On the use of macros for iteration

A. G. Middleton

Department of Computer Science, University College of Swansea, Singleton Park, Swansea, SA2 8PP, Wales

Iterative constructs are often associated with a stream of data items (a 'data path'). This paper shows how macros can be used to allow the programmer, in certain circumstances, to request iteration in such a manner that the data path utilised is made explicit. The resulting program text is considered more natural and informative than that which might result from a straightforward application of 'repeat'-constructs, 'for'-constructs, etc.

(Received November 1974)

General approach

The problem

The techniques discussed are considered to fall in the general area of 'structured programming'. However, (see Denning, 1973 and Gries, 1974) the term 'structured programming' has no precise, universally accepted definition. In this paper, the term will be used to refer to the use of source constructs of the type discussed by Dijkstra (see Dahl *et al*, 1972) to build up a program from a small set of readily understood control structures: these tools possibly being used to develop a program by stepwise refinement (Wirth, 1971).

Such techniques will be referred to as 'conventional' structured programming, as they have influenced current language design (for example, Woodward and Bond, 1974 and Jensen and Wirth, 1974) and are at the centre of much debate on structured programming (for example, Dijkstra, 1968 and Wulf, 1972). It is felt that this 'conventional' approach does not always convey an immediate grasp of what is happening in terms of accessing data structures.

This paper discusses the use of macros to permit the source constructs in a structured program to convey readily the nature of the data path accessed in certain iterations.

All iterative constructs implemented take the form:

```
FOR x IN dp:
a
ENDFOR
```

where:

x—is used to denote a representative item from the data path.

dp—is the data path associated with the iteration.

a—is the action to be performed each time an item is accessed from the data path.

A comment on notation

The macros discussed were implemented in the programming language POP-2 (Burstall and others, 1971). This is not a widely used language, so one or two minor modifications were introduced to convenience the general reader. In particular, the symbols VARIABLES, HEAD, TAIL, PRINT and NEWLINE were used instead of the less obvious VARS, HD, TL, PR and NL.

Examples

Frequently, in discussing iteration, the programmer will use such phrases as 'take a row at a time' or 'for all the items in the list *L*'. This suggests that the programmer finds it natural to discuss iteration in terms of data paths. The conventional structured programming approach does not make this aspect of iteration immediately apparent.

Consider the following code for printing a list, *L*, at the top level:

```
PRINT("[");
WHILE NOT(NULL(L))
DO    PRINT(HEAD(L));
      TAIL(L) → L;
ENDWHILE
PRINT("]");
```

Use of macros permits the following, alternative code:

```
PRINT("[");
FOR X IN LIST L:
      PRINT(X)
ENDFOR
PRINT("]");
```

It is felt that the second variation conveys the intent of the construction much more clearly than the straightforward use of the WHILE-construct.

The code actually generated in this implementation is:

```
      L → v1;
lab1: IF NOT(/NULL(v1)) THEN
      HEAD(v1) → X;
      PRINT(X);
      TAIL(v1) → v1;
      GOTO lab1 CLOSE;
```

where lab_1 is a unique label generated by the macro and v_1 is used to save the original list value—in case the construct should be repeatedly executed.

There is nothing new about using macros to implement structured programming (see Barron, 1974 and Leavenworth, 1966). Nor does the author's approach differ radically from recent developments. However, it is felt that an important, and useful, change of emphasis is introduced. Using the conventional approach, the programmer must think both about the data path involved in the iteration and about the program control mechanism he uses to effect access to this data path. Using the alternative approach, he need only think about the data path.

Another example emphasises the point. Frequently, a programmer wishes to operate on diagonals. This can be done in the following manner:

```
0 → TOTAL;
FOR X IN DIAGONAL I J OF A [M N]:
      X + TOTAL → TOTAL
ENDFOR
```

This would sum the elements of the *I*, *J*th diagonal of an array *A* (diagonals are taken to run 'downwards, from left to right').

The intermediate code generated is of the form:

```
0 → TOTAL;
IF I = 1 THEN N − J
  ELSE M − I CLOSE → v1;
FOR v2 IN INTERVAL 0,v1,1:
TOTAL + A(I + v2, J + v2) → TOTAL
ENDFOR
```

(the notation: FOR v IN INTERVAL start, stop, step: is used instead of the ALGOL:
For v := start **Step** step **Until** stop)

It is not at all obvious in the second piece of code that a diagonal is being accessed.

Suppose we wish to add 1 to each element in an $M \times N \times P$ array, A. This could be specified so:

```
FOR X IN ARRAY A [M N P]:
    X + 1 → X
ENDFOR
```

The code generated would have the form:

```
FOR v1 IN INTERVAL 1,M,1:
FOR v2 IN INTERVAL 1,N,1:
FOR v3 IN INTERVAL 1,P,1:
A(v1,v2,v3) + 1 → A(v1,v2,v3)
ENDFOR
ENDFOR
ENDFOR
```

where v_1, v_2 and v_3 are distinct identifiers generated by the macro. (An approach could have been used in which it was unnecessary to declare the array bounds if they were declared elsewhere—however, this is not essential to the main arguments presented here.)

Implementation

In POP-2, a macro can be regarded as a parameterless function which is triggered by the occurrence of its identifier in the input stream. The macro can use all the facilities available to any other function in POP-2. Thus, in POP-2, macros have great generality. It may, therefore, not be clear how readily such macros may be implemented in some other macro-processing system. To alleviate this problem, an explanation will be given of the typical transformations involved and the reader may then adjudicate on this point.

The following macro-expansion requires the most involved mechanism:

```
FOR X IN ROWS OF A [M N]
NEWLINE(1);
        FOR Y IN X;
            PRINT(Y)
        ENDFOR
ENDFOR
```

This code would cause an array to be printed row by row, each row being printed on a new line. The code generated is:

```
FOR v1 IN INTERVAL 1,M,1:
NEWLINE(1);
        FOR v2 IN INTERVAL 1,N,1:
        PRINT (A(v1,v2))
        ENDFOR
ENDFOR
```

The FOR macro expects the sequence:

FOR *x* IN *keyword*

x and *keyword* are saved and *keyword* is tested to determine the nature of the expansion. *x* is maintained in a push-down list to allow for nested macros. If the value of *keyword* is 'ROWS', the result of the macro is:

ROWSMAC

which is the name of a macro to deal with iteration by rows. When this macro is triggered, the flag FORFLAG is set to TRUE. This flag is tested by FOR and will cause that macro to produce the result:

FORWORD

when set to TRUE. (The purpose will become apparent later.) ROWSMAC scans ahead, expecting the pattern:

OF *arrayname* [*dimensions*]:

The *arrayname* and *dimensions* are saved, ROWSMAC then

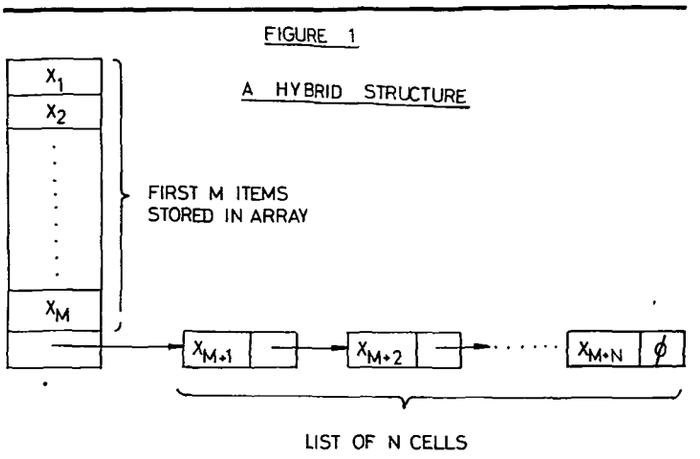

Fig. 1 A hybrid structure

scans ahead, regarding FORWORD as a left bracket, accumulating in a list (call it t) all intermediate text, until the matching occurrence of ENDFOR is found. (Note that the use of FORFLAG inhibits the expansion of FOR-macros until t is processed as below.) The list t is then searched for occurrences of the pattern:

```
FORWORD y IN x:
t'
ENDFOR
```

If such a pattern is found, a new version of t is generated in which this pattern is replaced by:

```
FORWORD y IN INTERVAL 1,N,1:
t''
ENDFOR
```

where t'' is a version of t' in which all occurrences of y are replaced by:

arrayname(x, y) .

Then all occurrences, in t, of FORWORD are replaced by FOR, the flag FORFLAG is switched off, and the final macro expansion is

```
FOR x IN INTERVAL 1,M,1:
t
ENDFOR
```

The main features required for macro-expansion are iteration, manipulation of lists of symbols, reading and testing the input stream, assignment to macro-time variables and conditional branching.

A more unusual example

The technique may prove especially useful in an application where unusual data structures are required. One example will be given. Suppose a data structure of the type shown in **Fig. 1** is used. This is a 'hybrid' structure. The first M items are stored in a vector of length $M + 1$. The last element of this vector points to a list of the remaining items. Such a data structure might be desirable in a program in which a mixture of both of the following operations was being performed on a data structure:

(*a*) Random access, using a numeric index.

(*b*) Appending items to the end of the data structure.

(Such a choice offers a compromise between the characteristics of a list and those of a vector—periodically the structure might be reorganised to increase the number of items which can be conveniently accessed randomly.)

A macro can be provided for iterating on such a structure, so that to, say, sum all the items in the structure, one would write:

```
0 → TOTAL;
FOR X IN HYBRID H:
X + TOTAL → TOTAL
ENDFOR
```

(The code generated is not given—it is left to the reader to realise that it is not particularly attractive.)

Summary

What does, and what does not, constitute 'convenience' in programming languages is somewhat a question of individual choice. The author feels that coding for iteration is more easily understood if, whenever a data path is involved, the nature of the data path is immediately apparent.

Like many language features, the above facility is decidedly no 'cure-all', but its use appears advantageous for a certain range of commonly occurring situations.

The techniques presented here are only suitable for iterations involving a single data path. This is quite a serious limitation (e.g. how does one handle element-by-element assignment?). More general techniques are being developed to handle such cases (this appears to involve the manipulation of text which represents program actions having multiple entries and/or multiple exits—and the use of 'connector functions' to combine these components in a more general manner than is allowed in 'conventional' structured programming—however that is a matter for further research).

One would expect the above techniques to produce (source) program code which is relatively insensitive to choices of data structure. This may allow convenient implementation of decisions concerning the mapping of abstract data structures to physical data structures when implementing very high level languages. However, this is again a topic of further research.

The author would like to thank the referee for useful suggestions on improving the presentation of this paper.

References

BARRON, D. W. (1974). APL and POP-2: What can we learn from Interactive Languages?, *High Level Languages—The Way Ahead*, BCS Conference Proceedings.

BURSTALL, R. M., COLLINS, J. S., and POPPLESTONE, R. J. (1971). *Programming in POP-2*, Edinburgh University Press.

DAHL, O. J., DIJKSTRA, E. W., and HOARE, C. A. R. (1972). *Structured Programming*, Academic Press.

DENNING, P. J. (1973). Letter to the Editor, *SIGPLAN Notices*, October 1973.

DIJKSTRA, E. W. (1968). Go To Statement Considered Harmful, *CACM*, Vol. 11, No. 3, pp. 147-148.

GRIES, D. (1974). Letter to the Editor, *CACM*, Vol. 17, No. 11, pp. 655-657.

JENSEN, K., and WIRTH, N. (1974). PASCAL—User Manual and Report, *Lecture Notes in Computer Science*, Springer-Verlag.

LEAVENWORTH, B. M. (1966). Syntax Macros and Extended Translation, *CACM*, Vol. 9, No. 11, pp. 790-793.

WIRTH, N. (1971). Program Development by Stepwise Refinement, *CACM*, Vol. 14, No. 4, pp. 221-227.

WOODWARD, P. M., and BOND, S. G. (1974). *ALGOL 68-R Users Guide*, HMSO.

WULF, W. A. (1972). A Case Against the GOTO, Proceedings of ACM National Conference, Boston, pp. 63-69.

Book reviews

Computer Science and Technology and their Application, General Editors: N. Metropolis, E. Piore and S. Ulam, 1975; 310 pages. Administrative Editors: Mark I. Halpern, William C. McGee; Contributing Editors: Louis Bolliet, Andrei P. Ershov, J. P. Laski. (*Pergamon Press*, £15·00)

Contents

A Tutorial on Data-Base Organization, R. W. Engles.
General Concepts of the Simula 67 Programming Language, J. D. Ichbiah and S. P. Morse.
Incremental Compilation and Conversational Interpretation, M. Berthaud and M. Griffiths.
Dynamic Syntax: A Concept for the Definition of the Syntax of Programming Languages, K. V. Hanford and C. B. Jones.
An Introduction to ALGOL 68, H. Bekic.
A General Purpose Conversational System for Graphical Programming, O. Lecarme.
Automatic Theorem Proving Based on Resolution, A Pirotte.
A Survey of Extensible Programming Languages, N. Solntseff and A. Yezerski.

It appears immediately that this volume is not annual, nor is it a review; it is hardly automatic programming, and the contributing editors did not contribute. Nevertheless, it is a selection of articles on topics closely related to high-level programming languages. They might have been contributed to a learned journal; but instead they have been collected in a book. On the whole they deserve to be: the general standard of the papers is distinctly higher than the average, and they are likely to appeal more consistently to a reader interested in high-level programming languages.

But it would be a rash reviewer who would venture to pass comment on all the papers individually; and to do so within the space allotted would be even more foolhardy.

C. A. R. HOARE (Belfast)

Computer Science: Programming in FORTRAN IV with WATFOR WATFIV, 210 pages. (*John Wiley and Sons*, £2·65)

The very successful *Computer Science: A First Course* has now been re-issued in a second, considerably expanded, edition which necessitates a similar revision of all the language supplements. This, the FORTRAN one, is the first that I have seen of the second editions. There are some changes in the set of authors and it is now in phototypescript rather than print, but the overall impression of workmanlike competence remains.

C. M. REEVES (Keele)

Environmental Data Handling, by G. B. Heaslip, 1975; 203 pages. (*John Wiley*, £12·25)

Environmental Data Handling is not a book about computing but about the way transducers and sensors work, how their outputs are generated, coded, transmitted, recorded, analysed and presented. It is simple in its approach with many a homely (in the British sense) line illustration to carry home a point. Remote sensing as practised in Earth resources programmes provides much of the inspiration to the author's approach, befitting his background in the NASA Lunar Program and the Grumman Environmental Data Services. Practical experience is evident in every page and the approach is not analytic. Transducers are only very simply described; the frequency response of an accelerometer, for example, is not discussed. Again the filtering of data at the demodulation stage, an all-important and technically interesting operation, is treated at an elementary level. The book will be a useful reference to those unfamiliar with the subject.

E. B. DORLING (Dorking)