A Case for the GOTO

Martin E. Hopkins, IBM

In recent years there has been much controversy over the use of the goto statement. This paper, while acknowledging that goto has been used too often, presents the case for its retention in current and future programming languages.

KEY WORDS AND PHRASES: goto, block structure languages, programming style
CR CATEGORIES: 4.22

INTRODUCTION

It is with some trepidation that I undertake to defend the goto statement, a construct, which while ancient and much used has been shown to be theoretically unnecessary (2) and in recent years has come under so much attack (3). In my opinion, there have been far too many goto statements in most programs, but to say this is not to say that goto should be eliminated from our programming languages. This paper contains a plea for the retention of goto in both current and future languages. Let us first examine the context in which the controversy occurs.

A wise philosopher once pointed out to a lazy king that there is no royal road to geometry. After discovering, in the late fifties, that programming was the computer problem, a search was made during the sixties for the royal road to programming. Various paths were tried including comprehensive operating systems, higher level languages, project management techniques, time sharing, virtual memory, programmer education, and applications packages. While each of these is useful, they have not solved the programming problem. Confronted with this unresolved problem and with few good ideas on the horizon, some people are now hoping that the royal road will be found through style, and that banishment of the goto statement will solve all. The existence of this controversy and the seriousness assigned to it by otherwise very sensible people are symptoms of a malaise in the computing community. We have few promising new ideas at hand. I also suspect that the controversy reflects something rather deep in human nature, the notion that language is magic and the mere utterance of certain words is dangerous or defiling. Is it an accident that "goto" has four letters?

Having indicated my belief that this controversy is not quite as momentous as some have made out, it is appropriate to point out some beneficial aspects. First, interest has been focused on programming style and while style is not everything it does have a great deal of importance. Second, the popularity of the no goto rule is, in large part, due to the fact that it is a simple rule which does improve the code produced by most programmers. As we shall see, this is not sufficient grounds for banishing the construct from our languages, although it may well justify teaching alternative methods of programming to beginners or restricting its use on a project. Perhaps the most beneficial aspect of the controversy will be to encourage the use of block structure languages and to discourage use of our most popular languages, COBOL and FORTRAN as they are not well suited to programming without the goto.

The principal motivation behind eliminating the goto statement is the hope that the resulting programs will not look like a bowl of spaghetti. Instead they will be simple, elegant and easy to read, both for the programmer who is engaged in composition and checkout as well as the poor soul attempting future modification. By avoiding goto one guarantees that a program will consist entirely of one-in-one-out control structures. It is easy to read a properly indented program without goto statements, which is written in a block structure language. The possible predecessors of every statement are obvious and, with the exception of loops, all predecessors are higher on the page. (I assume nobody writes inner procedures longer than a page anymore?) Why then should we retain the goto statement in our current and future programming languages?

THEORETICAL CONSIDERATIONS

It has been demonstrated that there are flow charts which cannot be converted to a procedural notation without goto statements [1,4]. It turns out though that this result is not really an argument for the retention of goto as there are means by which a procedure can be rewritten in a systematic manner to eliminate all instances of goto. An almost trivial method is to introduce a new variable which can be thought of as the instruction counter along with tests and sets of this counter. The method is fully described by Bohm and Jacopini [2]. The results of this procedure when applied to a large program with many instances of goto would usually be a program which is less readable than the original program with goto statements. However, nobody seems to be advocating using such unconsidered methods.

The real issue is that theoretical work has

suggested a number of techniques that can be used to rewrite programs, eliminating the instances of goto. These include replication of code (node splitting), the introduction of new variables and tests as well as the introduction of procedures. Any of these techniques, when used with discretion, can increase the readability of code. The question is whether there are any instances when the application of such methods decreases clarity or produces some other undesirable effect. Whether or not to retain the goto does not seem to be a theoretical issue. It is rather a matter of taste, style and the practical considerations of day to day computer use.

ALTERNATIVES TO GOTO

With respect to current languages which are in wide use such as COBOL and FORTRAN, there is the practical consideration that the goto statement is necessary. Even where a language is reasonably well suited to programming without the goto, the elimination of this construct may be at once too loose and too restrictive. PL/I provides some interesting examples here. One exits from an Algol procedure when the flow of control reaches the end bracket. PL/I provides an additional mechanism, an explicit RETURN statement. Consider the table look up in Fig. 1 which is similar to an example of Floyd and Knuth (4). The problem is to find an instance of X in the vector A and if there is no instance of X in A, then make a new entry in A of the argument X. In either case the index of the entry is returned. A count of the number of matches associated with each entry in A is also maintained in an associated vector, B. In this example there are no goto statements but the two RETURN statements cause an exit from both the procedure and the iterative DO. Thus the procedure has control structures which have more than one exit and one-in-one-out control structures were a principal reason for avoiding goto. Should the PL/I programmer add a rule forbidding RETURN? The procedure could then be rewritten as in Fig. 2. This involves the introduction of a new variable, SWITCH, and a new test. If one assumes that the introduction of gratuitous identifiers and tests is undesirable perhaps RETURN is a desirable construct even though it can result in multiple exit control structures. It is my feeling that procedures with several RETURN statements are easy to read and modify because they follow the top to bottom pattern and maintain the obvious predecessor characteristic, while avoiding the introduction of new variables. RETURN is therefore preferable to the alternative of introducing new variables and tests.

However, RETURN is a very specialized statement. It only permits an exit from one level of one type of control, the procedure. One could generalize the construct to apply to multiple levels of control and to DO groups or BEGIN blocks as well as procedures. This is exactly the flavor of the Bliss leave (6) construct. Lacking such language, the PL/I user must content himself with goto. But is this a bad thing? The good programmer, who understands the potential complexity which results from excessive use of goto, will attempt to recast such an algorithm. Failing to find an elegant restatement, he will insert the label and its associated goto out of the desired control structure. The label stands there as a warning to the reader of the routine that this is a procedure with more than the usual complexity. Note also that the label point catches the eye. It is immediately apparent when looking at this statement that it has an unusual predecessor. The careful reader will want to consult a cross reference listing to determine the potential flow of control. Note that the BLISS leave construct is somewhat less than ideal here. In BLISS when one examines the code which follows a bracket terminating a level of control, its potential predecessors are not immediately apparent. One must look upward on the page for its associated label, which indicates a potential unusual predecessor and then find the leave. It is my feeling that unusual exits from levels of control should be avoided. The multiple level case is especially ugly. Where such constructs are necessary, it should be made completely obvious to all. Statements such as the Bliss leave encourage unusual exits from multiple levels of control. One should not cover up the fact that there is an awkward bit of logic by the introduction of a new control construct.

Another interesting PL/I control construct is the ON unit. This is a named block which is automatically invoked on certain events such as overflow, but it can also be invoked explicitly by a statement of the form:

```
SIGNAL    CONDITION (name);
```

The name is established dynamically and need not be declared in the scope of the SIGNAL. This facility often eliminates the need to pass special error return parameters or test a return code which indicates abnormal termination of a lower level procedure. After completion of an ON unit activated by SIGNAL, control is passed back to the statement following the SIGNAL. This is usually not useful. One wants to terminate the signaling block and the only way to do this in PL/I is with a goto out of the ON unit. Is SIGNAL permissible under the no goto criteria? Elimination of goto seems too restrictive here as SIGNAL is a useful facility which can eliminate much messy programming detail. However, the natural consequence of using the SIGNAL statement is to terminate an ON unit with a goto. Perhaps it is best to admit that there is no very good alternative to a goto statement in this situation.

GOTO AS A BASIC BUILDING BLOCK

The lack of a case statement in PL/I is a clear deficiency. The resourceful programmer will construct one out of a goto. This does not make up for the lack of a case statement, but it does point up an interesting and highly legitimate use of goto. One can use it as a primitive to construct more advanced and elegant control structures. Imaginative programmers will, from time to time, develop new control constructs as Hoare invented the case statement (5). Those that are worthwhile will be informally defined and implemented with a macro preprocessor. The better ones will appear in experimental compilers and eventually the best will find their way into the standard languages. Such inventions are often very hard to implement with macro preprocessors for existing languages without use of the goto construct. There is still room for the incorporation of unusual control mechanisms into existing block structure languages. Decision tables are a prime example. One way of handling decision tables is to have a

preprocessor convert them to source language statements. If a convenient translation process introduces goto statements, this is not important as the basic documentation is at the decision table level. The source language is treated as an internal language. The ease of translation is more important than the introduction of goto statements.

Another related reason for retaining goto even in our newest languages is that it is often possible to use a language as the target to which one translates a secondary source language. If the secondary language has goto or even a different set of control constructs, then translation could be very difficult without a goto in the target language. In other words source languages and their associated compilers are useful building blocks for the development of special constructs or languages and elimination of goto decreases the range of usefulness of a language.

GOTO AS AN ESCAPE

Part of the reason for retaining goto is that the the world is not always a very elegant place and sometimes a goto is a useful, if ugly, tool to handle an awkward situation. Algorithms are often messy. Sometimes this may be due to inherent complexity. I suspect, however, that most of the time it is because not enough time or intelligence has been applied. Where time or intelligence are lacking, a goto may do the job. Every program will not be published. Many may be used only once. I tend to sympathize with the programmer who fixes up a one time program at 3:00 a.m. with a goto. Of course, there is always the danger that the programmer will lapse into bad habits but I am willing to take that chance. Perhaps it is an opportunity, for when the intelligent supervisor reads the code of those under him, he can focus on any goto statements. A programmer should be able to justify each use of goto.

I have avoided discussing performance, which like death and taxes, none of us can avoid forever. Suppose a procedure runs too slowly or takes up too much space. A rewrite of the procedure or restructuring of the data may be in order. But if that fails one may be driven to a rewrite in assembly language. There is an intermediate alternative which may solve the problem without resort to an assembler. The programmer who writes structured programs uses certain techniques such as the introduction of procedures and the repetition of code which can result in the loss of time and space. Given the idiosyncracies of many compilers, a little reorganization of code and a few goto statements inserted by a clever programmer can often improve performance. This is not a practice which I recommend for those starting a project, even where it is known to have stringent performance requirements. One should give up a structured program in a higher level language only after performance bottlenecks have been clearly identified and then only give up what is absolutely necessary. My guess is that very few such situations will exist but when they do, a slightly contorted procedure in a higher level language may be an attractive alternative to one written in assembly language. The villain here is the compiler which produces bad code in some situations. Would elimination (as opposed to avoidance) of goto significantly ease the task of compiler writers and thus help us to get better object code? It is difficult to do justice to this problem as there are so many different compiling techniques and some would be helped and some would not. My feeling is that elimination of the goto would not dramatically ease the problems of compiler writers. Even in compilers which do extensive control flow analysis, a small percentage of implementation effort is devoted to that task. A more interesting subject for compiler writers is the identification of those optimizations which improve the performance of programs written with none or very few goto statements. Viewed in this light the existence of well structured programs imposes an additional obligation and more work on compiler writers. This is work which they should eagerly accept so that programmers will not have to make the trade off between a well structured program and one that performs well. More work is required in this area.

VARIETIES OF PROGRAMMING STYLE

The goto issue is part of the larger topic of overall programming style. One of my worries is that we will become the prisoners of one currently fashionable "classical" style. Perhaps other rules of style are better. For example we might say that only a goto which was directed forward was elegant. Perhaps it is useful to restrict ourselves to standard type labels such as "PROC_EXIT". Vagaries of style or fashion need not disturb students who should be taught in a rather constrained way which is established by the teacher. Also, those working on large projects will have to conform to standards. However, experienced programmers and language designers of taste and imagination will want to experiment and they should be encouraged to do so. APL provides an interesting example of a diverse style. A computed goto, in the form of a right pointing arrow exists in APL, but other than function invocation there are no control constructs such as IF THEN ELSE or an iteration statement. Surprisingly one does not get a maze of goto statements in a well written APL function, for the powerful array operators can be used in situations where loops occur in other languages. Sequential execution of statements thus becomes the general rule and few right pointing arrows are required. Whether an algorithm written in APL is clearer than the same algorithm written in a block structure language seems to be a matter on which intelligent people of taste will disagree.

Elegance in programming involves more than avoiding goto, and beyond the goto controversy there are a great many other important issues of style. There is the question about the clarity of array operations in APL and PL/1, as well as structure operations in COBOL and PL/I. To what extent are implicit conversions a subsumption of extreneous detail and in what instances do they produce surprising results? There are many questions about optimal size and complexity with respect to expressions, nesting of IF and iteration statements as well as the size and complexity of procedures. To what extent do declarations properly subsume detail and to what extent do they leave the meaning of a statement unclear unless one is simultaneously examining the declaration? Under what circumstances, if any, should functions have side effects or should iteration replace recursion? To what extent can we eliminate assignment? These and other questions are subtle but important stylistic problems which

we are likely to pass over if we concentrate too heavily on the relatively simple and unimportant issue of goto.

CONCLUSION

goto should be retained in both current and future future languages because it is useful in a limited number of situations. Programmers should work hard to produce well structured programs with one-in-one-out control structures which have no goto statements. Where this is not possible, we should not think that elegance is achieved with a magic language formula. It is far better to admit the awkwardness and use the goto. Furthermore, goto is a useful means to synthesize more complex control structures and increases the usefulness of a language as a target to which other languages can be translated. Viewed in the light of practical programming as an ultimate escape, goto can also be justified if not encouraged. Finally our wisdom has not yet reached the point where future languages should eliminate the goto. If future work indicates that by avoiding goto we can gain some important advantage such as routine proofs that programs are correct, then the decision to retain the goto construct should be reconsidered. But until then, it is wise to retain it.

```
LOOK_UP:
      PROC (X);
            DO  I = 1  TO A_TOP;
                  IF  A(I) = X   THEN
                        DO:
                              B(I) = B(I) + 1;
                              RETURN (I);
                        END;
            END;
            A(I) = X;
            B(I) = 1;
            A_TOP = A_TOP + 1;
            RETURN (I);
      END;
```

Figure 1

```
LOOK_UP2:
      PROC (X);
            SWITCH = 1;
            DO I = 1 TO A_TOP WHILE (SWITCH = 1);
                  IF A(I) = X THEN
                        SWITCH = 0;
            END;
      IF SWITCH = 0  THEN
                  B(I) = B(I) + 1;
            ELSE
                  DO;
                        A(I) = X;
                        B(I) = 1;
                        A_TOP = A_TOP + 1;
                  END;
            RETURN(I);
      END;
```

Figure 2

REFERENCES

[1] Ashcroft, E. and Manna, Z., "The Translation of 'go to' Programs to 'while' Programs", Proc. IFIP Congress 71, Ljubljana, August 1971

[2] Bohm, Corrado and Jacopini, G., "Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules", CACM 9 (Aug. 1966)

[3] Dijkstra, E.W., "GO TO Statement Considered Harmful", letter to the editor, CACM 11, 3 (March 1968)

[4] Knuth, D. E. and Floyd, R. W., "Notes on Avoiding 'GO TO' Statements", Information Processing Letters (1971) 23-31 North-Holland Publishing Co.

[5] Wirth, Niklaus and Hoare, C. A. R., "A Contribution to the Development of Algol", CACM 9, 6 (June 1966)

[6] Wulf, W. A., Russell, D. B., and Habermann, A. N., "Bliss: A Language for Systems Programming", CACM 14, 12 (December 1971)