CSc 372 — Comparative Programming Languages
Spring 2017 (McCann)

The Great CACM GOTO Argument

In 1968, Professor Edsger W. Dijkstra wrote a letter to the editors of the *Communications of the ACM*, the flagship publication of the Association for Computing Machinery. He titled it "A Case against the GO TO Statement," but the editor re-titled it "Go To Statement Considered Harmful." It grew to become one of the most cited items in the magazine's history.

- Dijkstra, Edsger W. "Letters to the Editor: Go to Statement Considered Harmful." Communications of the ACM, Volume 11 Issue 3, March 1968, pp. 147-8. `https://doi.org/10.1145/362929.362947`

In 1987, Frank Rubin submitted a letter to the editor titled " 'GOTO Considered Harmful' Considered Harmful," in which he argued that the GOTO statement is often very useful. It was published in the March issue.

- Rubin, Frank. "ACM Forum: 'GOTO Considered Harmful' Considered Harmful," Communications of the ACM, Volume 30, Number 3, March 1987, pp. 195-6. `https://doi.org/10.1145/214748.315722`

Rubin's letter prompted a flurry of responses that were published in the May, August, and December 1987 issues. These responses included a response from Rubin to the flurry, a response from Dijkstra to the flurry, and a reponse from Rubin to Dijkstra's response. We can only imagine the volume and tone of the responses and rejoinders had this occurred today on social media.

- Ashenhurst, Robert L., Ed. "ACM Forum," Communications of the ACM, Volume 30, Number 5, May 1987, pp. 350–5. `https://doi.org/10.1145/22899.315729`
- Ashenhurst, Robert L., Ed. "ACM Forum," Communications of the ACM, Volume 30, Number 8, August 1987, pp. 658–62. `https://doi.org/10.1145/27651.315742`
- Ashenhurst, Robert L., Ed. "ACM Forum," Communications of the ACM, Volume 30, Number 12, December 1987, pp. 996–9. `https://doi.org/10.1145/33447.315758`

The following pages are from scans of the original pages in the CACM. I expect you'll find these letters to be interesting reading, particularly from your points of view as programmers who were taught programming long after the GOTO statement ceased to be an option in most new languages.

> "At the IFIP Congress in 1971 I had the pleasure of meeting Dr. Eiichi Goto of Japan, who cheerfully complained that he was always being eliminated."
>
> — Donald Knuth, in "Structured Programming with `go to` Statements," Computing Surveys, Vol. 6, No. 4, December 1974.

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 jumpless 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.

acm forum

PCs and CPs

This letter is in response to the February 1987 President's Letter in *Communications* ("Personal Computers and Computing Professionals," pp. 101–102). Right on and write on, Paul Abrahams. Last summer (1986), ACM-SIGGRAPH awarded me an educational resource grant to assist with the creation of computer art workshops for high school and middle school children. Since attending SIGGRAPH '86, I have gone into the classrooms of our children. These young people know a great deal about personal computers, video technology, computer music, . . . the electronic world. Their heroes, in some cases, are the hackers and computer wizards to whom Abrahams refers in his letter. I have been able to reach young people and have received the support of Parent Teacher Associations (PTAs) for my use of personal computers in the creation of computer art. SIGGRAPH Video Reviews are exciting for children to watch, but the opportunity to see and do creative work on an Apple II, Amiga or Macintosh goes a long way in educating children. As a result, it seems like a great idea for ACM to find a place for the hardware tinkers and software wizards who have made such a wonderful contribution to the development of young people.

Theresa-Marie Rhyne
Computer Artist/Art Educator
P.O. Box 3446
Stanford, California 94305

View from Watergate Bridge

The Forum strives for balanced presentation. One way to achieve this is by soliciting responses to received letters. Another is to publish all or a representative sampling of subsequent reader responses to letters. The former expedient was followed for the letter from Herb Grosch, to which the following response refers. The latter expedient is adopted here, the "balance" being perhaps skewed by the fact that this was the only response received. The editor accepts full responsibility for delaying its publication somewhat until it seemed reasonably certain that no more responses were forthcoming.
—R. L. Ashenhurst.

While reading Herb Grosch's letter in a recent ACM forum ("An ACM Watergate," *Communications*, Oct. 1986, p. 928–930), I was reminded of an old Dutch expression that my late father used for this sort of situations: "Vechten tegen de bierkaai," he used to say. It meant that no matter how hard one fought and argued and obtained agreements, the thing would crop up again and again. It was a fight without an end. And that is what the ACM has become.

For those of us who have been convinced of the necessity of Chapters and have been fighting for twenty years now for Chapter Rights and to make life more bearable for the common programmer, Herb is the only visible and audible voice left, it seems. Most of us gave up after the Council elections of 1982 and stopped paying dues. I still pay my dues every year and will for as long as Herb is on the Council. Unless they kick me out once this piece is published.

The publications boys in New York have tricks up the kazoo in order to protect their jobs. It has happened to me and to others that a piece is put "on hold" for publication until the establishment has thought of enough smart answers for publishing the piece with their comments. But the original author does not see their comments until he reads them in *Communications*. And if he then tries to get a rebuttal published, it is refused "because there is no sense in dragging it out," as I was once told after inquiring. We now read that the same thing again has befallen Herb Grosch. It's the secrecy that gets ye! They only do what they are legally obligated to and not what is morally right. I know: that is hard to prove, and they probably will scream of slander and libel and threaten legal action because their usual response is to hide behind the law and the rules of the Association. It's the way that the staff interprets figures and doctors up reports, holding the interesting stuff close to their chests and publishing good-to-them items only.

Slowly it's becoming impossible to say anything or ask questions anymore. Over the years the staff and the Council have become sacred and we, the rank-and-file members, we are the sacred jack-

asses who have let them become that holy in the first place.

It may be true that the total number of members is at an all time high, as Adele Goldberg states. And as long as the sign-up rate of new members is higher that the drop-out rate of old members, that number will continue to rise. But the number is deceiving. More than half the membership is Associate and Student members who have no vote in the ACM. We advertise some 300 Chapters but that number is also deceiving. Some 200 are Student Chapters, and you know how it is at school: if the professor says that it will help your grade if you pay nine dollars for ACM student membership, especially "if you are a borderline case" ("and you are all borderline," he adds!), then the whole class joins the ACM. However, not many become full-fledged ACM members after they have received their diplomas. As far as Regular Chapters go, perhaps some 60 of them show some degree of activity. The rest have died since 1982 because the leaders were burnt out by a lack of administrative and financial support from the National organization. Long-time members drop out because of disappointment in the ACM. Some months the number of members who do not renew their memberships is huge. That is what Herb refers to when he speaks of membership falling off. And that was also the reason why they were talking merger with the IEEE there for a while.

To maintain an oversized office in a high rent area costs hands full of money. That is the main reason why Chapter services have been cut to practically nothing. In order to get funds for Chapters and the common programmer, I suggest getting that office out of Manhattan and moving it west. This will accomplish two purposes: lower rent, and half of the staff will quit. Then we will have money for Chapters and local activities.

Jan Matser
ACM Arrowhead Chapter Chair (1967)
ACM San Francisco Peninsula Chair (1977)

"'GOTO Considered Harmful' Considered Harmful" Considered Harmful?

I enjoyed Frank Rubin's letter ("'**GOTO** Considered Harmful' Considered Harmful," March 1987, pp. 195–196), and welcome it as an opportunity to get a discussion started. As a software engineer, I have found it interesting over the last 10 years to write programs both with and without **GOTO** statements at key points. There are cases where adding a **GOTO** as a quick exit from a deeply nested structure is convenient, and there are cases where revising to eliminate the **GOTO** actually simplifies the program.

Rubin's letter attempts to "prove" that a **GOTO** can simplify the program, but instead proves to me that his implementation language is deficient. In the first solution example the **GOTO** programmers got the answer very effectively with no wasted effort:

```
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;
```

In the consolidated second example, the GOTO-less version seems somewhat more complex, even after the subscript beyond the end of the array is exchanged for a binary flag to determine the result:

```
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);
```

Both programs, however, serve to point out a missing feature of the language. In the first, the automatic incrementation of a counter is used, but the end condition cannot be tested with the loop construct. In the second, the loop construct tests for end condition, but cannot then increment the counter.

The ideal would be to take both good ideas and use them in combination:

```
found := false;
for i := 1 to n while (^
                    found)
  do for j := 1 to n
       while (x[i, j] = 0)
    do if j = n then
            found := true;
if found then
  writeln('The first all
       zero row is ', i);
```

This is not a legal program in Pascal, but the ability to use both a counter and a condition in the loop construct makes the entire job much simpler. The loop counting is done (correctly) by the looping construct, as is the exit testing. I have included a flag to avoid depending on the value of a loop index after exhausting the count, which could be undefined. If a language specifies the counter to be left one past the end of range, this flag would not be needed.

I generally prefer **GOTO**-less code, but will disagree with anyone who thinks there are no valid

uses for the **GOTO** in practical engineering. The **GOTO** statement can be easily misused and should therefore be avoided. The hand-coded counters in the second example are also easily misused and should be avoided whenever possible.

The **IF** and **GOTO** are a minimum subset of control flow features, to which the programmer can return when the "correct" feature is not available. **GOTO**, hand coded counters, and extra flags should all be avoided when possible because their use is error prone. I would like to challenge language designers to make the **GOTO** useless by allowing its use and then providing "better alternatives" for each situation where a **GOTO** is needed to work around a language limitation.

Donald Moore
Prime Computer, Inc
192 Old Connecticut Path
Framingham, MA 01701

It was with a mixture of dismay and exasperation that I read Frank Rubin's letter to the Forum. I was dismayed to see this dead horse beaten once again, and exasperated by Rubin's sweeping claims about the virtues of the **GOTO** statement.

This is primarily a religious issue, and those of us who oppose the **GOTO** statement have little hope of converting those who insist on using it. To be sure, the statement has its place in programming, but, recalling Rubin's reference to butcher knives, it should be used only with great care. The fundamental problem is that a programmer, when encountering a **GOTO** in some fragment of code, is forced to begin a sequential search of the entire program to determine where the flow of control has gone. Even in Rubin's simplistic example I had to read the code twice to find the label he was jumping to.

Obviously, an occasional need arises for some type of **GOTO** statement. The solution is for the programming language to provide a **GOTO** statement which has restricted semantics, making it possible to easily determine the target of the desired branch. For example, here is Rubin's example program (determining the first all-zero row of an $N \times N$ matrix of integers), written in C:

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    if (x[i, j] != 0)
      break;
  if (j < n) {
    printf(''The first
          all-zero row is
             %d\n'', i);
    break;
    }
}
```

This fragment has two **GOTO** statements, both named **break**. [*Note: Rubin's program had the second **break** but not the first—Ed.*] **break** has the effect of jumping to the statement following the innermost loop enclosing the **break** statement. In both uses, the effect of a **GOTO** has been achieved, but the restricted semantics of **break** allow the programmer to easily determine the destination of the branch.

I contend that my version of this program is far more understandable than either of Rubin's programs, with or without **GOTO**. In fact, Mr. Rubin's examples of **GOTO**-less programming do more to highlight a problem in Pascal (which has no **BREAK** statement) than they do to convince me that a **GOTO** statement is required. He starts with an absolutely egregious program, and "improves" it by removing a flag. Here is my attempt at a **GOTO**-less version of the same program, in Pascal:

```
i := 1;
done := false;
while i <= n and not done
                        do
  begin
  j := 1;
  while j <= n and x[i, j]
                    = 0 do
    j := j + 1;
  if j <= n then
    begin
    writeln(''The first
      all-zero row is i);
    done := true
    end;
  i := i + 1
  end;
```

For lack of a **BREAK** statement, I had to use a flag to terminate the outer **while** loop. Unlike Rubin, I did not mix **while** and **repeat** loops, which is confusing, nor did I force the variable *i* to serve dual roles, indexing the array and pointing to the row following the first all-zero row. While I prefer my C version of this program, I would still stand my Pascal against any of Rubin's attempts.

The conclusion to be drawn from this exercise is that good **GOTO**-less code can almost always be written to be better than any equivalent code containing **GOTO**s. Contrary to Mr. Rubin's claims, I (and many others) have had many experiences trying to debug and maintain someone else's code containing **GOTO**s, and have yet to come away from such an experience feeling good about the individual who wrote the original code.

Chuck Musciano
Lead Software Engineer
Harris Corporation
PO Box 37, MS 3A/1912
Melbourne, FL 32902

My congratulations to Frank Rubin for coming out of the closet on "**GOTO**-less" programming. As a professional programmer for many years, I have read and lis-

tened to all the arguments in favor of **GOTO**-less programming, hoping that one of them would convince me to give up **GOTO**s. None has so far succeeded. Such an argument would have to show that **GOTO**s always violate the structure of a program even when they are used in accordance with good programming practices. Obviously **GOTO**s are misused, but it is usually not much easier to untangle heavily nested code than it is to decipher spaghetti code.

Both the overuse and the total elimination of **GOTO**s constitute misunderstandings of the relationship among syntactic elements in a programming language. **GOTO**s transfer control just like other, related transfer commands (e.g., **IF**. . .**THEN**). Hence, they should be used when other forms would be inappropriate—by leading to needlessly complex code, for instance. A linguistic analogy can be found in active and passive sentences. Active sentences are easier to produce and understand in relation to their passive counterparts. A "passive-less" English would certainly lead to simpler (better?) structures. However, most linguists would agree that English would loose a portion of its expressive power.

Finally, I will continue to do what I have always been doing: listening to **GOTO**-less arguments and writing well-organized and commented software that makes appropriate use of *all* available features of a programming language.

Michael J. Liebhaber
Child Language Program
University of Kansas
1043 Indiana
Lawrence, KS 66044

Frank Rubin's letter stated that ". . . **GOTO**-less programs are harder and costlier to create, test, and modify." He describes Dijkstra's original letter on the subject (*Communications*, March 1968, pp. 147–148) as ". . . academic and unconvincing . . ." without any support or justification. Finally, he concludes with some example programs which purport to illustrate the logical simplicity of programs which freely use **GOTO** plus BREAK contructs.

Example programs are claimed to fit the sample specification "Let X be an $N \times N$ matrix of integers. Write a program that will print the first all-zero row of X, if any." I had to make several assumptions in order to write the sample program:

1) the language does not support partial evaluation of logical expressions,
2) performance of the final product is not an issue, and
3) performance in the absence of any all-zero row is not specified—in particular, termination is not required.

Apparently, there are also several additional unstated assumptions:

1) the algorithm should test as few elements of matrix X as necessary,
2) the algorithm need not be easily changed to meet a different specification,
3) the language does not support recursion or multiple procedures,
4) the language does support both **GOTO** and **BREAK**, and
5) the program should terminate if a non-all-zero row is found.

Rubin's first example, of a program ". . . where **GOTO**s significantly reduce program complexity," will not run on my UCSD 1.1 **Pascal** system. My Pascal has no **BREAK** statement. This, however, can be circumvented by use of an *additional* **GOTO** and label as follows:

```
    .
    .
  writeln
  ('the first all zero
            row is ', i);
    goto break
  reject: end;
  break: (*etc.*)
```

By violating all of the *unstated* assumptions, I was able to produce some relatively pleasant solutions to this problem, none of which caused me "to use extra flags, nest statements excessively, or use gratuitous subroutines."

The first solution tests additional elements of the matrix X as necessary, is easily changed to meet a different specification, uses multiple procedures, and does not use either GOTO or BREAK:

```
function allZero:boolean;
var
  az:boolean;
begin az := true;
  for j := 1 to n do
    az := az AND (x[i, j] =
                        0);
  allZero := az
end;

procedure firstZero;
begin i := 1;
  while not allZero do i :=
                      i + 1;
  WRITELN('First all zero
              row is ', i)
end;
```

The second solution uses recursion. With a minor change, the recursive solution tests minimal values of X. Many reject recursion as a viable candidate, but recent evidence [2] confirms that recursion is indeed faster for many classes of problems.

```
function allZero(i, j:
                  integer):
boolean;
begin
  if j > n then
    allZero := true
  else
    allZero := (x[i, j] =
    0) and allZero(i,
                    j + 1)
end;
```

```
procedure firstZero(i:
                    integer);
begin
  if i ≤ n then
    if allZero(i, 1) then
      writeln(''First all
        zero row is ', i)
    else
      firstZero(i + 1)
  else
    writeln('No all zero
                    row')
end;
```

It seems that Rubin takes issue with the complexity of deeply nested control structures. Recent work [3] sheds some light on ways to cope with such problems. In general, poor program layout results from a failure to understand an algorithm, not from the language or from the specific techniques used for implementation.

I submit that there are two issues here:

1) Poor and good programming are language independent. That Rubin is able to reduce the complexity of poor programs is not an indictment of the programming style, but rather an indictment of the programmer(s), and a tribute to Rubin's obvious skill.
2) Modifying programs in which there is a "... conceptual gap between the static program and the dynamic process ..." (to quote Dijkstra's original letter) is generally quite difficult. While some advocate scrapping programs instead of patching them ([1] is a recent example), it seems that writing a program as generally as possible can only make it less expensive to modify.

In order to see the real limitations of **GOTO** programming, try to modify the example programs in Rubin's letter. Modifications should include:

1) locating all rows which are all zero,
2) locating and computing an arithmetic mean for all rows which contain nonzero values, and
3) locating all rows in which the sum of the elements is odd.

Steven F. Lott
Computer Task Group
6700 Old Collamer Road
Syracuse, NY 13057

REFERENCES

1. Hekmatpour, S. Experience with Evolutionary Prototyping in a Large Software Project. *Software Engineering Notes 12:1*, 38–41. January 1987.
2. Louden, K. Recursion Versus Non-Recursion in Pascal: Recursion Can Be Faster. *SIGPLAN Notices 22:2*, 62–67 February 1987.
3. Perkins, G. R., R. W. Norman, S. Dancic. Coping with Deeply Nested Control Structures. *SIGPLAN Notices 22:2*, 68–77 February 1987.

I would like to comment on Frank Rubin's article on **GOTO**s. Although I agree with him in spirit, unfortunately he did not give a fair shake to the non-**GOTO** camp for a correct solution. The problem is to find the first row of all zeroes in an $n \times n$ matrix if such a row exists. A simple correct solution can be derived from the English description of the problem/solution. First, a practical definition of an algorithm can be given as:

1) if the current matrix element is equal to zero then look at the next element in the row;
2) if the current matrix element is not equal to zero then look at the first element in the next row;

But WHOOPS, ...

3) if the column number is equal to $n + 1$, then we have found a row with all zeroes, so write out that row number;
4) if the row number is equal to $n + 1$, then we have run out of rows and there are no rows in matrix X that is full of zeroes.

An English-definition of a procedure that accomplishes the above is

FIND(X, *n*, *r*, *c*) =

Returns the row number of the first row of an *n* by *n* matrix X that has all zeroes if such a row exists, or the value of $n + 1$ if the row does not exist. It also

Assumes that all rows whose index is less than *r* have at least one non-zero element, and that row *r* has zeroes as all of its elements from 1 to $c - 1$.

[Assumes $(\forall\, r')$ if $r' < r$ then $X[r'][1..n] \neq \overline{0}$) and $X[r][1..c-1] = \overline{0}$, and gives the first r'' where $r'' \geq r$, $X[r''][1..n] = \overline{0}$, else it gives the value of $n + 1$].

Thus, the LISP-like, tail-recursive definition of "Given an $n \times n$ matrix X, print out the row number of the first row with all zeroes if there exists such a row", is:

FIND(X, *n*, *r*, *c*) = [[[
  $c = n + 1 \rightarrow r$. {from clause 3}
  $r = n + 1 \rightarrow r$. {from clause 4}
  $X[r, c] = 0 \rightarrow$ FIND(X, *n*, *r*, $c + 1$). {from clause 1}
  $X[r, c] \neq 0 \rightarrow$ FIND(X, *n*, $r + 1$, 1). {from clause 2}
]]]

This definition FIND would be run as "FIND(X, *n*, 1, 1)" with *n* already instantiated as some integer. From the definition of FIND, it is easy to write the following program:

```
    .
    .
r := 1;
c := 1;
while (c <> n + 1) and
            (r <> n + 1) do
  if X[r, c] = 0 then
    c := c + 1
  else
    begin
    r := r + 1;
    c := 1
    end;
if r <> n + then
  writelin('Found the
       first row with all
    zeroes, it is :', r);
    .
    .
```

This program was written by putting the recursive clauses in order in a "**if** . . . **then** . . . **else if** . . . etc . . . ," and by putting the escape clauses into the **while** clause predicate location. Since there were two escape clauses, we have to differentiate as to which one terminated the **while** loop. We do this by using an **if** statement after the loop.

The loop invariant for the **while** is:

There exists no row previous to r that is all zeroes, and of row r, its elements from 1 to $c - 1$ are all zeroes,

$$(\neg(\exists r')(r' < r, X[r'][1..n] = \overline{0}))$$

and $\quad X[r][1..c-1] = \overline{0}.$

The condition that will be true at termination of the **while**, after 0 or more iterations is:

We ran out of rows and there was no row of all zeroes, or, the current row r is all zeroes and all the previous rows had at least one nonzero element each.

$$(r = n + 1 \quad \text{and} \quad (\neg(\exists r')(r' \leq n, X[r'][1..n] = \overline{0})))$$

or $\quad (X[r][1..n] = \overline{0} \quad \text{and} \quad (\neg(\exists r')(r' < r, X[r'][1..n] = \overline{0}))).$

—which is nothing more than a conjunction of the loop invariant with the negation of the **while** loop guard. (This paragraph may be clouding the point).

Now I would like to criticize Rubin's example programs. In the third program in his letter, in which he eliminated the flag, one can tell that the program was written and then hodged-podged into being hopefully correct. This is shown by the "$i := i + 1$;" statement. If a row was all zeroes, then why increment i? Because it is necessary to make the program work.

Thus, all the statements are not fully (correctly) utilized, and an unnecessary loop construct seems to be an unwarranted complication.

In the first program (the "preferred" **GOTO** program) the "**for** $j := 1$ **to** n **do**" behavior is not consistent with the commonly understood definition of the **FOR** loop. A **FOR** loop specifies a definite number of iterations. Depending on the data of row i, the **FOR** j loop may do its body for n iterations, or it may do it for less. The construct used in that program is a quasi-**FOR** definition where it is somewhat like a **FOR** definition except. . . . So you have a **GOTO** which can prematurely break you out of the "**FOR** $j := 1$ **to** n **do**" loop, and a BREAK that can break you prematurely out of the "**for** $i := 1$ **to** n" loop. These two quasi-loops make the program error prone and make proving program correctness harder.

In conclusion, although the derivation of my program may appear contrived, I did derive a similar program in less than five minutes intuitively, except that the guards for the **while** loop were not as good as those in the presented version. Then I thought of how to systematically derive a correct solution from the problem, and thus, the letter.

Incidentally, there are intuitive ways to write non-**GOTO** programs that will run as efficiently as Rubin's **GOTO** program (or better). One involves a different data-structure, which would be an $n + 1$ by $n + 1$ matrix containing sentinels in the extra row and column.

Lee Starr
10 Overlook Terrace
Walden, NY 12586

TO OUR MEMBERS:

More than 15,000 members took advantage of the special multiple-year renewal offer in November and December 1986. As a result of this enthusiastic response, for which we were not fully prepared, processing of normal membership renewals was delayed, and some members who renewed through the special offer received incorrect second notices. If you received such a notice, we wish to assure you that your payments have been applied properly and your publications will arrive on schedule.

In addition, membership cards were not sent with the multiple-year renewal offer because of the nature of that offer. For those of you who responded to the offer, new membership cards are being prepared and will be sent as soon as possible.

We apologize for any inconvenience that these processing problems may have caused you, and urge you to contact the ACM Member Services Department at ACM Headquarters if you have any remaining unresolved problems with your membership.

Don't Sell Technology Short

Alan Borning's treatise, "Computer System Reliability and Nuclear War" (*Communications*, February 1987, pp. 112–31), is a remarkably thorough and perceptive piece of work. Its persuasive exposition of the dangers of over-reliance on computers should not, however, deter us from fully utilizing their great and growing potential for improving weapons command and control systems.

The threat of nuclear war is indeed a problem to be solved "in the political, human realm," but in the computer technology realm we can do something about computer reliability. Every day brings new tools which, if used properly, will help us deal with the problem.

Borning's references to the interaction/integration difficulties associated with projects like the SDI are an important case in point. Relatively new "second generation" tools such as those produced by my firm have vastly improved the reliability of design work in software engineering for large-scale systems, permitting dozens—potentially even hundreds—of engineers to do design work interactively in a highly reliable manner. Within a few months, such PDL tools will be superseded by new CASE (Computer-Aided Software Engineering) tools which will give design engineers access to a single, fully-integrated, monolithic development path. Such tools will cover software development from architectural design through maintenance, and will be especially geared to the requirements of DoD's Ada programming language.

Accuracy, reliability, and elimination of "bugs" before a new system even reaches the testing stage are the objectives being realized in this software design work; work that has obvious beneficial implications for massive interactive projects like the SDI. Improvements of this sort are being made across the board in the computer industry at an increasingly rapid pace.

Yes, computer system reliability has been, and continues to be, a cloud in the SDI sky, but please, let us recognize the very significant progress being made, continuously, in this crucial area.

P.E. Borkovitz
Executive Vice President
Advanced Technology International, Inc.
350 Fifth Ave.
New York, NY 10118

Communications, A Matter of Course?

I was pleased to see the ongoing effort to guide and encourage excellence in computer science education (Alfs Berztiss, "A Mathematically Focused Curriculum for Computer Science," *Communications*, May 1987, pp. 356–65). The purpose of teaching computer science is not to fill students with data but rather to teach them how to think, and the curriculum propounded by Berztiss certainly seems to emphasize this theory.

Although it is difficult to predict accurately what computers will be like and how they will be used a decade or two in the future, one steady trend in computing has been the increase in intercomputer communication (witness the inclusion of the author's CSNET address). This field was ignored in the author's curriculum, however, to what I feel is the detriment of computer science education.

A half or full year elective covering topics in communication would go a long way toward exposing students to the field, and would lay the groundwork for those students who target computer communications as a career. The course(s) could cover telecommunication facilities, the OSI model, network concepts, real-world problems and solutions in communication and ethical issues of global computer communication, and still be general enough in scope not to be outdated by advances in hardware and architectures.

Erik Fenna
CNCP Telecommunications
Toronto, Canada

Ain't Got No Body?

I enjoyed Carolynn Van Dyke's article, "Taking 'Computer Literacy' Literally" in the May issue of *Communications* (pp. 366–74), but I was puzzled by the footnote on the bottom of page 369 which stated, in effect, that computing has no body of great work comparable to literary culture. In fact, the literature of algorithms is quite close to being such a body. Algorithms are short; they are not analogous to novels, but perhaps correspond to short stories or even haiku. Competent programmers must be familiar with this literature just as competent writers must be familiar with their literary culture.

There are differences between algorithms and other literary works. The author of a poem or novel need not publish commentary on that work, but we require that the author of an algorithm publish a considerable volume of commentary with the original publication of the work. With traditional literature, most of the credit goes to the author who expressed the idea, not to the original inventor of the idea being expressed. In contrast, we credit the original inventor of an algorithm long after the expression of that algorithm has been modified into a form the inventor would no longer recognize.

Charles Babbage admonished that, "the man who aspires to fortune or to fame by new discoveries must be content to examine with care the knowledge of his contemporaries, or to exhaust his efforts in inventing

again, what he will most probably find has been better executed before" (Paragraph 327, *On the Economy of Machinery and Manufacturers*, 4th ed., Charles Knight, London, 1835). This applies equally to the authors of algorithms and to the authors of traditional literary works.

Douglas W. Jones
Assistant Professor
Department of Computer Science
The University of Iowa
Iowa City, IA

Mail Call

We would like to correct an unfortunate comment made in Dongarra and Grosse's article on "Distribution of Mathematical Software via Electronic Mail" (*Communications*, May 1987, pp. 403–7), that there are no software distribution services comparable to *netlib*.

There are several comparable automated information retrieval systems which use electronic mail as the transport mechanism. Most of these support retrieval of software (in addition to other retrieval functions). Three of the best known are the CSNET Info Server, the suite of systems operated by the BITNET BITNIC, and NIC Service. The BITNIC services and the CSNET Info Server have been generally accessible to electronic mail users for more than two years.

The CSNET Info Server is a service of the CSNET Coordination and Information Center (CIC). The CIC is administered by the University Corporation for Atmospheric Research and operated by BBN Laboratories Inc. in Cambridge, Mass. Versions of the server run under the 4.3bsd and System V UNIX systems with either the Sendmail or MMDF2 mail systems. Users mail requests to info@sh.cs.net, where a query processor scans the request and sends back the desired information (or a suitable error message). The user interface is patterned after that of the MOSIS chip fabrication system developed at the USC Information Sciences Institute (MOSIS was probably the first major information service to rely on electronic mail to transfer data).

Several automated retrieval systems are provided by the BITNET Network Information Center (BITNIC), which is located at EDUCOM in Princeton, New Jersey. Two of these systems, NICSERVE and DATABASE, offer services very similar to *netlib*. NICSERVE provides access to BITNET-related software and information. DATABASE provides keyword access to a variety of databases.

NIC Service is operated by the DDN Network Information Center (NIC) at SRI International in Menlo Park, Calif. Users mail requests to service@sri-nic.arpa. The subject field of the request contains keywords that are used to locate the desired information, which is then mailed back to the user.

Interested users can get more information about these services by contacting the network centers.

Dan Oberst
BITNIC at EDUCOM
Princeton, NJ

Craig Partridge
CSNET CIC
BBN Laboratories Inc.
10 Moulton St.
Cambridge, MA 02238

Jack Be Nimble, Jack Be ...

This letter is a plea to reinstitute the old, much-beloved, "KWIC Index to Computing Literature." I would tolerate a dues increase just to be able to have a reliable and convenient index containing a citation to (nearly) every journal article, book, thesis, and proceedings paper in the CR categories.

I do not really find much use for *Computing Reviews*. It is okay, but completeness and timeliness are what I really want; not reviews. In fact, I would be happy to trade *Computing Reviews* and two SIG publications for a comprehensive KWIC index periodical.

Indeed, with KWIC indexing of authors and titles, no further indexing or categorization need be done. The big job is typing in all the new entries every month, but a bimonthly or quarterly publication would be sufficiently up-to-date.

What do you say? How about polling the members and get going KWICkly?

Gary D. Knott
Dept. of Computer Science
University of Maryland
College Park, Md. 20742

Last month we announced the major burden of handling GOTO letters has been shifted to Technical Correspondence. However, it seems appropriate that the following letters appear in Forum since they relate to the first and second batch of responses which appeared in the May and June issues of Communications.—Ed.

GOTO, One More Time

The GOTO is back, and not only in the pages of the ACM Forum! The Ollie North of language commands is turning up in myriad "end user" tools intended to produce programs without the involvement of programmers. While computer professionals—such as Frank Rubin—may consciously choose to use the feared GOTO in certain cases, users of spreadsheet, database and other macro languages often do so without information on the other constructs available. In the worst cases, other constructs may not even be available.

The debate over GOTOism is too narrowly defined. A growing percentage of code is being written by users whose entire knowledge of algorithm design is bound to the syntax of their favorite packages. If professional programmers can introduce subtle errors into program code, how much greater is the risk when end users do what comes naturally—and let their code jump all over the place?

The new class of "end users" (do-it-yourself programmers) out there need tools that embody principles of well-structured design. Without such tools, they will only reproduce the expensive maintenance headaches professional programmers are so familiar with. The GOTO will continue to spread unchecked.

David Foster
1130 S. Michigan #1006
Chicago, IL 60605

Among all the comments appearing in the May 1987 Forum on Frank Rubin's "'GOTO Considered Harmful' Considered Harmful" letter (*Communications*, March 1987, pp. 195–6), I am surprised that there was not one citation of Donald Knuth's "Structured Programming with **go to** Statements" (*Computing Surveys*, December 1974, pp. 261–301).* Knuth clearly illustrated how the goals of structured programming—ease of understanding, maintainability, and simplification of validation, among others—often cannot be met without GOTO statements.

David E. Ross

* but see Harrison letter in the Technical Correspondence section, July 1987, pp. 634—Ed.

At risk of being accused of "beating a dead horse," I feel compelled to respond to all of the responses generated by Frank Rubin's "GOTO Considered Harmful etc." It seems that much attention has been devoted to demonstrating individual programming prowess at the expense of the author of the original program, while overlooking the problem—namely the relative merits of the GOTO statement.

I have often found myself in the position of arguing against the use of the GOTO, most notably with students attempting to learn Pascal. In this situation it was clear that they should not be allowed to use the GOTO statement, given their lack of experience in making "mature programming decisions" about the use of control structures. Personally, I use only the limited forms of the GOTO allowed in C, and believe (for "religious" reasons) that others should do the same.

This last statement highlights the fundamental problem at issue in these discussions of "GOTO-less" vs. "GOTO-ful" programming. The arguments are mostly dogmatic, and frequently break down into heated discussions of the fundamental strengths and weaknesses of the preferred languages of the authors. With the current knowledge in software metrics being as it is, I do not believe we are capable of adequately analyzing the problem in a purely *scientific* light—that is, our tools for analysis (no matter how unbiased they may seem) always lack objectivity. In fact, the very nature of the metrics are always slanted either for or against unconditional branch statements simply by the opinions of their authors.

For this reason, I suggest to all (myself included) who argue so religiously on this subject to change the focus of attack, from the programming languages used and/or the pragmatic attitudes acquired through years of experience, to a more useful avenue. Let us instead address the issue of developing the appropriate measures for making an objective judgement as to the merits (or lack of same) of the GOTO.

Frederick J. Bourgeois, III
Computer Scientist & Software Engineer
The Eaton Corporation
31717 LaTienda Dr.
Box 5009, M/S 216
Westlake Village, CA 91360

Steven F. Lott's contribution to the great GOTO debate (May 1987, Forum, pp. 353–354) left me stunned. In his zeal to solve Frank Rubin's sample problem without using GOTOs, Lott produces a solution which deliberately fails to terminate. Apparently Lott thinks this is OK: "I had to make several assumptions," he writes. "... performance in the absence of any all-zero row is not specified—in particular, termination is not required."

The GOTO debate is about program complexity, reliability, and maintainability, is it not? In the real world (I am not talking Turing machines or finite automata), termination is *always* required. Lott's first solution, which does not terminate if there is no all-zero row, is a perfect example of why non-termination is disastrous. Depending on the hardware and software environment it's running in, it may (1) return the wrong answer, (2) crash, or (3) loop forever. The problem is: "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." Lott's solution just keeps incrementing the row number until it finds an all-zero row, without checking for the end of the matrix. Consider:

1. The program continues examining memory after the end of matrix X. It happens to find what it thinks is an all-zero row and terminates, returning the invalid row number.

2. The program does not happen to find a spurious solution, but continues through memory. If the hardware incorporates address checking or memory protection, the program will eventually exceed the limits of its address space and crash with an address exception.

3. In the absence of such hardware, the address references and row counter eventually wrap around and the program never terminates. This is presumably what Lott had in mind, though no one I know would consider it acceptable.

Surely an unterminated loop is a programming error—one which, ironically, is much harder to debug than a wayward GOTO statement, since its behavior depends on obscure side effects of the system hardware and software.

Some things go without saying. A program should always terminate. A program should not crash. A program should not return incorrect results. Must these be part of the specification? Let's not lose sight of the forest for the trees when discussing the merits of the GOTO statement.

Lawrence C. Kuekes
Software Discoveries, Inc.
137 Krawski Dr.
South Windsor, CT 06074

The PL/I excerpt in Conrad Weisert's otherwise thoughtful letter (June, 1987 issue) contains a dangerous assumption.

Unless a language definition and compiler implementation specify otherwise, it is generally incorrect to assert anything at all about the value of a loop index variable after exiting the loop. For example, the Pascal User Manual and Report explicitly states, "The final value of

the control variable is left undefined upon normal exit from the for statement" (p 24). Some instructional languages (such as PL/C) actually prohibit the use of an index variable in this manner.

The fact that the test produces proper results in many programs is an artifact of compiler pragmatics, but nevertheless is not strictly correct.

Norman E. Cohen
Manager, Product Development
McIntosh Computer Systems, Inc.
472 S. Salina St.
Syracuse, NY 13202

Response:
I believe the PLI language definition does indeed specify otherwise. Upon normal loop exit the index variable is available and defined in the natural way. As the example showed, this convention allows considerable economy of expression and poses no threat to structured programming objectives.

Norman Cohen's concern is appropriate for other programming languages that cater more to compiler writers than to user-programmers.

—Conrad Weisert

The huge response to my GOTO letter in the March *Communications* shows that this issue is alive and hot. I will keep my remarks about the first five letters (*Communications*, May 1987, pp. 351–55) brief, as I know there are many more to come.

Moore makes some valid points about programming language features. However, my purpose was not to discuss languages, but to discuss the GOTO. I chose Pascal as a base for comparison because it is so widely known, then devised a sample problem to fit that language. If I had chosen PL/I or C, I would have devised a different sample problem.

Musciano makes the point that a programmer who meets a GOTO must search for the target. In most cases the target is nearby. When labels are outdented from the text, they are spotted instantly. If the label is distant, online the FIND command is used, offline the compiler cross-reference is used. Visually, it is usually easier to spot the target of a GOTO than the target of a CALL.

By contrast, the problem of matching each DO with its END and each IF with its ELSE, if any, is much more difficult and error-prone. There can be only one statement with a given label, but there can be numerous ENDs and ELSEs before you find the matching one. Once you have found the label, it stays found, but one END looks like any other, so you find yourself searching over and over for the same pairs. On a paper listing, you can bracket them in pencil, but on a terminal that is infeasible.

I sympathize with Musciano's last paragraph where he indicates that programs misusing GOTOs can be difficult to modify. However, tangled programs without GOTOs can be equally difficult. Anecdotal evidence won't resolve this issue. A disciplined study is needed.

The letter from Lott consists of three unrelated sections. In the first part he describes eight assumptions that he made when solving the sample problem. He fails to mention whether he believes the programmers I surveyed made similar or opposite assumptions, and whether he considers that good or bad.

In the second portion he presents sample programs that illustrate many of the things I find objectionable about GOTO-less programming: extra flags (az), gratutitous subroutines (allZero), and unneeded recursion (second program). Both of these programs are inefficient and complex. The first is also incorrect when the matrix has no zero row, and likely to produce addressing exceptions.

In the third segment, he repeats the usual claim that programs with GOTOs are harder to modify than those without. I dealt with this issue earlier.

Starr's letter presents a very clever GOTO-less solution to the sample problem. He uses this as evidence that I did not give a fair shake to GOTO-less programming. It was expressly to prevent this charge that I did not simply solve the sample problem myself, but rather surveyed other programmers, using only those judged expert by their peers. The first two sample programs are from the survey. The third program shows that a casual solution using GOTOs can even beat a carefully worked-over solution without GOTOs.

Naturally, I agree with Liebhaber's letter. However, I would like to add one thought. When Dijkstra's letter first appeared, I took it as a joke. When the notion the GOTOs were harmful began to spread, I did not become alarmed. I felt that either the notion would be confirmed by studies, or it would disappear. Instead, it has grown without supporting evidence, and my alarm has grown with it. It is time to put this nonsense firmly behind us, and say that GOTOs, properly used, are a valuable tool that can reduce program complexity and improve programmer productivity.

Let me close with an observation: It is easy to find problems where the best known solution with GOTOs permitted is simpler and/or faster than the best known solution with GOTOs forbidden. The opposite is impossible.

Frank Rubin
The Contest Center
59 DeGarmo Hills Road
Wappingers Falls, NY 12590

I did not react to Frank Rubin's original letter [0], confident that all my potential comments would be made by others. But in the five letters published two months later [1], I found none of them expressed. So, I reluctantly concluded that I had better record my concerns, big and small.

(0) The problem statement refers to an N by N matrix X; Rubin's programs refer to an n by n matrix x. In other contexts this might be considered a minor discrepancy, but I thought that by now professional programmers had learned to be more demanding on themselves and not to belittle the virtue of accuracy.

I shall stick to the capital letters.

(1) Rubin still starts indexing the rows and the columns at 1. I thought that by now professional programmers knew how much more preferable it is to let the natural numbers start at 0. I shall start indexing at 0.

(2) Rubin's third program fails for $N = 0$ (in which case his second program succeeds only by accident—see below. I thought that by now professional programmers would know the stuff the silly bugs are made of.

(3) Rubin's second program fails to detect the first all-zero row if it is the last row of the matrix.

(4) Rubin's third program relies—without stating it explicitly—on the "conditional *and*," for which, if the first operand is *false*, the second operand is allowed to be undefined. The conditional connectives—"*cand*" and "*cor*" for short—are, however, less innocent than they might seem at first sight. For instance, *cor* does not distribute over *cand*: compare.

(A *cand* B) *cor* C

with (A *cor* C) *cand* (B *cor* C);

in the case $\neg A \wedge C$, the second expression requires B to be defined, the first one does not. Because the conditional connectives thus complicate the formal reasoning about programs, they are better avoided.

(5) Rubin's letter effectively conceals that his problem can be solved systematically by a nested application of the same algorithm (sometimes known as "The bounded linear search"). His statement of the problem is: "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." Now, concentrate to begin with on the "if any"; nothing should be printed if all rows differ from the all-zero row; formally, if

$$(\mathbf{A}i{:}\ 0 \le i < N{:}\ \neg(\mathbf{A}j{:}\ 0 \le j < N{:}\ X[i, j] = 0))$$

The theorem of "The bounded linear search" states for any boolean function B on the first N natural numbers ($N \ge 0$)

```
[[ var f: bool; var n: int
; f, n := true, 0 {P}
; do f ∧ n ≠ N → f, n := B(n),
                                n + 1 od
  {P ∧ (¬f ∨ n = N)}
]]
```

with the invariant P given by

$$P{:}\quad 0 \le n \le N \wedge (f \equiv (\mathbf{A}k{:}\ 0 \le k < n{:}\ B(k))) \wedge (\mathbf{A}k{:}\ 0 \le k < n - 1{:}\ B(k)),$$

which states that, upon termination in the case f: all B's are true, and in the case $\neg f$: $n - 1$ is the smallest value for which B is false.

Applying the above theorem twice yields for Rubin's problem:

```
[[var c: bool; var i: int
; c, i := true, 0
; do c ∧ i ≠ N →
    [[var d: bool; var j: int
    ; d, j := true, 0
    ; do d ∧ j ≠ N → d, j := X[i, j]
                              = 0, j + 1 od
    ; c, i := ¬d, i + 1
    ]]
  od
; if c → skip □¬c → print (i − 1) fi
]]
```

and for me this settles the problem.

By my standards, a competent professional programmer in 1987:

(i) should recognize that Rubin's problem asks to be solved by a nested application of the same algorithm;
(ii) should know the theorem of "The bounded linear search";
(iii) should be able to derive that theorem and its proof;
(iv) should not hesitate to use it;
(v) should not waste his time in pointing out that the boolean variable d is superfluous;
(vi) should keep his repetitions simple and disentangled
(vii) etc. . .

Evidently, my priorities are not shared by everyone, for Rubin's letter and most of the five reactions it evoked were conducted instead in terms of all sorts of "programming language features" that seem better ignored than exploited. The whole correspondence was carried out at a level that vividly reminded me of the intellectual climate of 20 years ago, as if stagnation were the major characteristic of the computing profession, and that was a disappointment.

REFERENCES

0. Rubin, Frank. ""GOTO Considered Harmful" Considered Harmful." *Commun. ACM 30*, 3 (Mar. 1987), 195–196.
1. Moore, Donald and Musciano, Chuck and Liebhaber, Michael J. and Lott, Steven F. and Starr, Lee. " " "GOTO Considered Harmful" Considered Harmful" Considered Harmful?" *Commun. ACM 30*, 5 (May 1987), 351–355.

Prof. Dr. Edsger W. Dijkstra
Department of Computer Sciences
The University of Texas
Austin, TX 78712-1188

After Nineteen Years

I resigned from the ACM Council effective the 13th of October.

I am the last active charter member of the Association. From 1947 to 1961 I served on a dozen ACM committees, usually in connection with the joint computer conference sponsorship. After I returned from my first European sojourn I was elected a Council member, in Spring of 1968. I have served as member-at-large, vice president, president, past president, and for almost eight years as member-at-large again.

In the 40 years since its founding, and in over 19 years on its Council, I have had a dream of what ACM should be. I wanted it to attract every serious computer person in America and many others across the world—and by example define the word "professional." I wanted its governance to be open and democratic. I wanted its members to be mutually supportive: a band of brothers and sisters.

I wanted ACM to be concerned with the impact of computers and computing on society, and to interact with our industry worldwide and with national, regional and local governments. For two decades I have wanted us to withdraw from AFIPS and work toward a merger with the IEEE Computer Society.

In the words of the old vaudeville joke, "You just can't get there from here!". The computer science establishment stands grimly across the path, and Council is its tool. Currently, services to the membership (other than publications, of course) are being cut back on the grounds of economic emergency, while our bank balance climbs past $9 million, and the final figures for FY '87 show a million-dollar surplus versus a budgeted loss. Meanwhile the lectureship program, its funding cancelled, struggles along on charity from the SIGs!

I've tried, dammit! I've endured the Establishment pinpricks: 11 years of expense account cheapness, 20 years of agenda trickery, unanswered correspondence. I've struggled against censorship and election nastiness. I've sat at the Council table and seen my motion to reconsider the cut of 63 percent in FY '88 chapters support fail *for lack of a second.* I've watched the Member-Officer Forum, Headquarters and regional newsletters, and other channels to and from the membership close down.

Some of it really hurt. My wife Nancy was thrown off the SIG Board as soon as I was safely out of the presidency. I was excluded from participation in the recent ACM conference on the history of scientific and numerical computation. I was walled off from the history panel of the sad, little 40th Anniversary celebration.

Enough is enough.

Now, how about those of you out there who still want to salvage the ACM enterprise? It is *not* impossible, but it cannot be done by genteel complaints. You will have to capture Council.

An anti-Establishment caucus must put together a reform slate of three officers (president, vice president and secretary), three members-at-large, and four regional representatives. They must be nominated by petition, and all 10 must state in their campaign material that the slate is running as a unit.

If elected in 1988, they would find three supporters already seated and some sympathy available from two others—naming the five in this letter would be counterproductive. That means reform power; there are 24 votes at the table, with the president voting only to break ties.

The president appoints most board and committee chairmen. But, only if he or she has a council majority can reform presidential appointments be meaningful, and Council elects several key people directly. The restrictive budget can be overridden or reformed, but only by Council authority.

Headquarters is neutral—perhaps even secretly sympathetic—and it is capable. Notably, the publications staff can keep *Communications of the ACM* and such afloat if the Establishment editors rebel, and can help salvage Computing Surveys from the academics.

Council should restore its third meeting, and hold it at SIGGRAPH to see what ACM people can do when they get free of the Establishment. A hundred other changes could be made—but only if members want it to happen and support a genuine reform movement.

The Association for Computing Machinery will survive, but unless it reforms itself, it will dwindle to a self-serving bunch of computer scientists—an establishment for the Establishment. And no one in the whole great world of computing will care.

Herb Grosch
Mies, Switzerland

Danger Signals

In the August Forum, p. 658, P.E. Borkovitz predicts, "Within a few months, . . . PDL tools will be superseded by new CASE (Computer-Aided Software Engineering) tools which will give design engineers access to a single, fully-integrated, monolithic development path." Literate readers will notice the warning: a boulder is soon to be rolled athwart the way where Dijkstra would have us strew pearls.

M.D. McIlroy
AT&T Bell Labs
600 Mountain Ave.
Murray Hill, N.J. 07974

The World's Shortest Mutual Exclusion Algorithm

This algorithm could have been published as a paper 20 years ago, but today it looks crazy and can be implemented only on a few antique processors. Researchers who communicated to me privately expressed their interest in this algorithm—perhaps because it is the *shortest* mutual exclusion algorithm, shorter than what they knew of. They also indicated that it should find a place in the published literature, better late than never.

I cannot think of a better forum than the Forum to publish it.

1. Preamble

This algorithm for mutual exclusion should have been born at least 20 years ago for machines like EDVAC or its immediate successors branded as antique by the present standard. This algorithm is unlikely to be used in machines designed after 1970. But, this is the world's shortest mutual exclusion algorithm—shorter than any I have read about. It uses self-modifying instructions to implement spin-lock on a multiprogrammed uniprocessor. The algorithm is as follows:

2. The Solution

Let *i* be a machine instruction. The symbol #(*i*) would be used to represent the bit pattern corresponding to the machine code for that instruction.

The following codes would be executed by each process contending for the critical section:

trick: *trick* := #(*go to trick*);
critical section;
trick := #(*trick* := #(*go to trick*));

The execution of an instruction in a uniprocessor is an atomic action. The first instruction, while executed by one process, changes itself to *go to trick* and this blocks all the other contending processes by forcing them to execute a self-loop. Eventually, when the first process completes the critical section, it restores the first instruction to its original form. As a result, another process can enter the critical section.

The algorithm is correct in the sense that it is free from deadlock and at most one process can be in the critical section. Also, it does not unduly delay a process from allowing entry to the critical section provided it is free. The fairness is left to the process scheduler.

This algorithm does not work on multiprocessors or on machines which do not permit instruction modification.

Sukumar Ghosh
Department of Computer Science
The University of Iowa
Iowa City, IA 52242

Last (Gasp!) GOTO

Several new responses to my GOTO letter have appeared in the June and August issues of *Communications.** Michael Harrison's July letter suggests that GOTOs be introduced by transforming GOTO-less programs. This seems the wrong way around. Why start with a complex program, then simplify it, when it is easier to start simple?

Frederick Bourgeois's August reply proposes forbidding the use of GOTOs by inexperienced students. The problem here is you only gain experience through use. Instead, I suggest that students be given progressive assignments, where they write an original program using any style and language features they wish. That program can then be modified to add new features, change output formats, etc . . . In this way, students discover first hand what practices make programs difficult to modify.

I will devote the rest of this letter to the response from Edsger Dijkstra in the August issue. Let me first answer his numbered criticisms, then remark on his proposed solution to the matrix problem.

(0) The upper/lower case argument is nonsense. Just as I use different styles in my puzzle magazine and in *Communications*, so I use different styles for problem specifications and programs. Each style is suited to the audience and the material.

(1) Personally, I use both 0- and 1-origin indexing. 1-origin is usually simpler, e.g. FOR $I := 1$ TO N is cleaner than FOR $I := 0$ TO $N - 1$. I use 0-origin if it results in simpler subscript expressions, e.g. $X[I + J]$ instead of $X[I + J - 1]$. Use determines form.

(2) I have never encountered a 1×1 matrix in practice. If I were writing a general purpose subroutine for widespread use, I would either handle that case, or document the restriction $N > 1$.

(3) The final test should be IF ALLZERO instead of IF $I <= N$. The fact that so many people responded to my letter, but did not see this error, supports my view that GOTO-less programming is not inherently clearer or easier to debug.

(4) My example need not depend on any sort of conditional Boolean connective. The expression can be fully evaluated. This will not cause a problem unless the programmer has turned on index range checking.

(5) All of the programmers I surveyed used nested search loops to solve the matrix problem. To make comparisons easier, I have translated Dijkstra's sample solution into Pascal.

```
c := true;
i := 0;
while c and (i < > n)
  do begin
     d := true;
     j := 0;
     while d and (j < > n)
       do begin
       d := x[i,j] = 0;
       j := j + 1;
     end;
     c := not d;
     i := i + 1;
end;
if c
     then skip
     else print (i - 1);
```

The survey programmers produced both 1- and 2-flag versions. Two programmers first wrote 2-flag programs, then eliminated the unnecessary flag. I used the simpler 1-flag

(*continued on p. 1085*)

* July 1987 Technical Correspondence, pp. 632–4; August Forum, pp. 659–62.

Language for Automation: Symbiotic and Intelligent Robotics, University of Maryland, College Park, MD, August 29–31. Submit 4c. of complete papers (20 pages maximum) to Prof. P.A. Ligomenides, Electrical Engineering Dept., University of Maryland, College Park, MD 20742.

March 1

ICDT 1988: International Conference on Database Theory, Bruges, Belgium. August 31–September 2. Submit 6c. of full paper to Dirk Van Gucht, Computer Science Dept., Indiana University, Bloomington, IN 47405.

March 15

CONCURRENCY 88, Hamburg, Fed. Rep. of Germany, October 18–19. Submit 5c. of paper (25 pages maximum) to Friedrich Vogt, Fachbereich Informatik, Universitat Hamburg, Bodenstedtstr. 16, D-2000 Hamburg 50, FRG, Tel: +49 40 4123-6060/61.

March 15

OOPSLA 88, San Diego, Calif., September 25–29. Submit 5c. of 25 double-spaced pages (approx. 4500 words), a cover sheet including authors' names, addresses and telephone numbers, and 100 word abstract to Kurt Schmucker, OOPSLA 88, Productivity Products International, Rocky Glen Mill, 75 Glen Rd., Sandy Hook, CT 06482; (203) 426-1875.

March 15

8th International Conference in Computer Science, Santiago de Chile, July 4–8. Submit 10c. of extended abstract (10 page maximum) in English, Spanish or Portuguese to Alberto Mendelzon, CSRI, University of Toronto, 10 King's College Rd., Toronto, Canada M5S 1A4

March 30

9th Annual International Conference on Information Systems, Minneapolis, Minn. November 30–December 3. Sponsors: TIMS, SIMS in coop. with ACM. 4c. of papers (maximum 25 pages) to Margaret H. Olson, Program Chair, Graduate School of Business Administration, New York University, 90 Trinity Place, New York, NY 10003.

May 10

3rd International Conference on Fifth Generation Computer Systems, Tokyo, Japan. November 28–December 2. 6c. of 5000-word manuscripts and 250-word abstracts to Hidehiko Tanaka, FGCS '88, Program Chair, ICOT, 28 Mita Kokusai Bldg., 1-4-28 Mita, Mianto-ku, Tokyo 108 Japan.

May 15

3rd International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness, Jerusalem, Israel. June 28–30. Organized by Information Processing Association of Israel in cooperation with ACM SIGMOD. For information concerning submission, contact Katriel Beeri, Data Base 88, P.O. Box 29313, Tel-Aviv 65121, Israel, 02-585266.

May 16

8th Conference on Foundations of Software Technology and Theoretical Computer Science, Pune, India. December 14–16. Sponsor: Tata Institute of Fundamental Research Developement & Design Centre. Submit 4c. of full paper (maximum 5000 words) to K.V. Nori, TRDDC, 1 Mangaldas Rd., Pune, India; Tel: (212)-61608, Telex: 0145-464.

April 1

Directions and Implications of Advanced Computing, St. Paul, Minn., August 21. Submit 4c. of complete papers including abstract (6000 words maximum) to Nancy Levenson, ICS Department, University of California, Irvine, CA 92717.

ACM Forum (*continued from p. 997*)

version in my letter to be fair to the GOTO-less advocates.

Dijkstra's solution is similar to the 2-flag versions, with minor differences. All of the survey programmers used an IF to test for zero elements. Dijkstra's $D := X[I, J] = 0$ is syntactically simpler, but obscure. I usually avoid that construct.

Dijkstra uses $I \langle \rangle N$ as a loop terminating condition. This is often unsafe, since statements inside a loop might cause I to skip the value N. It is better to use $I < N$.

I presume that the SKIP in the final IF is some form of null statement. Thus, it serves as a type of disguised GOTO. The 2-flag programs in my survey avoided this awkwardness by reversing the sense of the outer flag, e.g. IF ZERO-FOUND THEN PRINT.

Such differences aside, I do not see how Dijkstra's sample program shows that GOTO-less programming is any simpler or more readable. Rather, the notational quirks of his example, like starting lines with semicolons and interleaving assignment statements, seem most inscrutable. I was amused to note that 11 of the 13 programmers in my survey produced better solutions than his.

Overall, I was very disappointed in Dijkstra's reply, which seemed to be a scattershot attack on everyone else who wrote.

Frank Rubin
The Contest Center
59 DeGarmo Hills Road
Wappingers Fall, NY 12590

With this second rejoinder from Frank Rubin it seems expedient to bring to a close the publication of correspondence generated by his March 1987 Forum letter, greater response by far than with any other issue ever considered in these pages—Ed.