# A Case Against the GOTO

William A. Wulf, Carnegie-Mellon University

ABSTRACT

It has been proposed, by E. W. Dijkstra and others, that the goto statement in programming language is a principal culprit in programs which are difficult to understand, modify, and debug. More correctly, the argument is that it is possible to use the goto to synthesize program structures with these undesirable properties. Not all uses of the goto are to be considered harmful; however, it is further argued that the "good" uses of the goto fall into one of a small number of specific cases which may be handled by specific language constructs. This paper summarizes the arguments in favor of eliminating the goto statement and some of the theoretical and practical implications of the proposal.

KEY WORDS AND PHRASES: programming, programming languages, goto-less programming, structured programming
CR CATEGORIES: 4.2, 4.22, 5.24

## INTRODUCTION

It has been suggested that the use of the goto construct is undesirable, is bad programming practice, and that at least one measure of the 'quality' of a program is inversely related to the number of goto statements contained in it. The rationale behind this suggestion is that it is possible to use the goto in ways which obscure the logical structure of a program, thus making it difficult to understand, modify, debug, and/or prove its correctness. It is quite clear that not all uses of the goto are obscure, but the hypothesis is that these situations fall into one of a small number of cases and therefore explicit and inherently well-structured language constructs may be introduced to handle them. Although the suggestion to ban the goto appears to have been a part of the computing folklore for several years, to this author's knowledge the suggestion was first made in print by Professor E. W. Dijkstra in a letter to the editor of the Communications of the ACM in 1968 (1).

In this paper we shall examine the rationale for the elimination of the goto in programming languages, and some of the theoretical and practical implications of its (total) elimination.

## RATIONALE

At one level, the rationale for eliminating the goto has already been given in the introduction. Namely, it is possible to use the goto in a manner which obscures the logical structure of a program to a point where it becomes virtually impossible to understand (1,3,4). It is not claimed that every use of the goto obscures the logical structure of a program; it is only claimed that it is possible to use the goto to fabricate a "rat's nest" of control flow which has the undesirable properties mentioned above. Hence this argument addresses the use of the goto rather than the goto itself.

As the basis for a proposal to totally eliminate the goto this argument is somewhat weak. It might reasonably be argued that the undesirable consequences of unrestricted branching may be eliminated by enforcing restrictions on the use of the goto rather than eliminating the construct. However, it will be seen that any rational set of restrictions is equivalent to eliminating the construct if an adequate set of other control primitives is provided. The strong reasons for eliminating the goto arise in the context of more positive proposals for a programming methodology which makes the goto unnecessary. It is not the purpose of this paper to explicate these methodologies (variously called "structured programming", "constructive programming", "stepwise refinement", etc.); however, since the major justification for eliminating the goto lies in this work, a few words are in order.

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.

It is, perhaps, pedantic to observe that the present practice of building large programming systems is a mess. Most, if not all, of the major operating systems, compilers, information systems, etc. developed in the last decade have been delivered late, have performed below expectation (at least initially), and have been filled with 'bugs'. This situation is intolerable, and has prompted several researchers ((2,3,4), (5,6), (7), (8), (9)) to consider whether a programming methodology might be developed to correct this situation. This work has proceeded from two premises:

1. Dijkstra speaks of our "human inability to do much" (at one time) to point up the necessity of decomposing large systems into smaller, more "human size" chunks. This observation is hardly startling, and in fact, most programming languages include features (modules, subroutines, and macros, for example) to aid in the mechanical aspects of this decomposition. However, the further observation that the particular decomposition chosen makes a significant difference to the understandability, modifiability, etc., of a program and that there is an a priori methodology for choosing a "good" decomposition is less expected.

2. Dijkstra has also said that debugging can show the presence of errors, but never their absence. Thus ultimately we will have to be able to prove the correctness of the programs we construct (rather than "debug" them) since their sheer size prohibits exhaustive testing. Although some progress has been made on the automatic proof of the correctness of programs (c.f., (10), (11), (12), (23), (24)), this approach appears to be far from a practical reality. The methodology proposed by Dijkstra (and others) proceeds so that the construction of a program guides a (comparatively) simple and intuitive proof of its correctness.

The methodology of "constructive programming" is quite simple and, in this context, best described by an (partial) example. Let us consider the problem of producing a KWIC* index. Construction of the program proceeds in a series of steps in which each step is a refinement of some portion of a previous step. We start with a single statement of the function to be performed:

*For those who may not be familiar with a KWIC (key word in context) index, the following description is adequate for this paper.

A KWIC system accepts a set of lines. Each line is an ordered set of words and each word is an ordered set of characters. A word may be one of a set of uninteresting words ("a", "the", "of", etc.), otherwise it is a key word. Any line may be circularly shifted by removing its first word and placing it at the end of the line. The KWIC index system generates an ordered (alphabetically by the first word) listing of all circular shifts of the input lines such that no line in the output begins with an uninteresting word.

```
Step 1:  PRINTKWIC
```

We may think of this as being an instruction in a language (or machine) in which the notion of generating a KWIC index is primitive. Since this operation is not primitive in most practical languages, we proceed to define it:

```
Step 2:  PRINTKWIC:  generate and save all
                     interesting circular
                     shifts

                     alphabetize the saved
                     lines

                     print alphabetized lines
```

Again, we may think of each of these lines as being an instruction in an appropriate language; and again, since they are not primitive in most existing languages, we must define them; for example:

```
Step 3a:  generate and save all interesting
          circular shifts:

                for each line in the input do
                   begin
                   generate and save all inter-
                      esting shifts of 'this
                      line'
                   end
etc.
```

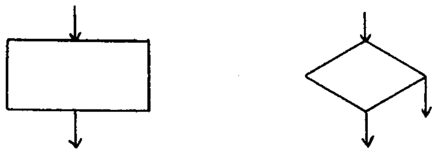
The construction of the program proceeds by small steps* in this way until ultimately each operation is expressed in the available primitive operations of the target language. We shall not carry out the details since the objective of this paper is not to be a tutorial on this methodology. However, note that the methodology achieves the goals set out for it. Since the context is small at each step it is relatively easy to understand what is going on; indeed, it is easy to prove that the program will work correctly if the primitives from which it is constructed are correct. Moreover, proving the correctness of the primitives used at step $i$ is a small set of proofs (of the same kind) at step $i$+1. (In the terminology of this methodology, step $i$ is an abstraction from its implementation in step $i$+1.)

Now, the constructive programming methodology relates to eliminating the goto in the following way. It is crucial to the constructive philosophy that it should be possible to define the behavior of each primitive action at the $i$th step independent of the context in which it occurs. If this were not so, it would not be possible to prove the correctness of these primitives at the $i$+1st step without reference to their context in the $i$th step. In particular, this suggests (using flow chart terminology) that it should be possible to represent each primitive at the $i$th step by a (sub) flow chart with a single entry and a single exit path. Since this must be true at each step of the

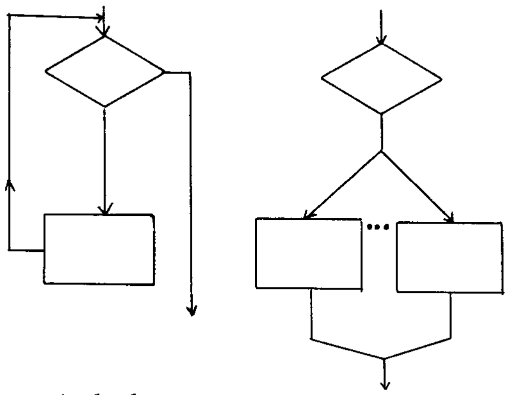
*A more complete explication of the methodology would concern itself with the nature and order of the decisions made at each step as well as the fact that they are small. See (22) for an analysis of two alternative decompositions of a KWIC system similar to the one defined here.

construction, the final flow chart of a program constructed in this way must consist of a set of totally nested (sub) flow charts. Such a flow chart can be constructed without an explicit goto if conditional and looping constructs are available.

Consider, now, programs which can be built from only simple conditional and loop 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 flow chart elements, a "process" box and a "binary decision" box:

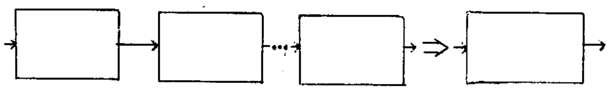


These boxes are connected by directed line segments in the usual way. We are interested in two special "goto-less" constructions fabricated from these primitives: a simple loop and an n-way conditional, or "case", construct. We consider these forms "goto-less" since they contain single entry and exit points and hence might reasonable be provided in a language by explicit syntactic constructs. (The loop considered here obviously 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.)

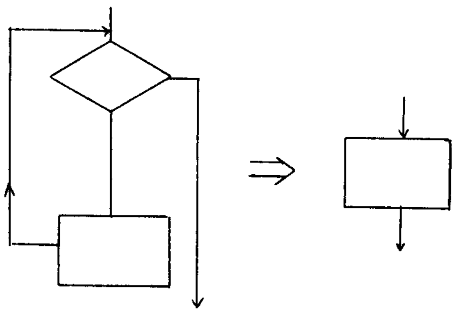

simple loop case

Consider the following three transformations (T1, T2, T3) defined on arbitrary flow charts:

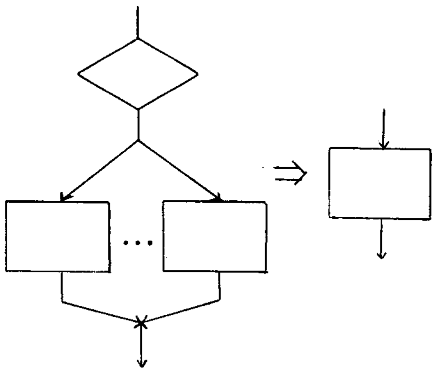
T1. any linear sequence of process boxes may be mapped into a single process box



T2. any simple loop may be mapped into a process box



T3. any n-way "case" construct may be mapped into a process box



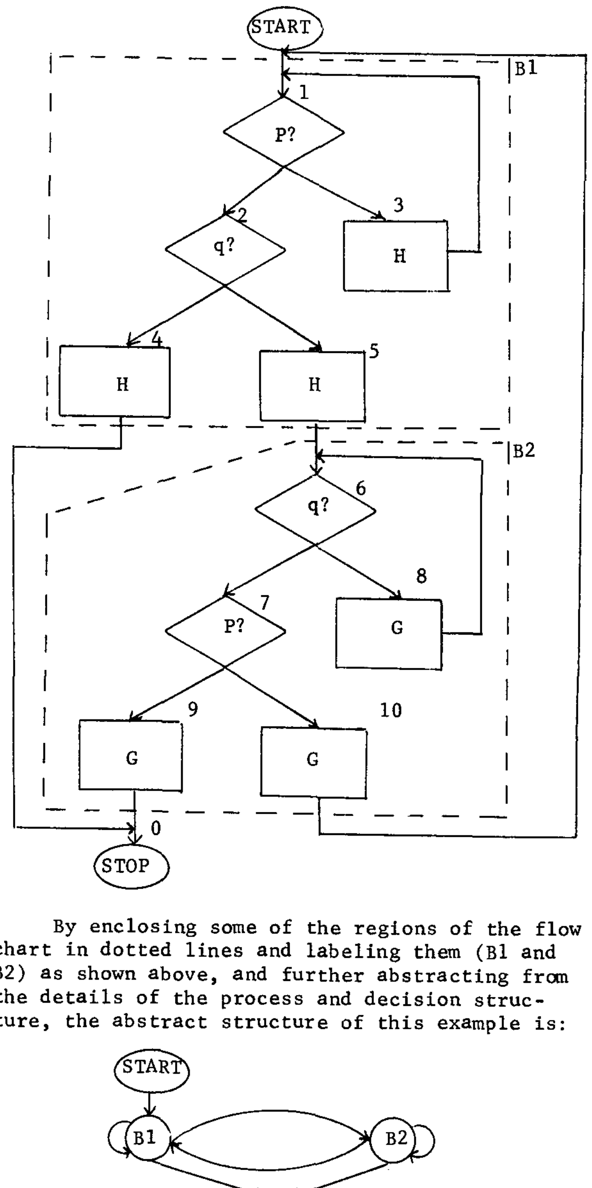
Any graph (flow chart) which may be derived by a sequence of these transformations we shall call a "reduced" form of the original. We shall say that a graph which may be reduced to a single node by some sequence of transformations is "goto-less" (independent of whether actual goto statements are used in its encoding) and that the sequence of transformations defines a set of nested "control environments". The sequence of transformations applied in order to reduce a graph to a single node may be used as a guide to both understanding and proving the correctness of the program (2,4,6,7,19).

The property of being "goto-less" in the sense defined above is a necessary condition for the program to have been designed by the constructive methodology. Moreover, the property depends only upon the topology of the program and not on the primitives from which it is synthesized; in particular, a goto statement might have been used. However, not only can such programs be constructed without a goto if conditionals and loops are available, but any use of a goto which is not equivalent to one of these will destroy the requisite topology. Hence any set of restrictions (on the use of the goto) which is intended to achieve this topology is equivalent to eliminating the goto.

## THE THEORETICAL POSSIBILITY OF ELIMINATING THE GOTO

It is possible to express the evaluation of an arbitrary computable function in a notation which does not have an explicit goto. This is not particularly surprising since: (1) several formal systems of computability theory, e.g., recursive functions, do not use the concept; (2) (pure) LISP does not use it; and (3) Van Wijgaarden (13), in defining the semantics of Algol, eliminated labels and goto's by systematic substitution of procedures. However, this does not say that an algorithm for the evaluation of these functions is especially convenient or transparent in goto-less form. Alan Perlis has referred to similar situations as the 'Turing Tarpit' in which everything is possible, but nothing is easy.

Knuth and Floyd (14) and Ashcroft and Manna (15) have shown that given an arbitrary flow chart it is not possible to construct another flow chart (using the same primitives and no additional variables) which performs the same algorithm and uses only simple conditional and loop constructs; of course other algorithms exist that compute the same function and which can be expressed with only simple conditionals and loops. The example given in Ashcroft and Manna of an algorithm which cannot be written in goto-less form without adding additional variables is:



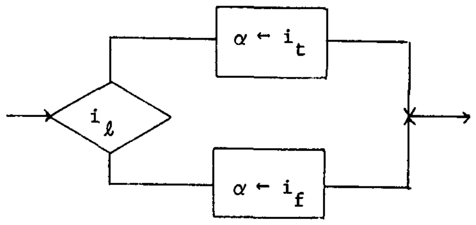
By enclosing some of the regions of the flow chart in dotted lines and labeling them (B1 and B2) as shown above, and further abstracting from the details of the process and decision structure, the abstract structure of this example is:

$$\begin{tikzcd} & \text{START} \arrow[d] & & \\ & \text{B1} \arrow[loop left] \arrow[rr, bend left] \arrow[dr] & & \text{B2} \arrow[loop right] \arrow[ll, bend left] \arrow[dl] \\ & & \text{STOP} & \end{tikzcd}$$

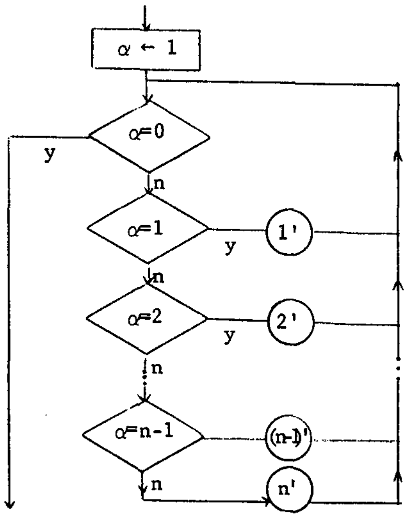
The reader is referred to (15) for a proof that such programs cannot be constructed from simple looping and conditional constructs unless an additional variable is added. Intuitively, however, it should be clear from the abstraction of the example that neither B1 nor B2 is inherently nested within the other. Moreover, the existence of multiple exit paths from B1 and B2 make it impossible to impose a superior (simple) loop (which inherently has a single exit path) to control the iteration between them unless some mechanism for path selection (e.g., an additional variable) is introduced.

In (21) Böhm and Jacopini show that an arbitrary flow chart program may be translated into an equivalent one with a single "while statement" by introducing new boolean variables, predicates to test them, and assignment statements to set them. A variant of this scheme involving the addition of a single integer variable, call it '$\alpha$', which serves as a 'program counter' is given below.

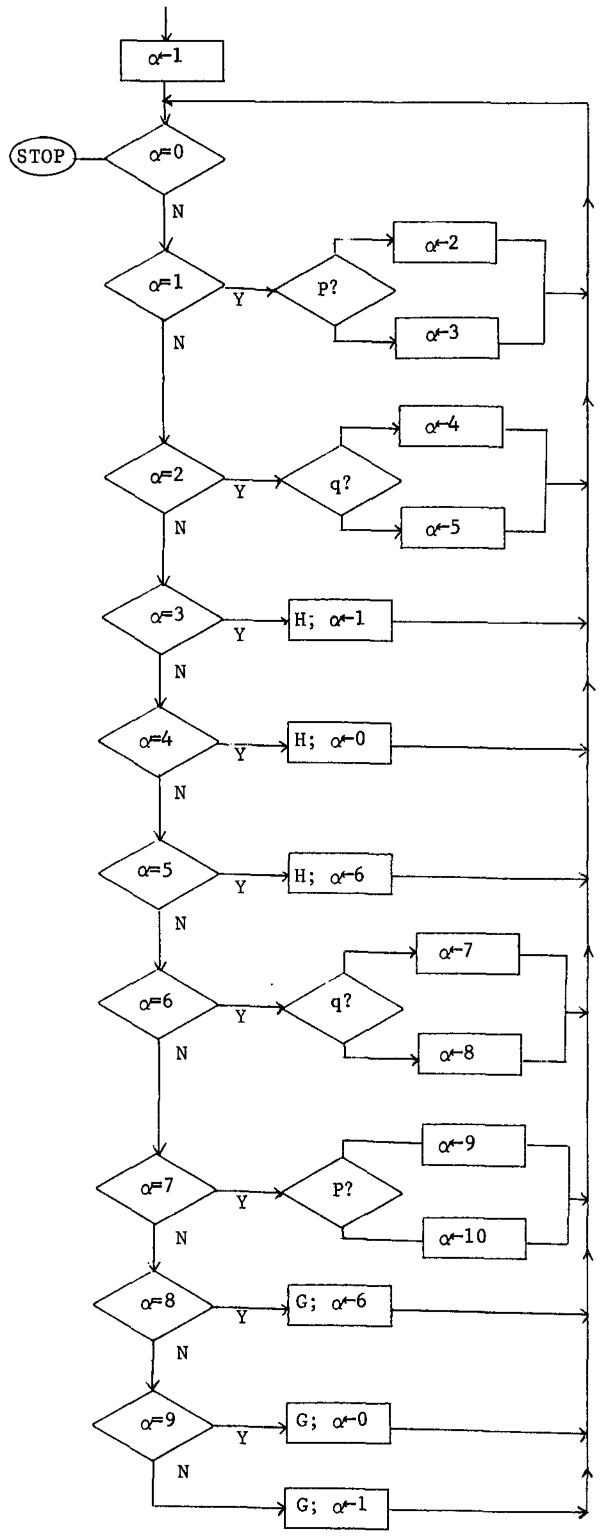
Suppose some flow chart program contains a set of process boxes assigned arbitrary integer labels $i_1, i_2, \ldots, i_n$, and decision boxes assigned arbitrary integer labels $i_{n+1}, i_{n+2}, \ldots, i_m$. (By convention assume the STOP box is assigned the label zero, and the entry box is assigned the label one.) For each process box, $i_j$, create a new box, $i_j'$, identical to the former except for the addition of the assignment '$\alpha \leftarrow i_k'$' where $i_k$ is the label of the successor of $i_j$ in the original program. For each decision box, $i_\ell$, create the macro box, $i_\ell'$,



where $i_t$ and $i_f$ are the labels of the successors of the true and false branches of the decision box, $i_\ell$, in the original program. Now create the following flow chart:



Thus, for example, the Ashcroft and Manna example given earlier (the labels are given on the earlier diagram) becomes:



Constructions such as the one given above are undesirable not only because of their inefficiency, but because they destroy the topology (loop structure) and locality of the original program and thus make it extremely difficult to understand. Nevertheless, the construction serves to illustrate the point that adding (at least one) control variable is an effective device for eliminating the goto. Ashcroft and Manna have given algorithms for translating arbitrary programs into goto-less form (with additional variables) which preserve the efficiency and topology of the original program.

## THE PRACTICAL POSSIBILITY OF ELIMINATING THE GOTO

As discussed in the previous section, it is theoretically possible to eliminate the goto. Moreover, there can be little quarrel with the objectives of the constructive programming methodology. A consequence of the particular methodology presented above is that it produces goto-less programs, thus the goto is unnecessary in programs produced according to this methodology. A key, perhaps the key, issue, then, is whether it is practical to remove the goto. In particular there is an appropriate suspicion among practicing programmers* that coding without the goto is both inconvenient and inefficient. In this section we shall investigate these two issues, for, if it is inconvenient or grossly inefficient to program without the goto then the practicality of the methodology is in question.

### Convenience:

Programming without the goto is not (necessarily) inconvenient. The author is one of the designers, implementors, and users of a 'systems implementation language', Bliss (16,17,18); Bliss does not have goto. The language has been in active use for three years; we have thus gained considerable practical experience programming without the goto. This experience spans many people and includes several compilers, a conversational programming system (APL), an operating system, as well as numerous applications programs.

The inescapable conclusion from the Bliss experience is that the purported inconvenience of programming without a goto is a myth! Programmers familiar with languages in which the goto is present go through a rather brief and painless adaptation period. Once passed this adaptation period they find that the lack of a goto is not a handicap; on the contrary, the invarient reaction is that the enforced discipline of programming without a goto structures and simplifies the task.

Bliss is not, however, a simple goto-less language; that is, it contains more than simple while-do and if-then-else (or case) constructs. There are natural forms of control flow that occur in real programs which, if not explicitly provided for in the language, either require a goto so that the programmer may synthesize them, or else will cause the programmer to contort himself to mold them into a goto-less form (e.g., in terms of the construction in the previous section). Contortion obscures and is therefore antipathetic with the

*Including this author when he first read Dijkstra's letter in 1968.

constructive philosophy; hence the approach in Bliss has been to provide explicit forms of these natural constructs which are also inherently well-structured. In (19) the author analyzes the forms of control flow which are not easily realized in a simple goto-less language and uses this analysis to motivate the facilities in Bliss. Here we shall merely list some of the results of that analysis as they manifest themselves in Bliss (and might manifest themselves in any goto-less language):

1. A collection of 'conventional' control structures: Many of the inconveniences of a simple goto-less language are eliminated by simply providing a fairly large collection of more-or-less 'conventional' control structures. In particular, for example, Bliss includes: control 'scopes' (blocks and compounds), conditionals (both if-then-else and case forms), several looping constructs (including while-do, do-while, and stepping forms), potentially recursive procedures, and co-routines.

2. Expression Language: As noted in an earlier section, one mechanism for expressing algorithms in goto-less form is through the introduction of at least one additional variable. The value of this variable serves to encode the state of the computation and direct subsequent flow. This is a common programming practice used even in languages in which the goto is present (e.g., the FORTRAN 'computed goto'). Bliss is an 'expression language' in the sense that every construct, including those which manifest control, is an expression and computes a value. The value of an expression (e.g., a block or loop) forms a natural and convenient implicit state variable.

3. Escape Mechanism: Analysis of real programs strongly suggests that one of the most common 'good' uses of a goto is to prematurely terminate execution of a control environment--for example, to exit from the middle of a loop before the usual termination condition is satisfied.* To accommodate this form of control Bliss allows any expression (control environment) to be labeled; an expression of the form "leave <label> with <expression>" may be executed within the scope of this labeled environment. When a leave expression is executed two things happen: (1) control immediately passes to the end of the control environment (expression) named in the leave, and (2) the value of the named environment is set to that of the <expression> following the with. Note that the leave expression is a restricted form of forward branch just as the various forms of loop constructs are restricted backward jumps. In both cases the constructs are less general, and less dangerous, than the general goto.

*A somewhat different form of the Bliss escape is described in (19); the form described in (19) has been replaced by that described above.

In summary, then, our experience with Bliss supports the notion that programming without the goto is no less convenient than with it. This conclusion rests heavily on the assumption that the goto was not merely removed from some existing language, but that a coherent selection of well-structured constructs were assembled as the basis of the control component of the new language. It would be unreasonable to expect that merely removing the goto from an existing language, say FORTRAN or PL/I, would result in a convenient notation. On the other hand, it is not unreasonable to expect that a relatively small set of additions to an existing language, especially the better structured ones such as Algol or PL/I, could reintroduce the requisite convenience. While not a unique set of solutions, the control mechanisms in Bliss are one model on which such a set of additions might be based.

Efficiency:

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason--including blind stupidity. One of these sins is the construction of a "rat's nest" of control flow which exploits a few common instruction sequences. This is precisely the form of programming which must be eliminated if we are ever to build correct, understandable, and modifiable systems.

There are applications (e.g., 'real time' processing) and there are (a few) portions of every program where efficiency is crucial. This is a real issue. However, the appropriate mechanism for achieving this efficiency is a highly optimizing compiler, not incomprehensible source code. In this context it is worth noting another benefit of removing the goto--a benefit which the author did not fully appreciate until the Bliss compiler was designed--namely, that of global optimization. The presence of goto in a block-structured language with dynamic storage allocation forces run-time overhead for jumping out of blocks and procedures and may imply a distributed overhead to support the possibility of usch jumps. Eliminating the goto removes both of these forms of overhead. More important, however, is that: (1) the scope of a control environment is statically defined, and (2) all control appears as one of a small set of explicit control constructs. A consequence of (1) is that the Fortran-H compiler (20), for example, expends a considerable amount of effort in order to achieve roughly the same picture of overall control as that implicit in the text of a Bliss program. The consequence of (2) is that the compiler need only deal with a small number of well defined control forms; thus failure to optimize a peculiarly constructed variant of a common control structure is impossible. Since flow analysis is pre-requisite to global optimization, this benefit of eliminating the goto must not be underestimated.

## SUMMARY

One goal of our profession must be to produce large programs of predictable reliability. To do this requires a methodology of program construction. Whatever the precise shape of this methodology, whether the one sketched earlier or not, one property of that methodology must be to isolate (sub) components of a program in such a way that the proof of the correctness of an abstraction from these components can be made independent of both their implementation and the context in which they occur. In particular this implies that unrestricted branching between components cannot be allowed.

Whether or not a language contains a goto and whether or not a programmer uses a goto in some context is related, in part, to the variety and extent of the other control features of the language. If the language fails to provide important control constructs, then the goto is a crutch from which the programmer may synthesize them. The danger in the goto is that the programmer may 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. In the latter case, both the human and the compiler will do a better job of interpreting them.

## REFERENCES

1. Dijkstra, E. W., "Goto Statement Considered Harmful", Letter to the Editor, CACM, 11, 3, March 1968.

2. Dijkstra, E. W., "A constructive approach to the problem of program correctness", BIT 8, 1968.

3. Dijkstra, E. W., "Structured programming", Software Engineering, October 1969, Rome.

4. Dijkstra, E. W., "Notes on Structured Programming", August 1969.

5. Naur, P., "Proof of algorithms by general snapshots", BIT 6, 1966.

6. Naur, P., "Programming by action clusters", BIT 9, 1969.

7. Hoare, C. A. R., "Proof of a program FIND", CACM 14, 1, June 1971.

8. Wirth, N., "Program development by stepwise refinement", CACM, April 1971.

9. Parnas, D. L., "Information distribution aspects of design methodology", IFIP, 1971.

10. King, J., A Program Verifier, Ph.D. Thesis, Carnegie-Mellon University, 1969.

11. Manna, Z., Termination of Algorithms, Ph.D. Thesis, Carnegie-Mellon University, April 1968.

12. Manna, Z., "The correctness problem of computer programs", Computer Science Research Review, 1968.

13. Van Wijngaarden, A., "Recursive Definition of Syntax and Semantics", in Formal Language Description Languages, (T. B. Steel, ed.), North-Holland Publishing Col, Amsterdam, 1966.

14. Knuth, Floyd, Notes on Avoiding 'GOTO' Statements, Technical Report CS 148, Stanford University, January 1970.

15. Ashcroft, E. and Manna, Z., "The translation of "goto" programs into "while" programs, IFIP, 1971.

16. Wulf, et al., Bliss Reference Manual, Computer Science Department Report, Carnegie-Mellon University.

17. Wulf, et al., "Bliss: a language for systems programming", CACM, December 1971.

18. Wulf, et al., "Reflections on a systems programming language", Proceedings of the SIGPLAN Symposium on Systems Implementation Languages, October 1971.

19. Wulf, W. A., "Programming without the goto", IFIP, 1971.

20. Lowery and Medlock, "Object code optimization", CACM, 12, 1, January 1969.

21. Böhm and Jacopini, "Flow diagrams, Turing machines, and languages with only two formation rules", CACM, 9, 5, May 1966.

22. Parnas, D., On the Criteria to be Used in Decomposing Systems into Modules, Computer Science Department Report, Carnegie-Mellon University, 1971.

23. Manna, Z., Ness, S., and Vaillemin, J., "Inductive methods for proving properties of programs", SIGPLAN/SIGACT Conference on Proving Assertions about Programs, January 1972.

24. Burstall, R., "An algebraic description of programs with assertions, verification and simulation", SIGPLAN/SIGACT Conference on Proving Assertions about Programs, January 1972.