A Control Structure for a Variable Number of Nested Loops

E. Skordalakis and G. Papakonstantinou
Computer Center, N.R.C. 'Democritos', Aghia Paraskevi, Athens, Greece

A new program control structure is proposed in this paper which is suitable for expressing a variable number of nested loops. This control structure is useful in combinatorial problems and in problems requiring backtracking. Implementation details are discussed and some illustrative examples are given which employ this new program control structure. The incorporation of this control structure in contemporary programming languages will considerably enhance them, particularly languages like FORTRAN, BASIC, and assembly.

INTRODUCTION

A new control structure which is suitable for programming algorithms having a variable number of nested loops is proposed in this paper. Algorithms employing nested loops of variable depth are found in combinatorial problems and in problems requiring backtracking. Although these algorithms can be programmed using recursion and the conventional control structures, programming them using the new control structure is more convenient and more natural. Furthermore, the implementation of this new control structure can be done easily and it does not require recursion, which means that its incorporation in languages like FORTRAN, BASIC, and assembly will enhance them considerably by allowing them to be used in programming a class of recursive algorithms.

While we agree that the conventional control structures (D- or D'-structures[4]) are sufficient for the practising programmer and that definite evidence is needed before a new control structure is to be adopted, we propose this new control structure in the belief that it meets this requirement.

DESCRIPTION OF THE CONTROL STRUCTURE

In Fig. 1 n nested loops are shown in flowchart form (the symbolism is explained in Fig. 2). The control variable, the starting parameter, the terminal parameter, and the step (incrementation) parameter of the loop k are, respectively, i, b, e, and s (the values of b, e, and s are considered positive integers). The body of the loop k is divided as shown in Fig. 1 into three parts, the 'precode$_k$', the 'loop $k+1$', and the 'postcode$_k$'. This is true for $k = 1(1)n-1$; when $k = n$, instead of 'loop $n+1$' which has no meaning we have 'corecode'. Such a structure is of practical value when 'precode$_k$' and 'postcode$_k$', $k = 1(1)n$, are instances of a parameterized 'precode', 'postcode'. In this case the structure is recursive and therefore it can be coded employing programming languages which provide recursion. The code of the structure is given in Appendix A with Algol 60 as the programming language. However, it can be coded in a way which is more convenient and natural by employing a new control structure, specially devised for this case.

This new control structure, called 'nest structure', in an Algol-like notation is as follows:

```
nest i = b, e, s, n, k;
precode;
endpre;
corecode;
```

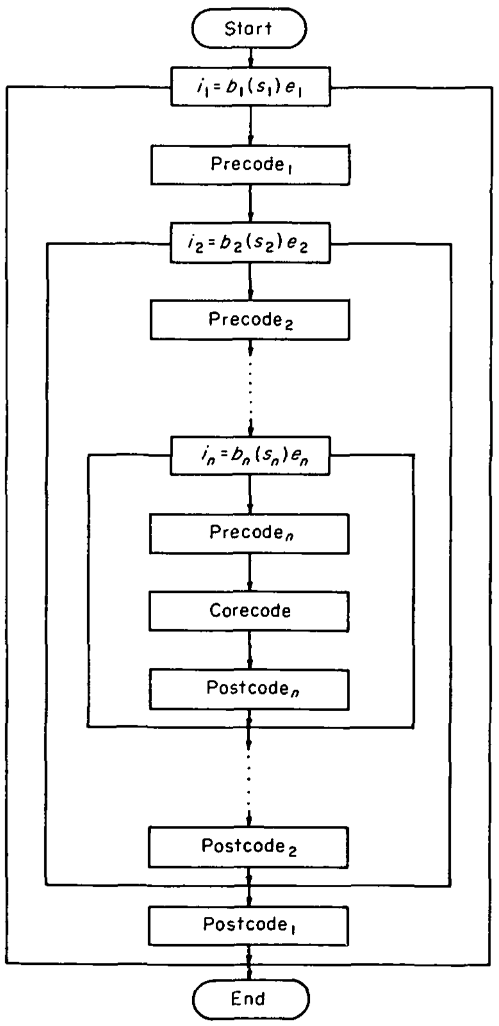

Figure 1. A control structure consisting of nested loops.

```
endcore;
postcode;
endpost;
endnest;
```

where
i, b, e, s are integer arrays of length n
$i[k]$ is the control variable for the k loop
$b[k]$ is the initial parameter for the k loop
$e[k]$ is the terminal parameter of the k loop
$s[k]$ is the step parameter of the k loop
n is an integer variable reference or an integer constant denoting the number of the nested loops
k is an integer variable to which is assigned the current depth of the nesting $(1 \le k \le n)$. It can be used within the range of the nest structure whenever the depth of the nesting is required
'precode', 'corecode', and 'postcode' are sets of instructions.

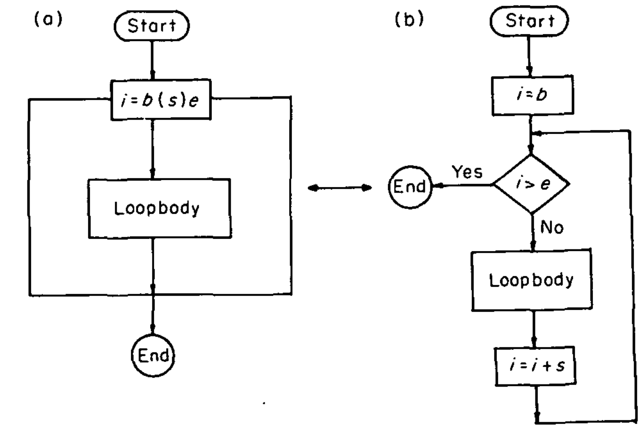

Figure 2. Flowcharts (a) and (b) are considered equivalent.

It is assumed that the arrays b, e, and s are initialized prior to using the nest structure.

This nest structure is associated with two more transfer of control instructions. The first has the format:
nextcycle;
which is interpreted as 'carry on the current loop with the next value of its control variable'.
The second has the format:
exitloop;
which is interpreted as 'exit from the current loop and carry on with the immediately outer loop'.

SOME ILLUSTRATIVE EXAMPLES

There is a class of problems the solution of which can be formulated according to the scheme of Fig. 1. The solution of these problems is a set of vectors $(x_1, x_2, \ldots, x_n)$ from a direct product space $X_1 \times X_2 \times \cdots \times X_n$ with x_i an element of X_i. Each loop i in Fig. 1, $1 \le i \le n$, generates the elements of X_i while the 'precode$_i$', 'postcode$_i$' select the element x_i, and 'corecode' consumes the solution vector $(x_1, x_2, \ldots, x_n)$. $X_1, X_2, \ldots, X_n$ may or may not be copies of one another.

Two illustrative examples are given in this section. The first is a pure combinatorial problem and the second is a search problem which requires backtracking. The second problem was taken from Ref. [5] but its solution is formulated there according to a scheme which is different from the one considered here. Areas in which such problems can be found, together with specific examples, are given in Refs 1–3 and 5.

Example 1. Find all combinations of q elements of a given set of p elements.

In this problem it is required to find all ordered vectors $(i_1, i_2, \ldots, i_q)$ such that $i_{k-1} < i_k$, $k = 2(1)q$
where
$1 \le i_k \le p$, $k = 1(1)q$
i_k stands for a particular element of the given set, the elements of which are somehow mapped into the set $\{1, 2, \ldots, p\}$.

Formulating the solution of this problem according to the scheme of Fig. 1 using nested loops we have q nested loops one for each component of a combination, and the control variable in each loop takes the values $1(1)p$. In other words
$X_i = \{1, 2, \ldots, p\}$ for $i = 1(1)q$, and
$(x_1, x_2, \ldots, x_q) = (i_1, i_2, \ldots, i_q)$. Using the symbolism of Fig. 1, we have

$n = q$
$b_1 = 1$
$e_m = p$, $m = 1(1)n$
$s_m = 1$, $m = 1(1)n$
precode$_k \equiv$ **if** $k < n$ **then** $b_{k+1} = i_k + 1$
corecode $\equiv$ print i_m, $m = 1(1)n$
postcode$_k \equiv$ nil

Example 2. Colouring the vertices of a graph G.

Let G be a graph of p vertices, and let q be a given positive integer. A proper colouring of the vertices of G in q colours is an assignment of a colour i_k $(1 \le i_k \le q)$ to each vertex k, $k = 1(1)p$, in such a way that for each edge e of G the two endpoints of e have different colours. It is required to find all the proper colourings.

In this problem we are actually searching for vectors $(i_1, i_2, \ldots, i_p)$ such that $\bigwedge_{k=1}^{p} (\neg P_k) = \textbf{true}$
where
$1 \le i_k \le q$, $k = 1(1)p$
k = vertex
i_k = colour of vertex k

$$P_k = \begin{cases} \textbf{false} & \textbf{for} \quad k = 1 \\ \bigvee_{j=1}^{k-1} (g_{kj} \wedge i_k = i_j) & \textbf{for} \quad 2 \le k \le p \end{cases}$$

g = adjacency matrix of the graph G.

Formulating the solution of this problem according to the scheme of Fig. 1 using nested loops we have p nested loops one for each vertex of the graph, and the control variable in each loop takes the values $1(1)q$. In other words $X_i = \{1, 2, \ldots, q\}$ for $i = 1(1)p$, and $(x_1, x_2, \ldots, x_p) = (i_1, i_2, \ldots, i_p)$. Using the symbolism of Fig. 1 we have

$n = p$
$b_m = 1$, $m = 1(1)n$
$e_m = q$, $m = 1(1)n$
$s_m = 1$, $m = 1(1)n$

$$\text{precode} \equiv \begin{cases} \textbf{if } k \neq 1 \textbf{ then} \\ \textbf{for } j = 1 \textbf{ step } 1 \textbf{ until } k - 1 \textbf{ do} \\ \textbf{if } g_{kj} \wedge i_k = i_j \textbf{ then nextcycle}; \end{cases}$$

corecode $\equiv$ print i_m, $m = 1(1)n$
postcode $\equiv$ nil

IMPLEMENTATION

The proposed new control structure '**nest**' together with its associated two transfer of control instructions '**nextcycle**' and '**exitloop**' can be incorporated into a host programming language. This can be accomplished in one of the following three ways: (1) by modifying the compiler (or interpreter) of the host programming language so as to cope with the new control structure, (2) by developing a preprocessor which will translate the augmented programming language (host programming language + new control structure) into the host programming language, (3) by implementing the new control structure through subroutine calls.

The first way is the most difficult of all. The second is less difficult than the first, and the third is the easiest. The following observation is important and can be used in any one of these three implementations: the flowchart of Fig. 1 which needs recursion in order to be coded, can be transformed into an equivalent flowchart, the one shown in Fig. 3, which does not need recursion in order to be coded.

Some comments concerning the flowchart of Fig. 3 seem appropriate. It requires the 'precode$_k$', 'corecode', and 'postcode$_k$' to be coded as subroutines. It utilizes a new (global) variable f which controls the flow of control within the same loop. The depth of nesting is explicitly controlled through the manipulation of the variable k.

In what follows in this section an implementation of the third way is presented with FORTRAN as the host programming language.

The control structure '**nest**' is used as a subroutine call according to:

CALL NEST (I, B, E, S, N, K, F, PRE, CORE, POST)

where

I, B, E, S are integer arrays of length N
$I(K)$ acts as the control variable of the loop K
$B(K)$ acts as the starting variable of the loop K
$E(K)$ acts as the terminal variable of the loop K
$S(K)$ acts as the step variable of the loop k
n is an integer variable which denotes the number of the nested loops
k is an integer variable which denotes the current depth of the nesting
F is an integer variable the value of which has to do with the transfer of control instructions '**nextcycle**', and '**exitloop**' as is explained below.
PRE is the name of a subroutine which holds the code for 'precode'
CORE is the name of a subroutine which holds the code for 'corecode'
POST is the name of a subroutine which holds the code for 'postcode'.

The subroutines with names PRE, CORE and POST have seven parameters which are the same (and in the same order) as the seven first parameters of the subroutine NEST.

The subroutine NEST corresponds to the flowchart in Fig. 3 and it is coded once. The subroutines PRE, CORE and POST are prepared by the user each time a control structure with a variable number of nested loops is programmed. Within these subroutines the transfer of control instructions '**nextcycle**' and '**exitloop**' are coded,

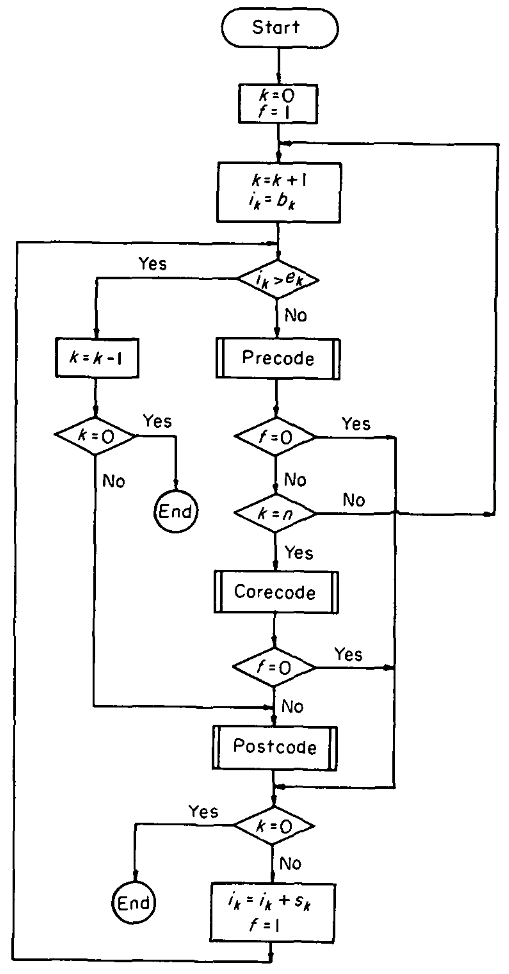

Figure 3. This flowchart is equivalent to the one in Fig. 1.

respectively:

$$\text{'nextcycle'} \rightarrow \begin{cases} F = 0 \\ \text{RETURN} \end{cases}$$

and

$$\text{'exitloop'} \rightarrow \begin{cases} F = 0 \\ I(K) = E(K) \\ \text{RETURN} \end{cases}$$

Besides the information passed to them through their parameters the subroutines PRE, CORE, and POST can share data through labelled COMMON.

The layout of a program part which is equivalent to the nest control structure in this implementation is as follows:

```
C
C  BEGIN OF THE NEST CONTROL
C  STRUCTURE
C
initialize parameters
CALL NEST ( , . . . , )
C
C  END OF THE NEST CONTROL STRUCTURE
C
```

The FORTRAN listing of the second example of Section 3 is given in Appendix B.

Acknowledgment

We would like to thank the referee for his helpful suggestions.

REFERENCES

1. J. Cohen and E. Carton, Non-deterministic FORTRAN, *The Computer Journal* **17** (No. 1), 44–51 (1974).
2. R. W. Floyd, Nondeterministic algorithms, *Journal of the ACM* **14** (No. 4), 636–644 (Oct. 1967).
3. S. W. Golomb and L. D. Baumert, Backtrack programming, *Journal of the ACM* **12** (No 4), 516–524 (Oct. 1965).
4. H. F. Ledgard and M. Marcotty, A genealogy of control structures, *Communications of the ACM* **18** (No. 11), 629–639 (Nov. 1975).
5. A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms*, Academic Press, New York (1975).

Received October 1980

APPENDIX A

```
begin
integer n;
comment initialize n;
initialize n: —
             ⋮
            —;
begin
integer array i, b, s, e [1:n];
procedure nest (k);
integer k;
begin
for i[k] := b[k] step s[k] until e[k] do
  begin
  comment code for precode k;
  precode k: —
              ⋮
             —;
     if k < n then nest (k + 1) else
          begin
          comment code for corecode;
          corecode: —
                      ⋮
                     —;
          end;
          comment code for postcode n;
          postcode n: —
                        ⋮
                       —;
     if k = 1 then goto L1;
     comment code for postcode k;
     postcode k: —
                   ⋮
                  —;
     L1 :;
     end;
end;
comment initialize b, s, e arrays;
initialize b s e: —
                    ⋮
                   —;
nest (1);
end;
end
```

APPENDIX B

```
C
C       COLORING THE VERTICES OF A GRAPH
C       USING THE 'NEST' CONTROL STRUCTURE
C
        DIMENSION I(30),B(30),E(30),S(30)
        INTEGER B,E,S,P,Q,F
        LOGICAL G
        EXTERNAL PRE2,CORE2,POST2
        COMMON/BNEST/ G(30,30)
C
C       P=NUMBER OF VERTICES, Q=NUMBER OF COLORS
C
        READ(60,10) P,Q
10      FORMAT(2I2)
        IF(P.LE.30) GO TO 30
        WRITE (61,20)
20      FORMAT(' GRAPH HAS MORE THAN 30 VERTICES PROGRAM STOPS')
        STOP
30      DO 35 M=1,P
        READ(60,40) (G(M,J),J=1,P)
35      CONTINUE
40      FORMAT(30L1)
C
C       BEGIN OF NEST CONTROL STRUCTURE
C
        N=P
        DO 50 M=1,N
        B(M)=1
        E(M)=Q
50      S(M)=1
        CALL NEST(I,B,E,S,N,K,F,PRE2,CORE2,POST2)
C
C       END OF NEST CONTROL STRUCTURE
C
        STOP
        END
        SUBROUTINE  PRE2(I,B,E,S,N,K,F)
        DIMENSION I(N),B(N),E(N),S(N)
        INTEGER B,E,S,F
        COMMON/BNEST/ G(30,30)
        LOGICAL G
        IF(K.EQ.1) RETURN
        K1=K-1
        DO 10 M=1,K1
        IF(G(K,M).AND.I(K).EQ.I(M)) GO TO 20
10      CONTINUE
        RETURN
20      F=0
        RETURN
        END
        SUBROUTINE CORE2(I,B,E,S,N,K,F)
        DIMENSION I(N),B(N),E(N),S(N)
        INTEGER B,E,S,F
        WRITE(61,10)  (I(M),M=1,N)
10      FORMAT(1H ,30(I2,2X))
        RETURN
        END
        SUBROUTINE POST2(I,B,E,S,N,K,F)
        DIMENSION I(N),B(N),E(N),S(N)
        INTEGER B,E,S,F
        RETURN
        END
        SUBROUTINE NEST(I,B,E,S,N,K,F,PRE,CORE,POST)
        DIMENSION I(N),B(N),E(N),S(N)
        INTEGER B,E,S,F
        F=1
        K=0
10      K=K+1
        I(K)=B(K)
20      IF(I(K).GT.E(K)) GO TO 50
        CALL PRE(I,B,E,S,N,K,F)
        IF(F.EQ.0) GO TO 40
        IF(K.NE.N) GO TO 10
        CALL CORE(I,B,E,S,N,K,F)
        IF(F.EQ.0) GO TO 40
30      CALL POST(I,B,E,S,N,K,F)
40      IF(K.EQ.0) RETURN
        I(K)=I(K)+S(K)
        F=1
        GO TO 20
50      K=K-1
        IF(K.EQ.0) RETURN
        GO TO 30
        END
```