The efficient administration of blocks in ALGOL

By P. A. Samet*

A scheme for administration of ALGOL blocks is proposed, based on the use of block numbers rather than block levels. It is claimed that this method simplifies the organization of procedure calls, including recursive calls.

The block structure of ALGOL, with its nested levels of nomenclature, is invaluable to the programmer but gives rise to difficulties in implementation. Briefly, the most serious troubles stem from the fact that reference must only be made to blocks with currently valid declarations. Within a procedure, and in particular a recursive procedure, it is not always easy to discover which blocks are active. As far as the procedure body is concerned the valid declarations are those valid at the time the procedure is declared, whereas the declarations valid for the actual parameters are those valid at the time of the procedure call. Particularly troublesome are **go to** statements leading out of the procedure, especially if used recursively or if there is recursion involving several procedures, because then it is necessary to reach the labelled statement in the correct activation of its block as well as restoring the declarations for the appropriate activations of the enclosing blocks.

The only well-known method of achieving this reactivation is by means of block levels, where an explicit record is kept of the levels of nomenclature current at any one time. This is the method used by Dijkstra (1961), and also by Randell and Russell (1964) a preliminary account of whose work is given by Randell. The need to update this record, called DISPLAY by Randell and Russell, every time a block or procedure is entered or left makes this extremely time-consuming. The method of van der Mey (1962), using ranks, is basically similar and has the same disadvantage.

In this paper I propose an alternative method of achieving the correct reactivation without using block levels. The method should therefore be an efficient way of controlling block structure. Two ideas are involved: the use of 'block numbers', and the storage of some administrative information in the block itself instead of in a stack.

(With few exceptions, details of compiler techniques are guarded like the secrets of a primitive religion; the method proposed may be known to a restricted group of initiates, but I have not met anyone familiar with it.)

Block level and block number: translation

At translation time it is very easy to keep a record of how many levels of nomenclature are current at any point in the program, as this is simply the number of **begin**s (followed by declarations) not yet cancelled by their matching **end**s. If a stack is used for the declarations it is in fact unnecessary to use these block levels explicitly during translation. This follows because the only declarations available when an identifier is being translated are the declarations of the enclosing blocks, which are of course the only declarations valid. The declaration list has to be scanned in reverse order to find the most recent declaration of an identifier, in case of clashes of names.

However, just as easy to determine as block levels are actual block numbers, which are obtained by a simple count of declarations. This block number can be used, as explained below, to determine the block to which a label refers. Identifiers other than labels need only refer to a place in the stack relative to the stack pointer current at entry to the block in which they are declared. The block number as defined here is different from the block number (BN) used by Dijkstra (1961), which is rather more akin to what I call block level.

At present the only use of block numbers that appears to be made by translators is indicating the location of errors.

Block level and block number: processing

The system of translation indicated above ensures that block levels are irrelevant at run-time, as references can only be made to those identifiers that are actually allowed. The problem is to give easy access to the locations holding variables, which can be done if we know how to reach the appropriate value of a stack pointer. The schemes described by Dijkstra, and Randell and Russell all have a vector of stack pointer values, corresponding to the block levels in use.

It is, however, more advantageous to store these block stack pointers in the blocks to which they belong. References to simple variables and arrays are translated to include a call for the relevant stack pointer from a particular (fixed) location, which can be implemented very conveniently with indirect addressing.

The administration of recursion now becomes an almost trivial matter. All that is necessary is that at entry to a block the block stack pointer corresponding to the previous activation of the block is stored away safely in the stack before being replaced by the value

**Director, Computation Laboratory, The University, Southampton.*

corresponding to the new activation. It will also be necessary to store the address within the block where the stack pointer is held, for use at block exit. (I am not concerned here with the link data needed for procedure exit, as this has nothing to do with recursion.) The only limit on the depth of recursion (or number of block levels) is now dictated by the store available, not by the arbitrary number of places reserved for the vector of levels in use. At exit from a block, the stack pointer of the previous activation is restored. It will be noticed that there is now no mention at all of the block level at which an identifier is declared.

In some implementations of ALGOL considerable difficulty is experienced in relating the static form of the program, as written, with the dynamic form, as obeyed. Randell and Russell, for instance, do this by 'static' and 'dynamic chains' which have to be continually updated and matched. The scheme proposed in this paper avoids this problem completely. At run-time it is the written form of the program that contains all the information about currently active blocks and procedures, whereas everything about dormant blocks is kept in the stack.

So far, nothing has been said about labels. It is here that the actual block number can be used. A word in the block is reserved for the block number as well as for the block stack pointer. A label has to be translated into an address in any case, and if we translate a label into the pair (*address, block number*) the processor can easily determine whether the label's block number is the number of the current block. If it is, the **go to** statement is just a jump. If, however, the **go to** statement involves an unnatural block exit we have to restore the stack pointer of the previous activation of the current block, and examine the block number of the block in which the current block was called. This means that the system of chaining in the stack has to include information about the stack pointer of the immediately enclosing block (of a block or procedure call). This information is required for normal block exit, in any case, so that nothing extra is required. By going back along the chain in this manner, always restoring the stack pointers of previous activations, we must arrive at the most recent activation of the required block, and at the same time restore the declarations of the surrounding blocks. There is no need whatever to count the depth of any recursion involved, as in some implementations of ALGOL.

An alternative, equally simple, system has been suggested to me by one of my colleagues, A. R. F. Reddaway. To effect the restoration of block stack pointers it is necessary to know where these are to be stored in their respective blocks. This address can be used in the label, which is then compiled as the pair

(*address of labelled statement,*
address of block stack pointer).

We have found the right block when the stack pointer comes from the place indicated. The advantage is that this address has to be used in any case for restoring the previous activation. Block numbers, however, may be easier to manipulate, especially if the program has to be segmented.

It may be advantageous to single out the labels of the main program, which do not occur inside a procedure. These cannot be involved in any recursion and so a jump to one of them can be made without having to restore stack pointers of intervening blocks.

General comment on labels

For a compiler that is to allow easy alterations of compiled programs without the need to recompile the whole source program, it is probably helpful if labels are translated so as to give the number of the label in its own block. The effect is that all labels behave as if they are elements of a switch list.

Label as formal parameter of a procedure

There is one case in which the straightforward application of the system proposed would give incorrect behaviour. If a recursive procedure has a label as formal parameter it is possible for the actual parameter of an inner call to be a label of the procedure itself, as in the following example:

```
procedure P (alpha); label alpha;
begin     .
          .
    P (omega);
          .
          .
          .
     go to alpha;
omega:
          .
          .
          .
end.
```

The dots stand for parts of the procedure, including conditional statements which can by-pass the inner call of *P*. At the inner call of *P* the **go to** *alpha* has become, in effect, **go to** *omega* and should cause an exit from the inner call of *P*. Simple examination of the block number would fail to do this. The remedy, however, is very simple. The object program of a formal parameter which is a label has to perform one exit from the procedure (i.e. one restoration of block stack pointer, etc.) before examining the block number.

Further remarks

The system I have described has some similarities with the method used by Irons and Feurzeig (1961), but is more flexible, as there is no need for a block to occupy consecutive locations in the store. This is relevant when a compiled program has to be amended, as corrections to an inner block can be joined on at the end

without having to recompile the outer blocks. (The only corrections to which this method can be applied involve no changes in declarations or labels, although labels can be accommodated by the suggestion above, of treating them like a switch list.)

Acknowledgements

I am grateful to my colleagues, A. R. F. Reddaway and P. J. Taylor, for helpful suggestions. Some obscurities in the original manuscript have been clarified at the suggestion of a referee.

References

DIJKSTRA, E. W. (1961). "An ALGOL Translator for the X1," *MTW*, Vol. 2, p. 54–6; and *MTW* Vol. 3, pp. 115–9. (See also *Annual Review of Automatic Programming*, No. 3, 1963, p. 329.)

IRONS, E. T., and FEURZEIG, W. (1961). "Comments on the Implementation of Recursive Procedures and Blocks in ALGOL 60," *Comm. A.C.M.*, Vol. 4, No. 1, pp. 65–9.

RANDELL, B., and RUSSELL, L. J. (1964). *ALGOL 60 Implementation*, Academic Press, London.

RANDELL, B. (1964). "The Whetstone KDF 9 ALGOL Translator," Chap. 8 of *Introduction to System Programming*, Academic Press, London.

VAN DER MEY, G. (1962). "Process for an ALGOL Translator," Report 164 MA, Dr. Neher Laboratorium, PTT, Leidshendam, Holland.

Book Review

Extended Mercury Autocode (EMA) for I.C.T. Atlas and Orion Computers, I.C.T. Ltd., 68 Newman Street, London, W.1 (199 + liv pages) 25s.

Readers and subscribers to this *Journal* who received the first two volumes will remember the original contributions by Mr. R. A. Brooker and others of Manchester University in 1957/59 to the development of autocodes for the Ferranti MK 1 and Mercury computers. Further contributions have been made by other Mercury users particularly at CERN Geneva, AERE Harwell, University of London, R.A.E. Farnborough, and the Ferranti programming development group taken over by I.C.T. in 1963. EMA is thus possibly in wider use in European scientific computing centres and industrial research establishments than many other programming languages. As a language which is problem-oriented it can be taught to G.C.E.-A level students in three days: programs are written in algebra, without the stilted English of certain other programming languages. It preserves sufficient contact with the organization of a computer with two-level store and magnetic-tape backing, to form a good fundamental course for selection of programmers to be trained in basic machine code, in which case the EMA course can usefully be extended to occupy five days with exercises.

Compilers are already available for Mercury, Orion, Atlas I and I.C.T. 1101/1301 computers and will be made available for the I.C.T. 1900 series: for the smaller machine configurations, facilities are restricted, but on the larger machines there is available 5/7-track paper-tape input and output, magnetic-tape backing store, lineprinter output and punched-card input. With a tape-editing set costing between £700 and £2500, any research, engineering or statistical department can prepare its own program and data tapes, hire time on an Atlas or Orion prepared to offer a 24-hour turnround, and receive back its results in tape form making multiple copies locally. Many such centres have Telex or other data-transmission facilities. Thus EMA is already a live and fully-developed European programming language, useful also for basic training in computer work. I.C.T. (the publishers) are prepared to offer the manual at a discount for use on approved programming courses.

The manual consists of seven sections. The first contains an introduction to digital computers, program layout, arithmetic operations with floating-point variables and integers, jumps and loops, input and output orders, text output, and the use of subroutines. Section 2 deals with more advanced subroutines and routines containing several chapters, dumping of integers and variables in backing store, manipulation of characters and tables, selection and relinquishing of input/output channels on multi-programmed installations, and the use of magnetic tape. The third section deals with such specialized features as integration of differential equations, the more common matrix operations, generation of pseudo-random numbers (normal or rectangular distributions), complex arithmetic, double-precision arithmetic and logical operations (e.g. counting bits and masking out packed data).

Section 4 covers preparation and development of programs, layout of input documents and paper-tape input/output or line printer. The *load and go* and the *compiling/running* alternatives are outlined and the monitoring facilities are described: these are so extensive that a program usually runs at the second attempt if not the first, producing some results, even if these fall short of or exceed the programmers expectations! Query and trace printing are available for following the path of calculations which misfire.

Sections 5 and 6 are devoted to the use of Atlas and Orion, including the incorporation of basic machine-code routines and other facilities peculiar to each computer. The Autocode List Processing facilities, described by D. C. Cooper and H. Whitfield in 1962 for the CHLF 3 version of Mercury Autocode (*Computer Journal*, Vol. 5, pp. 28–32), are available on Atlas. On Orion, programs can be compiled in parts and extra chapters incorporated: this has been found to be a useful feature in certain statistical programs, e.g. regression analysis, where the development of a comprehensive procedure naturally falls into several stages (chapters).

Section 7 and the Appendices include an alphabetical check list of permissible order formats, summarised programming information and an index. At a time when there is so much ill-informed or misguided newspaper comment about software facilities not being available for British computers on time, it is pleasing to report that the compilers for EMA on the machines described in this manual are fully developed, subject only to minor diversions from mispunched tapes, which are being methodically cleared.

H. W. GEARING