FILE COPY

CERN - DATA HANDLING DIVISION
DD/74/6
C.T. Zahn
March 1974

A CONTROL STATEMENT FOR
NATURAL TOP-DOWN STRUCTURED PROGRAMMING

(Presented at "Colloque sur la Programmation", Paris
9-11 avril 1974.)

A CONTROL STATEMENT
FOR NATURAL TOP-DOWN STRUCTURED PROGRAMMING

Charles T. Zahn, Jr.*
European Organization for Nuclear Research (CERN)
Geneva, Switzerland.

ABSTRACT

In recent years there has been a great deal of discussion concerned with the advantages of a top-down structured style of programming and the question of whether or not to banish the goto control statement. The most compelling arguments for the retention of goto have referred to common programming situations where the absence of goto causes some awkwardness in the programming task or loss of efficiency for the running program. We propose a new control statement which alleviates many of these known difficulties while maintaining the philosophy of structured control. The new statement is well-matched to top-down programming and in some respects allows program modifications to be performed more easily.

INTRODUCTION

Since Dijkstra's famous letter to the editor of the ACM Communications [DIJKSTRA 1968] there has been considerable serious discussion of how to write programs without use of goto statements. The Notes on Structured Programming [DIJKSTRA 1972] argue persuasively for a constrained use of program control limited to control statements whose flow charts have one entry and one exit; this leads to decomposable programs whose correctness can be established by a series of arguments each one concerning a small piece of program. There is considerable agreement that if B then S₁ else S₂, while B do S, repeat S until B, the case statement, a simple iterative for statement, recursive procedures and normal program sequence are "safe" for structured programming. There has been, nevertheless, among the advocates of structured programming a considerable controversy over the advisability of actually eliminating the goto control statement from higher-level languages. An entire session of a recent ACM conference [LEAVENWORTH 1972, HOPKINS 1972, WULF 1972] was devoted to this topic and the short paper [LEAVENWORTH 1972] gives a good summary of the relevant issues as well as a large bibliography on the goto controversy. The arguments for retaining goto reveal several fairly common situations where programming without goto is awkward and leads to less readable or less efficient programs than would be obtained if an occasional goto were permitted. We have noted four classes of such situations:

- 1) Multiple (textually) terminations of a repetition
- 2) Imbedded termination of a repetition
- 3) Decision trees with duplicated nodes
- 4) Multi-level error-handling

Searching a table sequentially for the first instance of a given value is the most typical example of a need to terminate a repetition in two distinct ways one an "early" termination of the repetition, the other the "normal" termination. The importance of this example is indicated by the attention given to it by several authors [HOPKINS 1972, KNUTH and FLOYD 1971, WULF 1971, BOCHMANN 1973].

* Present address: Stanford Linear Accelerator Center (SLAC), Computation Group,
P.O. Box 4349, Stanford California 94305, USA.

FIGURE 1

FIGURE 2

Besides the need for multiple terminations there is often a need to terminate from inside the compound statement which is the body of the repetition. This is discussed in [WIRTH 1967, KNUTH and FLOYD 1971] and more recently in [KNUTH 1973]. There are frequent needs for a repetition like the following suggestion in [WIRTH 1967]:

repeat begin S_1 ; when B exit; S_2 end which generalizes the more familiar while and repeat statements.

An example of the third type of difficulty is indicated by a flowchart (figure 1) of the clean enough looking statement:

if (A and B) or (C and not D) then S_1 else S_2

Although the entire flowchart has one entry and one exit, the decision part has 4 exits of two distinct types and within the decision part the two paths into the C test break all the rules of structured programming. A similar problem is mentioned in [WEGNER 1973].

The price for cleaning up this flowchart is replication of the C and D tests as well as 2 extra copies of S_1 and 3 extra copies of S_2 ; either that or making these all procedures. Both these solutions adversely affect efficiency and one is left searching for an alternative. Such an alternative begins to present itself if we embellish the flowchart somewhat and decompose it as shown in figure 2. The new flowchart not only provides documentation for why this arrangement of tests is correct; it also reveals that the need for multiple outcomes is not restricted to repetition.

The fourth type of difficulty exhibits the need for "escapes" from more than a single level of nesting of control statements (including procedure invocations) to handle errors of different levels of severity. Some errors require immediate termination of the main program while others require abnormal termination of some smaller context without causing global confusion. Careful programming requires that such error conditions be passed methodically up the chain of procedure invocations by some case analysis directly after each procedure invocation. This is very tedious and taxes the patience of the most well-intentioned programmer. This problem is well-known to most programmers and was discussed in the context of structured programming

at a recent SIGPLAN meeting [LISKOV 1973]

A number of solutions or partial solutions have been suggested to reconcile these real programming problems with the requirements of structured control. We have already mentioned Wirth's proposal for a general repeat statement with an imbedded exit. Knuth and Floyd mention an extension of Wirth's proposal to allow multiple imbedded exits and they show how it can be used to program the table search. Hopkins uses the return statement of PL/1 to accomplish the same effect. The language BLISS [WULF et al. 1971] excluded goto and labels in favour of single-level exits from any control environment; after some experience was gained using the language these exits were replaced by a potentially multi-level labelled exit returning a value [WULF 1971, 1972]. The syntax is leave L with E where L is the label on some block enclosing the leave statement and E is the returned expression. [CLINT and HOARE 1972] recognizes the need for return-jumps and exit-jumps and proposes that labels should have bodies and be declared at the top of a block. At termination of the label body the enclosing block is implicitly terminated. Such jumps had been proposed earlier [LANDIN 1965] in a slightly different form. This proposal recognizes the usual requirement to perform some post-processing associated with the particular exit. [BOCHMANN 1973] has more recently proposed multiple abnormal exits from a repetition or procedure with associated case-analysis.

AN EVENT DRIVEN CASE-STATEMENT

The control statement proposed in this paper was motivated by a desire to cater for the real difficulties discussed above while retaining the advantages of carefully constrained program control. Another absolutely crucial consideration was the naturalness of the control statement for a top-down style of programming with levels of abstraction [DIJKSTRA 1972]. The readability and modifiability of programs was also considered very important.

After some experimentation we finally settled on what might be called an event-driven case statement. It has a flowchart representation (figure 3) which shows a strong similarity to if B then S_1 else S_2 .

The syntax for this control statement is

```
until Ev1 or Ev2 or...or Evn do
  S0
then case
  Ev1 : S1
  Ev2 : S2
  ⋮
  Evn : Sn
```

where the S_k are arbitrary compound statements and the Ev_k are event-identifiers. Indentation is used to convey nesting and grouping of statements. The semantics of the statement consist of a case-determination statement S_0 which terminates via the occurrence of one of the specified events Ev_k and a subsequent case-analysis statement driven by the particular Ev_k responsible for terminating S_0 . Inside the compound statement S_0 an event-statement (indicated simply by identifier Ev_k) causes the named event to occur and statement S_0 to be immediately terminated. Any termination of S_0 other than by an event-statement is illegal. The entire until...then case statement can be thought of as a generalized if...then...else statement whose test part S_0 has multiple outcomes not restricted to exactly two (see figure 3).

Table Search

It is instructive to see how the table search problem considered by [KNUTH and FLOYD 1971, HOPKINS 1972, WULF 1971] can be programmed top-down using the until...then case statement. The problem is to find the first occurrence of x in a sequential table A and increment a counter variable associated with the matching table entry. If no match exists then a new table entry is to be created for x and the associated counter variable initialized to 1. We can immediately write the program

```
until Table_match or No_match do
  Search_table_for_x
then case
  Table_match : Increment_count
  No_match : Create_new_entry
```

whose correctness depends only on the correct expansion of the three statements Search_table_for_x, Increment_count and Create_new_entry. It is of some interest that the present program says nothing about the structure of the table A and also leaves open the details of how the counter variables are to be implemented. Hence the program can be further elaborated to perform the table search on a linked-allocation or hashed or array data structure. This is a simple case of "judicious postponement" [DIJKSTRA 1972] wherein the problem is subdivided into several smaller problems without specifying details which are irrelevant to the current level of abstraction. If we now decide that the table is an array A , n is the index of the last entry and the counter variables reside in array B then we can expand the above program to

```
until Table_match or No_match do
  repeat for  $i \leftarrow 1$  to  $n$ 
    if  $A[i] = x$  then
      Match_index  $\leftarrow i$ 
      Table_match
  No_match
then case
  Table_match :
     $B[Match\_index] \leftarrow B[Match\_index] + 1$ 
  No_match :
     $n \leftarrow n + 1$ 
     $A[n] \leftarrow x$ 
     $B[n] \leftarrow 1$ 
```

Decision Trees

We shall now program an efficient version of the statement if (A and B) or (C and not D) then S_1 else S_2 using the until...then case statement. The program mirrors the flowchart in figure 2 and capitalizes on the ability of the until...then case statement to handle multiple outcomes.

```

until Yes or No do
  until Yes1 or No1 do
    Test_A&B :
      if A then
        if B then Yes1 else No1
      else No1
    then case
      Yes1 : Yes
      No1 : until Yes2 or No2 do
        Test_C&D:
          if C then
            if D then No2 else Yes2
          else No2
        then case
          Yes2 : Yes
          No2 : No
    then case
      Yes : S1
      No : S2

```

Because this program mirrors figure 2 it is easy to see that it correctly implements the original statement. In the same way we can program general decision trees with many terminal nodes some of which are identical with respect to the subsequent case-analysis.

Error Handling

Let us now consider a program to make a table with frequency counts from the values on an input file. Without much difficulty we arrive at the program

```

until No_more_data do
  Build_table_with_counts :
    Initialize_table
    repeat
      Get_new_x_from_input_file
      if End_of_input_file then
        No_more_data
      Enter_x_in_table
    Print_table_with_counts

```

The then case part of the statement has been omitted since a single-event case-analysis is redundant. This program contains the very familiar repeat loop with an imbedded termination which is somewhat awkward to express using while...do or repeat...until statements [KNUTH 1973]. The statement Enter_x_in_table represents the table search we programmed above as our first example. We soon find that this program may overrun the amount of storage allocated to arrays A,B[1:nmax]. One of the nicer aspects of the until...then case statement is that the appropriate modifications to handle the table overflow situation are easy to make and tend to be additive rather than disruptive of the existing program logic. In the case-analysis statement associated with No_match we find the statement n plus 1 which may cause an error in the subsequent statement. The solution is to insert directly before n plus 1 the test

```

  if n = nmax then Table_overflow

```

and next decide which enclosing block should be terminated as a result of the table overflow. If we decide that the overflow is just as terminal as No_more_data then we can simply insert the new event-identifier Table_overflow into the event list now containing the single event-identifier No_more_data and add a case-analysis part

```

  then case
    No_more_data : Print ('COMPLETE TABLE')
    Table_overflow : Print ('TABLE THRU RECORD',Input_count)

```

In general, the programmer can as an afterthought cause certain events to occur, choose what enclosing environment should thereby be immediately terminated and specify what actions should be taken subsequent and particular to the event.

JUSTIFICATION FOR UNTIL...THEN CASE

We should now like to enumerate what we think are the principal advantages of the until...then case statement.

Top-down Modular Structured Programming

The most important aspect of the until...then case statement is that it encourages a top-down modular programming style with structured program control. The until phrase lists all the mutually exclusive possible outcomes of the test-statement at the beginning while the then case part indicates a case-analysis based on the actual outcome. Details of how the case-determination is performed by the compound test-statement appear at a lower level of abstraction which is revealed by the syntactical nesting of the until...then case statement. Programmers are encouraged by this syntax to separate case-determination from case-analysis; the latter by its form enforces a clean separation of the various cases. Since the until...then case statement is equivalent to a flowchart (figure 3) with a single entry and a single exit, the requirement of structured control is fulfilled. Indeed, the two parts, case-determination and case-analysis each have this property if one envisages a "hidden" event variable whose value is set to the appropriate event at each event-statement while control passes immediately to the beginning of the case part.

Problem-oriented Syntax

The semantics of the until...then case statement, although equivalent to certain constructions in the BLISS language [WULF et al. 1971, WULF 1972] and exits previously proposed [LANDIN 1965, CLINT and HOARE 1972, BOCHMANN 1973], are nevertheless expressed by a syntax which seems more problem-oriented. One's understanding of the table search program is not enhanced by words like goto, exit, leave and jumpout or by concepts like "escape", "early" termination and "abnormal" exit. There are two possible outcomes for the table search and both are perfectly normal; when a matching entry is encountered in the table the immediate termination of the search is neither "early" nor "late" but, in fact, perfectly timed. The termination is not "abnormal" or in the nature of an "escape". The programmer is free to suggest a connotation of "abnormality" by appropriate choice of event-identifiers but nothing in the syntax makes such an arbitrary choice for him.

Correctness Proofs

Event-identifiers can be naturally associated with assertions [FLOYD 1967, HOARE 1969] of conditions which should be true at the time when a named event occurs via an event-statement. This not only helps in a posteriori proofs of correctness but also facilitates the construction of programs which are a priori correct. The formal semantics for event-statements have essentially been given by [CLINT and HOARE 1972] in their treatment of return-jumps and exit-jumps.

Program Modification

The modularity implied by separation of case-determination from case-analysis and the inherent subdivision of case-analysis allows modifications to be made in a clean and well-documented fashion. Furthermore, the changes to the existing program tend to be additions of new program pieces rather than reorganization of the old logic. The reader is reminded of our treatment of the table overflow possibility in the table search program. In general, case-determination can be redesigned independent of the case-analysis and the syntax shows textually where to focus attention.

Reduced and Simplified Control

A programming language whose control consists of normal sequence, if...then...else, a case statement, until...then case, a repeat statement with an optional for clause, and procedures will allow a convenient structured programming style. The for clause should be optionally open-ended (e.g. for $i \leftarrow 1$ by 1) to allow the programmer to terminate the repetition via an event whose defining condition is

rather complicated. Since three of these forms of control are case-analysis the entire control is reduced conceptually to sequence, case-analysis and repetition. The more familiar while...do, repeat...until, return, exit and stop statements are all subsumed in the new until... then case statement.

When the until...then case statement is used to control the termination of a repetition the textual placement of the event statements indicates exactly where in the time sequence of program execution the terminations can occur. No special conventions regarding the placement of implicit termination tests is required as it is with while...do and repeat...until. All repetition is indicated explicitly by the word repeat and never implied by convention as in the while...do statement.

A NON-TRIVIAL EXAMPLE

Our best success using the until...then case statement has been in programming an algorithm for partial sorting described by [DIJKSTRA 1971] based on the QUICKSORT algorithm [HOARE 1961]. The problem is to rearrange an array interval $A[m:n]$ so that all values less than a splitting value V are below all values greater than V and V is stored at the split location $A[f]$ between these two sets of values. The algorithm described by Dijkstra consists of alternating upward and downward scans working simultaneously from both ends of the array interval towards the middle and terminating when all array elements have been scanned and correctly placed. It is useful to quote directly from the description [DIJKSTRA 1971] of the algorithm:

"That is, we initialize $i = m$ and $j = n$ and 'take out' $A[n]$ - by assigning it to the variable V - thereby initializing the situation where scanning starts at element $A[i]$, while ' j ' points to the 'hole' just made. When the upward scan (under control of increasing ' i ') finds a large element, i.e. when for the first time $A[i] > V$, this value is placed in the hole, now leaving the hole in the place pointed to by ' i '. From then onwards a downward scan (under control of decreasing ' j ') can operate until a small element has been encountered which will be placed in the hole at position ' i ', leaving the hole in the place pointed to by ' j '. Such upward and downward scans have to succeed each other alternatively until $i = j$, i.e. until both point to the hole at the position around which the split has been effectuated. Finally the hole receives the value V which had been taken out at the beginning".

Dijkstra reports that his attempts to translate this algorithm into a repetition loop whose repeatable part consisted of an upward scan followed by a downward scan resulted in a "very ugly and messy" program whose termination was not easily proved because the condition for termination could arise during upward or downward scan. He then produced a very elegant program which scans exactly one array element during each execution of a repeatable statement. While appreciating the revised algorithm we were still bothered that the perfectly understandable algorithm quoted above was not programmable in a natural way employing the if...then...else and while...do statements alone.

The following program is an attempt to implement the quoted algorithm with little or no conceptual change:

```
until All_values_split do
  Initialize
  repeat
    until Large_element or Hole do
      Upward_scan
    then case
      Large_element : Put_value_into_hole_at_j
      Hole : All_values_split
    until Small_element or Hole do
      Downward_scan
    then case
      Small_element : Put_value_into_hole_at_i
      Hole : All_values_split
  Put_split_value_into_hole
```


The further specifications of Upward_scan and Put_value_into_hole_at_j are:

```
Upward_scan :  
  repeat  
    if i=j then Hole  
    if A[i] > V then Large_element  
    i plus 1
```

and

```
Put_value_into_hole_at_j :  
  A[j] ← A[i]  
  j minus 1
```

The downward scan is similar and therefore we omit its further expansion.

There is a temptation to consider the event Hole as unnecessary in view of the fact that it immediately triggers the event All_values_split. We have come to view this redundancy or verbosity as appropriate since the termination condition for each scan is most naturally expressed as the encountering of the "hole", whereas the ensuing event All_values_split occurs by virtue of the particular use being made of the sequence of scans. These two events are thus seen to exist at different levels of abstraction.

To return to the main motivation for performing this exercise, we feel that the above program is a reasonably accurate reflection of the algorithm informally described by Dijkstra and is fairly easy to construct in top-down fashion from the description. The use of event-identifiers which are words or phrases from the informal description makes this close correspondence more obvious.

EXTENSIONS AND OPEN QUESTIONS

Parametrized Events

It has been suggested [KNUTH 1974] based on the use of parametrized labels by [LANDIN 1965] to allow events to have parameters so that the various cases of the case-analysis part of until...then case may execute like procedures. The principal advantage of this extension is that we can now achieve more fully the modularity claimed for the until...then case statement. In the table search program there was a need for a global variable Match_index to communicate to the subsequent case-part the position of the matching entry in table A. It would be clearer to write in the test-statement:

```
if A[i] =x then Table_match(i)
```

and in the case-part:

```
Table_match(integer Match_index):  
  B[Match_index] plus 1
```

We do not favour the elimination of all global variables but it is helpful to minimize their use and make certain interfaces more explicit.

Preamble for the Case-analysis

It is conceivable that in using the until...then case statement a programmer would want to execute a piece of program after the termination of the test-statement but before the case-analysis part. In the present form of the statement this cannot be accomplished and we fear this restriction might reduce the usefulness of the until...then case statement for some applications. The simplest solution is to allow an optional compound statement to be inserted before the then case clause with the syntax then S. With a single parametrized event the programmer could achieve the same effect at the inconvenience of declaring and assigning values to an explicit event parameter. Since this negates many of the conceptual advantages of the proposed statement we find the latter solution unattractive. Another possibility is to replace the then case part by then S and allow within S a case-statement without explicit selection value to indicate a case-analysis based on the recently occurring

event. This would allow the programmer more freedom in his use of the event-outcome but the simplicity of the statement would suffer. To counterbalance these remarks we should mention that we have not yet encountered any real problems requiring the extra flexibility.

Events in Procedures

The usefulness of the event-statement in an enclosing until...then case statement to handle errors suggests that event-statements should be allowed in procedures even when the controlling until phrase is in a separately compiled procedure. The global attribute could be attached to an event-identifier in an until phrase to signify that the programmer wishes to allow the named event to occur via event-statement in some separately compiled procedure. A linking-loader should be able to establish the name correspondence but a run-time check will still be necessary to ensure that the corresponding test-statement is being executed when a global event occurs.

CONCLUSION

We have presented a control statement which combines the power of the case-statement with that of the if...then...else statement in a syntax which encourages a top-down programming style. Several examples have been given to exhibit the "naturalness" of this new control statement in coping with situations not easily handled by if...then...else, while...do and repeat...until. We are hopeful that such a statement will be implemented as one of the control statements of some future languages but in anticipation of and to provide guidance for that development more and wider experience is needed than that of a single programmer. We have reason to suspect that the reduced control structure discussed above will be adequate for a convenient structured style of program composition, particularly if procedure is interpreted to include routines, functions and coroutines with recursion.

It is possible to view our proposal as an invitation to think in terms of figure 3 when constructing programs. Lacking an actually implemented until...then case statement one can still write "abstract" programs employing this statement and then simulate the statement in some implemented language. PASCAL and PL/1 each have language features which facilitate such simulations. The semantic equivalence to the leave statement in BLISS has already been mentioned. In PASCAL the ability to construct one's own "scalar" types (for event lists) and the existence of a very general case-statement makes the transformation less distorting conceptually than it would be in other languages. There remain some nasty goto <number> statements and their concomitant numeric labels. In PL/1 the until...then case statement can be implemented through the use of programmer defined "conditions" which can be "signalled" to produce pseudo-interrupts. These interrupts can be handled by a programmer-specified block which can contain as its final statement a global goto. The connection between PL/1 conditions and the semantics of until...then case was pointed out by [CLINT and HOARE 1972].

We would like to encourage feedback from programmers who have tried to write "abstract" programs with the until...then case statement. It was the difficulties encountered with while...do and repeat...until that led to our proposal and we anticipate that experience rather than guesswork will give a better guide to desirable modifications.

ACKNOWLEDGEMENTS

I thank Dr. Paolo Zanella of the CERN Data Handling Division for his friendly encouragement of my interest in programming methods while I was visiting CERN. I am indebted to both CERN and SLAC for making possible the leave of absence during which this work was done. My first serious thoughts about programming style date from my first reading of Edsger Dijkstra's notes on structured programming several years ago. I also thank Donald Knuth, Bob Russell and Brian Carpenter for reading a first draft of this paper and providing helpful feedback. The encouragement and enthusiasm of Donald Knuth has been a stimulus to make a careful revision of the first draft which I hope has been beneficial.

REFERENCES

- BOCHMANN G.V., "Multiple exits from a loop without the goto", Comm. ACM 16 (1973), pp. 443-444.
- CLINT M. and HOARE C.A.R., "Program proving : jumps and functions", Acta Informatica 1 (1972), pp. 214-224.
- DIJKSTRA E.W., "Goto statement considered harmful", Comm. ACM 11 (1968), pp. 147-148.
- DIJKSTRA E.W., "A short introduction to the art of programming", Technological University Eindhoven report EWD 316 (August 1971).
- DIJKSTRA E.W., "Notes on structured programming" in Structured Programming by Dahl, Dijkstra and Hoare, Academic Press (1972).
- FLOYD R.W., "Assigning meanings to programs", AMS Symposium in Applied Mathematics, Vol. XIX, 1967.
- HOARE C.A.R., Algorithms 63-65, Comm. ACM 4 (1961), pp. 321-322.
- HOARE C.A.R., "An axiomatic basis for computer programming", Comm. ACM 12 (1969), pp. 576-581.
- HOPKINS M.E., "A case for the goto", National ACM Conf. Proc. (1972), pp. 787-790.
- KNUTH D.E., "A review of structured programming", Stanford University Computer Science Department report CS-73-371 (June 1973).
- KNUTH D.E., Personal communication (January 1974).
- KNUTH D.E. and FLOYD R.W., "Notes on avoiding goto statements", Information Processing Letters 1 (1971), pp. 23-31.
- LANDIN P.J., "A correspondence between ALGOL 60 and Church's lambda-notation : part I", Comm. ACM 8 (1965), pp. 89-101.
- LEAVENWORTH B.M., "Programming with(out) the goto", National ACM Conf. Proc. (1972), pp. 782-786.
- LISKOV B., "Report of session on structured programming", Notices of ACM Special Interest Group on Programming Languages, (SIGPLAN), Vol. 8, No. 9 (September 1973), pp. 5-10.
- WEGNER E., "Tree-structured programs", Comm. ACM 16 (1973), pp. 704-705.
- WIRTH N., "On certain basic concepts of programming languages", Stanford University Computer Science Department report CS-65 (May 1967).

WULF W.A., "Programming without the goto", Conf. Proc. IFIP-71 (1971), pp. 408-413.

WULF W.A., "A case against the goto", National ACM Conf. Proc. (1972), pp. 791-797.

WULF W.A., RUSSELL D.B. and HABERMANN A.N., "BLISS : A Language for systems programming", Comm. ACM 14 (1971), pp. 780-790.