Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** B **then** A), alternative clauses (**if** B **then** $A1$ **else** $A2$), choice clauses as introduced by C. A. R. Hoare (case[i] of $(A1, A2, \cdots, An)$), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1$, $B2 \rightarrow E2$, $\cdots$, $Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** B **repeat** A or **repeat** A **until** B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommendation to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the **go to** statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jump-less one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

REFERENCES:

1. WIRTH, NIKLAUS, AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June 1966), 413–432.
2. BÖHM, CORRADO, AND JACOPINI, GUISEPPE. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9* (May 1966), 366–371.

EDSGER W. DIJKSTRA
Technological University
Eindhoven, The Netherlands

Language Protection by Trademark Ill-advised

Key Words and Phrases: TRAC languages, procedure-oriented language, proprietary software, protection of software, trademarks, copyright protection, patent protection, standardization, licensing, Mooers doctrine

CR Categories: 2.12, 2.2, 4.0, 4.2

EDITOR:

I would like to comment on a policy published 25 August 1967 by the Rockford Research Institute Inc., for trademark control of the TRAC language "originated by Calvin N. Mooers of that corporation": "It is the belief at Rockford Research that an aggressive course of action can and should be taken to protect the integrity of its carefully designed languages." Mr. Mooers believes that "well-drawn standards are not enough to prevent irresponsible deviations in computer languages," and that therefore "Rockford Research shall insist that all software and supporting services for its TRAC languages and related services be furnished for a price by Rockford, or by sources licensed and authorized by Rockford in a contract arrangement." Mooers' policy, which applies to academic institutions as well as commercial users, includes "authorized use of the algorithm and primitives of a specific TRAC language; authorization for experimentation with the language . . ."

I think that this attempt to protect a language and its software by controlling the name is very ill-advised. One is reminded of the COMIT language, whose developers (under V. Yngve) restricted its source-level distribution. As a result, that effort was bypassed by the people at Bell Laboratories who developed SNOBOL. This latter language and its software were inevitably superior, and were immediately available to everyone, including the right to make extensions. Later versions benefitted from "meritorious extensions" by "irrepressible young people" at universities, with the result that SNOBOL today is an important and prominent language, while COMIT enjoys relative obscurity.

Mr. Mooers will find that new TRAC-like languages will appear whose documentation, because of the trademark restriction, cannot mention TRAC. Textbook references will be similarly inhibited. It is unfortunate.

BERNARD A. GALLER
University of Michigan
Ann Arbor, Mich. 48104

Mr. Mooer's Reply

EDITOR:

Professor Galler's letter, commenting on our Rockford Research policy statement on software protection of 25 August 1967, opens the discussion of what may be a very significant development to our computing profession. This policy statement applies to our TRAC (TM) computer-controlling languages. The statement includes a new doctrine of software protection which may be generally applicable to a variety of different kinds of complex computer systems, computer services, languages, and software. Already it is evident that this doctrine has a number of interesting legal and commercial implications. It is accordingly appropriate that it be subject to critical discussion.

The doctrine is very simple. For specificity, I shall describe it in regard to the TRAC languages which we have developed: (1) Rockford Research has designated itself as the sole authority for the development and publication of authentic standards and specifications for our TRAC languages; and (2) we have adopted TRAC as our commercial trademark (and service mark) for use in connection with our computer-controlling languages, our publications providing standards for the languages and any other related goods or services.

The power of this doctrine derives from the unique manner in which it serves the interests of the consuming public—the people who will be using computer services. The visible and recognized TRAC trademark informs this public—the engineers, the sociology professors, the business systems people, and the nonprogrammers everywhere—that the language or computer capability identified by this trademark adheres authentically and exactly to a carefully drawn Rockford Research standard for one of our TRAC languages or some related service. This is in accord with a long commercial and legal tradition.

The evils of the present situation and the need to find a suitable remedy are well known. An adequate basis for proprietary software development and marketing is urgently needed, particularly in view of the doubtful capabilities of copyright, patent, or "trade secret" methods when applied to software. Developers of valuable systems—including languages—deserve to have some vehicle to give them a return. On the user side the nonexistence of standards in the computer systems area is a continuing nuisance. The proliferation of dialects on valuable languages (e.g. SNOBOL or FORTRAN) is sheer madness. The layman user (read "nonprogrammer") who now has access to any of several dozen computer facilities (each with incompatible systems and dialects) needs relief. It is my opinion that this new doctrine of autonomous standardization coupled with resort to commercial trademark can provide a substantial contribution to remedying a variety of our problems in this area.

Several points of Professor Galler's letter deserve specific comment. The full impact of our Rockford Research policy (and

Taulbee Survey Report

I was disappointed in the report by David Gries on the 1984–1985 Taulbee Survey (*Communications*, October 1986, pp. 972–977). Although it was well presented, reasonably laid out and, most likely, accurate, it was not useful information. Data in this form need commentary to become information. I often hear of "industry eating its own seed corn" in reference to the hiring of Ph.D.'s away from academia, and of a shortfall in Ph.D.'s for computer science overall. I jumped at the chance to learn from the Gries report. Alas, there were no conclusions drawn, no help for all us uninformed. I know that time spent pouring over the data would give me some feel for the condition I am concerned over (e.g., potential lack of sufficient Ph.D.'s), but I know I do not have the time and I fear I lack the knowledge to draw proper conclusions.

Roger S. Gourd
MASSCOMP
One Technology Park
Westford, MA 01886

Response:

Perhaps reader Gourd is right in asking for more commentary and conclusions. Inexperience, a reluctance to draw too many conclusions, and a lack of space all contributed to the form and content of the report. We will try to address this criticism in the next report.

David Gries
Department of Computer Science
Cornell University
405 Upson Hall
Ithaca, NY 14853-7501

Network Noted

In the "Notable Computer Networks" article by John S. Quarterman and Josiah C. Hoskins (*Communications*, October 1986, 932–971) a few company networks are detailed. One such network which is not detailed seems to be a fairly well-kept secret. This is the internal network belonging to Tandem Computers Incorporated. This network has 200 NonStop hosts connected via 150 links consisting of microwave, laser, satellite, fiber, and copper running at speeds up to 3 Mbit/s. The aggregate processing power of this virtual machine is 1.6 BIPS (billion instructions per second). Both the systems and the network are fault-tolerant.

A staff of four employees in Cupertino, CA, and one in Germany support the user community of 6500 hard-wired and 2500 dial-up terminals and PCs. While the network, spanning 23 countries, is running 24 hours a day, the support staff works normal 40 hour weeks. Because of its fault-tolerant nature, communications failures are not critical to network connectivity.

This ease of maintainability is due to Tandem's proprietary protocol, EXPAND, which is modeled after X.25. Addition, deletion or moves of hosts do not require a Network Sysgen. When a new host is added to the network, a "ripple effect" takes place until each host knows the best path to the new host. During a network failure and after the subsequent recovery, the network performs its own rerouting.

The network supports over 100 production applications including Electronic Mail, Order Entry, Manufacturing, VLSI Design, Customer Engineering Dispatch, Problem Reporting and Software Patch Distribution.

A typical Tandem electronic-mail name looks like 'LaPedis_Ron', or 'Payroll', the second being a department name rather than a person. There is no need to specify the geographical location of a mail correspondent.

An on-line telephone book, telephone messages, and request form application round out the average employee's interface with the network. An article on the Tandem network has appeared in *Data Communications* magazine (August and September 1985).

Ron LaPedis
Corinne DeBra
Tandem Computers Incorporated
19191 Vallco Parkway
Cupertino, CA 95014-2599

"GOTO Considered Harmful" Considered Harmful

The most-noted item ever published in *Communications* was a letter from Edsger W. Dijkstra entitled "Go To Statement Considered Harmful" [1] which attempted to give a reason why the **GOTO** statement might be harmful. Although the argument was academic and unconvincing, its title seems to have become fixed in the mind of every programming manager and methodologist. Consequently, the notion that the **GOTO** is harmful is accepted almost universally, without question or doubt. To many people, "structured programming" and "**GOTO**-less programming" have become synonymous.

This has caused incalculable

harm to the field of programming, which has lost an efficacious tool. It is like butchers banning knives because workers sometimes cut themselves. Programmers must devise elaborate workarounds, use extra flags, nest statements excessively, or use gratuitous subroutines. The result is that **GOTO**-less programs are harder and costlier to create, test, and modify. The cost to business has already been hundreds of millions of dollars in excess development and maintenance costs, plus the hidden cost of programs never developed due to insufficient resources.

I have yet to see a single study that supported the supposition that **GOTO**s are harmful (I presume this is not because nobody has tried). Nonetheless, people seem to need to believe that avoiding **GOTO**s will automatically make programs cheap and reliable. They will accept any statement affirming that belief, and dismiss any statement opposing it.

It has gone so far that some people have devised program complexity metrics penalizing **GOTO**s so heavily that any program with a **GOTO** is *ipso facto* rated more complex than even the clumsiest **GOTO**-less program. Then they turn around and say, "See, the program with **GOTO**s is more complex." In short, the belief that **GOTO**s are harmful appears to have become a religious doctrine, unassailable by evidence.

I do not know if I can do anything that will dislodge such deeply entrenched dogma. At least I can attempt to reopen the discussion by showing a clearcut instance where **GOTO**s significantly reduce program complexity.

I posed the following problem to a group of expert computer programmers: "Let X be an $N \times N$ matrix of integers. Write a program that will print the number of the first all-zero row of X, if any."

Three of the group regularly used **GOTO**s in their work. They produced seven-line programs nearly identical to this:

```
for i := 1 to n
do begin
  for j := 1 to n do
    if x[i, j]<>0
      then goto reject;
  writeln
('The first all-zero
                    row is ', i);
  break;
reject: end;
```

The other ten programmers normally avoided **GOTO**s. Eight of them produced 13- or 14-line programs using a flag to indicate when an all-zero row was found. (The other two programs were either incorrect or far more complex.) The following is typical of the programs produced:

```
i := 1;
repeat
  j := 1;
  allzero := true;
  while (j<=n) and allzero
  do begin
    if x[i, j]<>0
      then allzero := false;
    j := j + 1;
  end;
  i := i + 1;
until (i>n) or allzero;
if i<=n
   then writeln
('The first all-zero
               row is ', i - 1);
```

After reviewing the various **GOTO**-less versions, I was able to eliminate the flag, and reduce the program to nine lines:

```
i := 1;
repeat
  j := 1;
  while (j<=n)
  and (x[i, j] = 0) do
    j := j + 1;
  i := i + 1;
until (i>n) or (j>n);
if j>n
   then writeln
('The first all-zero
                row is', i - 1);
```

By any measure not intentionally biased against **GOTO**s, the two **GOTO**-less programs are more complex than the program using **GOTO**s. Aside from fewer lines of code, the program with **GOTO**s has only 13 operators, compared to 21 and 19 for the **GOTO**-less programs, and only 41 total tokens, compared to 74 and 66 for the other programs. More importantly, the programmers who used **GOTO**s took less time to arrive at their solutions.

In recent years I have taken over a number of programs that were written without **GOTO**s. As I introduce **GOTO**s to untangle each deeply nested mess of code, I have found that the number of lines of code often drops by 20–25 percent, with a small decrease in the total number of variables. I conclude that the matrix example here is not an odd case, but typical of the improvements that using **GOTO**s can accomplish.

I am aware that some awful programs have been written using **GOTO**s. This is often the fault of the language (because it lacks other constructs), or the text editor (because it lacks a block move). With a proper language and editor, and adequate instruction in the use of **GOTO**, this should not be a consideration.

All of my experiences compel me to conclude that it is time to part from the dogma of **GOTO**-less programming. It has failed to prove its merit.

Frank Rubin
The Contest Center
P.O. Box 1660
Wappingers Falls, NY 12590

REFERENCE

1. Dijkstra, E.W. "Go to statement considered harmful." *Commun. ACM 11*, 3 (Mar. 1968), 147–148.