPEAT ENTRY RETURNS(CHAR);
      %DCL UNTIL ENTRY(CHAR) RETURNS(CHAR);
      %DCL (COUNTER,N) FIXED;
      %DCL PILHA CHAR;
      %DEACT COUNTER,N,PILHA;
      %COUNTER = 10000;
      %N = 0;
      %PILHA = '' ;
      %REPEAT:PROCEDURE RETURNS(CHAR);
           DCL SUFFIX CHAR;
           COUNTER = COUNTER + 1;
           SUFFIX = SUBSTR(COUNTER,5,4);
           PILHA = SUBSTR(PILHA,1,N) || SUFFIX;
           N = N + 4;
           RETURN ('REP' || SUFFIX || ' : DO');
      %END REPEAT;
      %UNTIL:PROCEDURE(ARG) RETURNS(CHAR);
           DCL (ARG, SUFFIX) CHAR;
           N = N - 4;
           SUFFIX = SUBSTR(PILHA, N+1, 4);
           RETURN ('IF ¬(' || ARG || ')
                          THEN GO TO REP' || SUFFIX || ';' || 'END');
      %END UNTIL;
```

```
/*   LEAVE   */
       %DCL $ ENTRY(CHAR) RETURNS(CHAR);
       %DCL $$ ENTRY(CHAR) RETURNS(CHAR);
       %DCL LEAVE CHAR;
       %LEAVE = ' GO TO ';
       %$ : PROCEDURE(LABEL) RETURNS(CHAR);
            DCL LABEL CHAR;
            RETURN ('$' || LABEL);
       %END $;
       %$$: PROCEDURE(LABEL) RETURNS(CHAR);
            DCL LABEL CHAR;
            RETURN ('$' || LABEL || ' ; ' || LABEL || ' : ');
      %END $$;


/*   DO FOREVER   */
       %DCL FOREVER CHAR;
       %FOREVER = 'WHILE (''1'' B)';
```

```
/*   CASE - ENDCASE   */
       %DCL OF ENTRY(CHAR) RETURNS(CHAR),
            CASE ENTRY(CHAR) RETURNS(CHAR),
            ENDCASE ENTRY RETURNS(CHAR);
       %DCL (NUMBER,KEY) FIXED,
            (EXPR,STACK) CHAR;
       %DEACT NUMBER,EXPR,KEY,STACK;
       %NUMBER = 10000;
       %STACK = ' ';
       %EXPR = ' ';
       %CASE:PROCEDURE(COND) RETURNS(CHAR);
            DCL COND CHAR;
            KEY = 0;
            STACK = EXPR || ':' || STACK;
            EXPR = COND;
            RETURN ('DO');
       %END CASE;
       %OF: PROCEDURE(COND) RETURNS(CHAR);
            DCL (COND,CHAR,AUX) CHAR,
                STRING CHAR,
                (PAR#, QUOTE, K) FIXED,
                HEAD FIXED;
            NUMBER = NUMBER + 1;
            AUX = ') = (' || EXPR;
            IF KEY = 0
            THEN DO;
                 STRING = 'IF (';
                 KEY = 1;
                 END;
```

```
ELSE STRING = 'ELSE IF(';
IF SUBSTR(COND,1,1) ¬= '('
THEN STRING = STRING  || COND  || AUX;
ELSE DO;
     PAR# = 1;
     QUOTE = 0;
     K = 1;
     HEAD = 1;
  SCAN: K = K + 1;
     CHAR = SUBSTR(COND,K,1);
     IF QUOTE = 1
     THEN DO;
          IF CHAR = ''''
          THEN QUOTE = 0;
          GO TO SCAN;
          END;
     ELSE IF PAR# > 1
          THEN DO;
               IF CHAR = ')'
               THEN PAR# = PAR# - 1;
               ELSE IF CHAR = '('
                    THEN PAR# = PAR#  + 1;
               GO TO SCAN;
               END;
```

```
          IF CHAR ¬= ')'
          THEN DO;
               IF CHAR = '('
               THEN PAR# = PAR# + 1;
               ELSE IF CHAR = ''''
                    THEN QUOTE = 1;
                    ELSE IF CHAR = ','
                         THEN DO;
                              STRING = STRING ||
                               SUBSTR(COND,HEAD,K-HEAD) ||
                               AUX || ') | (' ;
                              HEAD = K + 1;
                              END;
               GO TO SCAN;
               END;
          IF HEAD > 1
          THEN AUX = ' = (' || EXPR || ')';
          STRING = STRING || SUBSTR(COND,HEAD) || AUX;
          END;
     RETURN (STRING || ') THEN CASE' || SUBSTR(NUMBER,5,4));
%END OF;
%ENDCASE:PROCEDURE RETURNS(CHAR);
     DCL I FIXED;
     I = 0;
  LOOP:I = I + 1;
     IF SUBSTR(STACK,I+1,1) ¬= ':'
     THEN GO TO LOOP
     EXPR = SUBSTR(STACK,1,I);
     STACK = SUBSTR(STACK,I+2);
     RETURN ('END');
%END ENDCASE;
```

8. References

[1] - Wirth, N. - "The Programming Language Pascal" - Acta Informatica - vol. 1, fasc. 1, 1971 - pp 35-63.

[2] - Dijkstra, E. W. - "Go-to Statement Considered Harmful" - Letter to the Editor - CACM, 11,3, March 1968.

[3] - Wulf, W.A. - "Programming without the goto" - Proceedings of the IFIP Congress, Computer Software, August 1971 - pp.84-88.