INFORMATION PROCESSING 71 – NORTH-HOLLAND PUBLISHING COMPANY (1972)

# PROGRAMMING WITHOUT THE GOTO

William A.WULF*
*Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, USA*

It has been proposed, by Dijkstra and others, that the use of the *goto* statement is a major villain in programs which are difficult to understand and debug. The proponents of eliminating the *goto* contend that when it is eliminated the resulting program structure admits a simple, systematic proof of correctness. This suggestion has met with skepticism in some circles. This paper analyzes the nature of control structures which cannot be easily synthesized from simple conditional and loop constructs. This analysis is then used as the basis for developing the control structures of a particular language, Bliss. The results of two years of experience programming in Bliss, and hence without *goto's*, are summarized.

## 1. INTRODUCTION

In 1968 E.W.Dijkstra suggested, in a letter to the editor of the Communications of the ACM [1], that use of the *goto* construct in Algol was undesirable, and in fact was bad programming practice. The rationale behind this suggestion was that it is possible to use the *goto* in ways which obscure the logical structure of a program, thus making it difficult to understand, debug, and prove its correctness. Not all uses of the *goto* are obscure, but the conjecture is that these situations are adequately handled by conditional and loop constructs.

This paper presents an analysis which led to the design of the control features of Bliss [2], an implementation language designed at Carnegie-Mellon University. This analysis reveals that the Algol conditional and looping constructs are, while adequate, not sufficiently convenient of themselves. The control features of Bliss are described and some comments are made concerning our experiences using a *goto*-less, language.

It is worth noting an additional benefit of removing the *goto* – a benefit which the author did not fully appreciate until the Bliss compiler was designed – that of code optimization. The presence of *goto* in a block-structured language with dynamic storage allocation forces runtime overhead for jumping out of blocks and procedures. Eliminating the *goto* removes this overhead. More important, however, is the fact that the scope of a control environment is statistically defined. The Fortran-H compiler [3], for example, does considerable analysis and achieves a less perfect picture of overall control flow than that implicit in the text of a Bliss program. Since flow analysis is prerequisite to global optimization, this benefit of eliminating the *goto* must not be underestimated.

It is not surprising that a language can be devised which does not use the *goto* since: (1) several formal systems of computability theory, e.g., recursive functions, do not contain the concept; (2) Lisp does not use it; and (3) Van Wijngaarden [4], in defining the semantics of Algol, eliminated labels and *goto's* by systematic substitution of procedures. Thus the question is not whether it is *possible* to remove the *goto*, only whether it is practical. In particular there is some suspicion among programmers that the advantages are outweighed by inconvenience and inefficiency. The objective of this paper is to investigate these issues.

## 2. ANALYSIS

We shall first consider programs which cannot be easily built from simple conditional and looping constructs. To do this we will use a flow chart representation because of the explicit way in which it manifests control. We assume two basic blocks (fig. 1). These boxes are connected by directed line segments in the usual way. We are interested in two special "*goto*-less" constructions – simple loop and *n*-way conditional (or

* This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research.

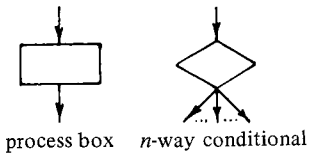


Fig. 1.

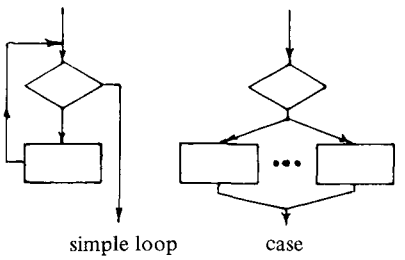


Fig. 2.

"case") constructs (fig. 2). We consider these forms "*goto*-less" since they contain single entry and exit points. (The loop considered here does not correspond to all variants of initialization, test before or after the loop body, etc. These variants would not change the arguments to follow and have been omitted.)

Consider three transformations:

(1) any linear sequence of process boxes may be replaced by a single process box (fig. 3);

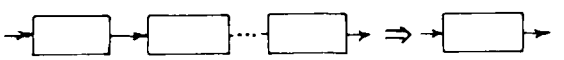

Fig. 3.

(2) any simple loop may be replaced by a process box (fig. 4);

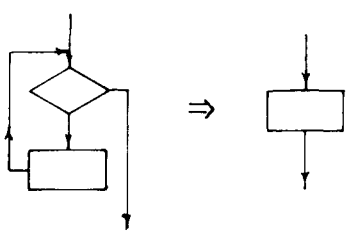

Fig. 4.

(3) any $n$-way case construct may be replaced by a process box (fig. 5).

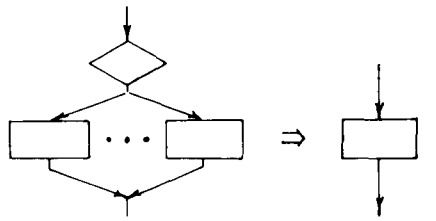

Fig. 5.

Any graph which may be derived by a sequence of these transformations we shall call a "reduced" form of the original. We shall say that a reduced graph consisting of a single node is "*goto*-less" and that the sequence of transformations defines a set of nested "control environments".

Not all graphs are of this type; they are of interest to us since they represent programs which cannot be realized by simple conditionals and loops. Examination of these graphs will reveal techniques for deriving simple *goto*-less graphs and provide insight leading to the control primitives to be described later.

By definition, a *goto*-less flow chart is susceptible to a sequence of transformations which reduces it to a single process box. Imagine such a sequence in which: (1) the correctness of the replaced construct has been verified, and (2) the new process box contains a more macroscopic description of what the replaced portion does. This sequence forms both a proof of the original program as well as documentation of what it does. This is not to say that other programs cannot be understood or proved correct [5], only that programs with this structure permit a specific methodical approach.

Returning to an analysis of programs, consider two cases; those with and without loops. Programs without loops have, at most, a latticelike structure. For example, consider the graph below (in this example, and the remainder of the paper, we shall use circles to represent subgraphs whose fine-structure we choose to ignore).

Consideration of such graphs reveals that it is always possible to construct a new graph which is similar to the original graph except for a finite number of "node splittings" (i.e., creation of duplicate nodes). This follows from the observation that there are at most a finite number of paths through the graph and each node occurs on only finitely many of them. Hence at most a finite number of replications of each node will guarantee that it occurs in only one path. For example, the graph shown in fig. 6 transforms into that shown in fig. 7. Node splitting is something we

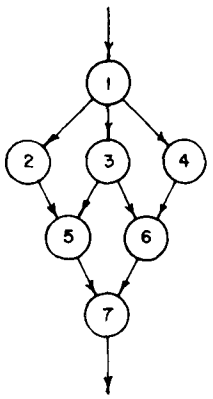


Fig. 6.

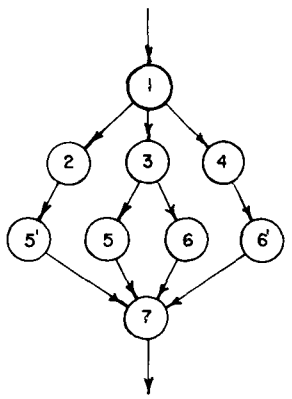


Fig. 7.

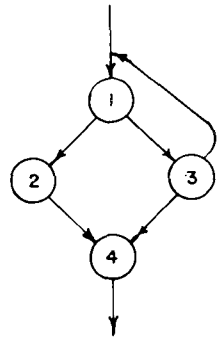


Fig. 8.

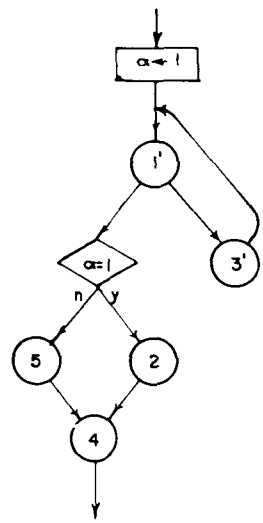


Fig. 9.

would like to avoid since it involves duplicating code. A second technique, which also might have been used above, will be discussed below.

The second case to be considered is that of graphs involving a loop (with more than one entry or exit point).

Floyd and Knuth [6] have proven (using flow charts as specifications for regular expressions) that node splitting is not adequate for deriving *goto*-less graphs in the presence of multiple entry/exit loops. This also follows from observing that the number of paths leading from the "second" exit point is unbounded. Therefore no finite number of replications of this node is sufficient. Consider the following program (fig. 8): There are two exit paths from the ① – ③ loop – that leading from ① to ② to ④ and that leading from ③ to ④. This is an example of a program where node splitting will not work. However, one can introduce a new variable, call it $\alpha$, and obtain the graph in fig. 9.

In this graph ①' is like node ① except that the exit condition of the loop has been augmented with "or $\alpha = 0$" as has any code preceding the test, and node ③' is like node ③ except that the exit to node ④ has been replaced by the operation "$\alpha \leftarrow 0$". Node ⑤ is the null operation. Conceptually we have introduced a "state" variable which, when the loop terminates, specifies whether or not to execute ②.

That the technique is general may be seen easily, consider a graph with nodes ①, ②, ..., ⓝ. Now construct a new graph as follows:

(1) if ⓘ is a process box construct ⓘ' by adding to (the end of) ⓘ "$\alpha \leftarrow k$" where ⓚ is the successor of $i$;

(2) if ⓘ is a decision box, then replace it by a process box of the form "$\alpha \leftarrow \epsilon$", where $\epsilon$ is an expression which dynamically evaluates to the appropriate successor label;

(3) consider all exit points as labeled by ⓪;

(4) construct the graph in fig. 10.

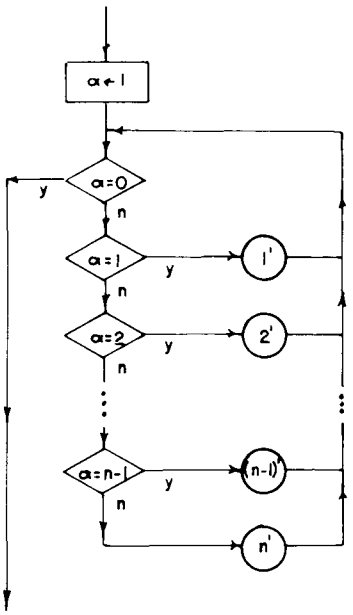


Fig. 10.

As with node splitting, this technique is odious because it is inefficient.

## 3. THE BLISS CONTROL STRUCTURE

The previous section describes programs which may not easily be constructed with only conditionals and loops. The present section addresses itself to the question of whether the class of constructs in a practical language (without an explicit *goto*) should be extended. If the decision is to extend the class, then what should the extensions be? The answer to these questions depends in part on the frequency with which multiple exits from loops, etc., are used, and in part on the answer to the second question. Hence we answer in the context of a specific language, namely Bliss [2].

Bliss is a block-structured "expression language". That is, every executable construct, including those which manifest control, is an expression and computes a value. This is relevant to the *goto* issue in the following way: the method described in the first section for translating programs into *goto*-less form involved an explicit state variable. The value of an expression (e.g., a block) forms a natural *implicit* state value.

Expressions may be concatenated with semicolons to form expression sequences. An expression sequence is evaluated in left-to-right order and its value is that of its last (rightmost) component expression. A pair of symbols, *begin* and *end*, or "(" and ")", may be used to embrace an expression sequence and form a simple expression. A block is a special case of this which contains declarations.

There are six explicit control forms in Bliss: conditional, loop, case-select, function, co-routine, and escape. We have omitted subroutines, and so shall omit functions and co-routines here.

The conditional expression

*if* $\epsilon_1$ *then* $\epsilon_2$ *else* $\epsilon_3$

is defined to have the value $\epsilon_2$ just in the case that $\epsilon_1$ evaluates to *true* and $\epsilon_3$ otherwise.

The *case* and *select* expression provide $n$-way branching.

*case* $e_0, e_1, ..., e_k$ *of nset* $\epsilon_0; \epsilon_1; ...; \epsilon_n$ *tesn*.

*select* $e_0, e_1, ..., e_k$ *of*

*nset* $\epsilon_0 : \epsilon_1; \epsilon_2 : \epsilon_3; ...; \epsilon_{2n} : \epsilon_{2n+1}$ *tesn*.

The *case* expression is executed as follows: (1) $e_0, ..., e_k$ are evaluated, (2) the value of $e_i$ $(0 \leqslant i \leqslant k)$ is, in turn from left to right, used as an index to choose one of the $\epsilon_j$'s. The $e_i$'s are constrained to $0 \leqslant e_i \leqslant n$. Execution is undefined for other values of $e$. The value of the *case* expression is $\epsilon_{e_k}$. The "case" has appeared in several other languages, e.g., Algol-W [7] and Euler [8].

The *select* expression is similar to the *case* except that the $e_i$'s are used in conjunction with the $\epsilon_{2j}$'s to choose among the $\epsilon_{2j+1}$'s. Execution proceeds as follows: (1) the $e_i$'s are evaluated, (2) $\epsilon_0$ is evaluated, (3) if the value of $\epsilon_0$ is identical to the value of one (or more) of the $e$'s then $\epsilon_1$ is executed, (4) $\epsilon_2$ is evaluated, (5) if the value of $\epsilon_2$ is identical to the value of one (or more) of the $e$'s then $\epsilon_3$ is executed, etc. The value of the select expression is that of the last $\epsilon_{2j+1}$ to be executed, or $-1$ if none of them is executed.

The use of the expression nature of Bliss may be illustrated using an earlier example, namely that in fig. 11 (fig. 6). This graph may be thought of as actually of the form shown in fig. 12, where (A) is formed from (1), (2), (3) and (4) as shown in fig. 13, which one might write in (pseudo) Bliss:

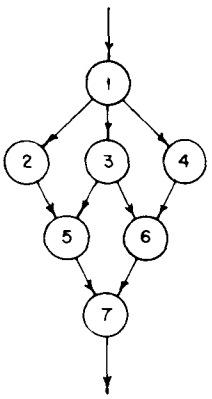


Fig. 11.

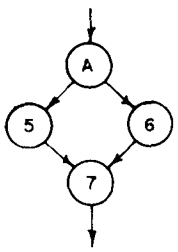


Fig. 12.

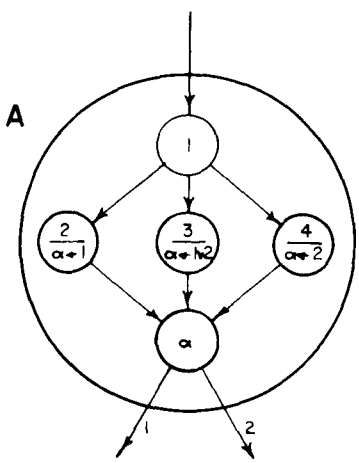


Fig. 13.

*case case* ① *of set* (②;0); (③;0 ∨ 1); (④;1)

*tes of set* ⑤; ⑥ *tes*; ⑦ .

The loop expressions imply repeated execution (possibly zero times) until a specified condition is satisfied. Two of the available forms are:

*while* $\epsilon_1$ *do* $\epsilon$

*incr* ⟨id⟩ *from* $\epsilon_1$ *to* $\epsilon_2$ *by* $\epsilon_3$ *do* $\epsilon$ .

In the first form the expression $\epsilon$ is repeated so long as $\epsilon_1$ is *true*. The second form is similar to the "*step ...until*" construct of Algol, except (1) the control variable is local to $\epsilon$, and (2) $\epsilon_1$, $\epsilon_2$ and $\epsilon_3$ are evaluated only once (before the first evaluation of the loop body). Except for an escape expression within $\epsilon$ (see below) the value of a loop is $-1$.

The control mechanisms described above are either similar to, or slight generalizations of, constructs in many other languages. They do not solve the problems discussed earlier. Another mechanism is needed – the *escape*. An escape is highly structured forward branch to the terminus of a control environment in which the escape is nested. The general form is:

⟨escapetype⟩ ⟨levels⟩ ⟨expression⟩

where ⟨escapetype⟩ is one of the (reserved) words listed below and ⟨levels⟩ is either an integer enclosed in square brackets or is empty (which implies "[1]").

| | |
|---|---|
| *exitblock* | *exitcase* |
| *exitcompound* | *exitselect* |
| *exitloop* | *exit* |
| *exitconditional* | *return* |

An escape forces control to exit from a specified control environment (a block, etc.). The ⟨levels⟩ construct permits exit from several nested loops (for example). The ⟨expression⟩ defines the value of the environment.

The use of an escape is illustrated by a typical problem involving multiple exits from a loop. Suppose a vector, $X$, is to be searched for a value, $x$. If an element of $X$ is equal to $x$, then the variable, $k$, is to be set to the index of this element. If no element of $X$ is equal to $x$, then the value of $x$ is to be inserted after the last element of $X$ and $k$ set to this index. If there are $N$ elements in $X$, the following Bliss expression* will do this:

*if* ($k \leftarrow$ *incr i from* 1 *to N by* 1 *do if* $X[i] = x$ *then*

*exitloop i*) $< 0$ *then* $X[k \leftarrow N \leftarrow N+1] \leftarrow x$ ;

Returning to the original questions of this section:

* The given expression is not Bliss; the differences are not essential to control.

these mechanisms are "adequate", but are they convenient and do they preserve the desirable *goto*-less properties? The answer to the first question lies principally in the experience of those who have used the language. These experiences are summarized below and essentially answered in the affirmative.

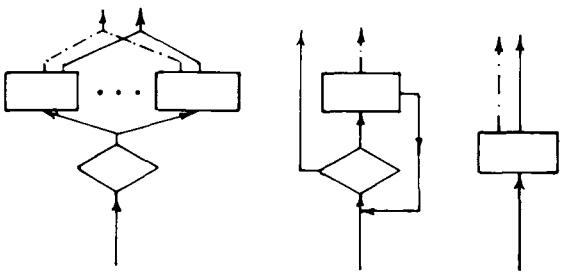

Fig. 14.

The second question, whether the Bliss structures retain the desirable properties of *goto*-less notations, requires more consideration. It is only the escape mechanism which violates the *goto*-less criteria. We now think of our flow chart primitives as those in fig. 14, where the dotted lines represent the set of flow lines which may be followed if an escape is invoked. The previous transformations are applicable if the dotted, "escape", lines are ignored. We are guaranteed that the escape lines will be totally enclosed at some stage in the reduction process. Hence, one can apply the former reasoning to sub-graphs from which no dotted lines emanate. After this attention must shift to a sub-graph which wholly contains its escape lines. This may or may not lead to the simpler form of graph, but in either case the process can be iterated. In this sense the desirable properties of *goto*-less graphs are retained.

Bliss has been in active use for two years and we have gained considerable experience programming without the *goto*. This experience includes several compilers, a conversational programming system (APL), parts of an operating system, and numerous applications programs. Writing programs presents no difficulty. Just as one adapts to the inability to jump into the middle of an Algol block, one also adapts to Bliss. But it is not that one merely survives in this mode; quite the contrary. I am convinced that programmer productivity has significantly improved due to this enforced style of programming.

We have found two inconvenient aspects of the Bliss structure. The "⟨levels⟩" construct embodies an important semantic notion. But, as a program is modified, the number of levels through which an escape should execute may change. Hence one should indicate its target symbolically – i.e., labels should be reintroduced. The other construct is one which allows an exit through several levels of subroutine call. Both of these can be incorporated into a simple syntax:

⟨expression⟩ ::= ··· |⟨label⟩ : ⟨expression⟩ .

This defines the ⟨label⟩ as the name of the control environment of the ⟨expression⟩. A subroutine name would be considered a ⟨label⟩ in the same sense. Now, replace all of the escapes by

*leave* ⟨label⟩ ⟨expression⟩ .

This causes control to exit from the control environment named ⟨label⟩ (including exit through several levels of subroutines) and defines ⟨expression⟩ as its value.

Whether or not a language includes the *goto* construct is immaterial. There are certain types of control flow which occur in real programs. If these constructs are not explicitly provided, the *goto* must be provided so that the programmer may synthesize them. The danger in the *goto* is that the programmer will do this in obscure ways. The advantage in eliminating the *goto* is that these same control structures will appear in regular and well-defined ways. Both the human and the compiler will do a better job of interpreting them.

## REFERENCES

[1] E.W.Dijkstra, Goto statement considered harmful, Letter to the Editor, CACM 11, 3 (March 1968).
[2] W.A.Wulf et al., Bliss reference manual, Computer Science Department Report, Carnegie-Mellon University.
[3] Lowery and Medlock, Object code optimization, CACM 12, 1 (January 1969).
[4] A.Van Wijngaarden, Recursive definition of syntax and semantics, in: Formal Language Description Languages, T.B.Steel, ed., (North-Holland, Amsterdam, 1966).
[5] J.King, A program verifier, Ph. D. Dissertation, Carnegie-Mellon University, 1969.
[6] F.Knuth, Notes on avoiding 'GOTO' statements, Technical Report CS 148, Stanford University, January 1970.
[7] Wirth and Hoare, A contribution to the development of Algol, CACM 9, 6 (June 1966).
[8] Wirth and Weber, Euler: A generalization of Algol and its formal definition, CACM 9, 1 and 2 (January and February 1966).