Structured Programming with go to Statements

DONALD E. KNUTH

Stanford University, Stanford, California 94305

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without **go to** statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not **go to** statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

Keywords and phrases: structured programming, **go to** statements, language design, event indicators, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems searching, Quicksort, efficiency

CR categories: 4.0, 4.10, 4.20, 5.20, 5.5, 6.1 (5.23, 5.24, 5.25, 5.27)

You may go when you will go,
And I will stay behind.
—*Edna St. Vincent Millay* [66]

Most likely you go your way and I'll go mine.
—*Song title by Bob Dylan* [33]

Do you suffer from painful elimination?
—*Advertisement, J. B. Williams Co.*

INTRODUCTION

A revolution is taking place in the way we write programs and teach programming, because we are beginning to understand the associated mental processes more deeply. It is impossible to read the recent book *Structured programming* [17; 55] without having it change your life. The reasons for this revolution and its future prospects have been aptly described by E. W. Dijkstra in his 1972 Turing Award Lecture, "The Humble Programmer" [27].

As we experience this revolution, each of us naturally is developing strong feelings one way or the other, as we agree or disagree with the revolutionary leaders. I must admit to being a non-humble programmer, egotisti-

This research was supported in part by the National Science Foundation under grant number GJ 36473X, and by IBM Corporation.

CONTENTS

cal enough to believe that my own opinions of the current trends are not a waste of the reader's time. Therefore I want to express in this article several of the things that struck me most forcefully as I have been thinking about structured programming during the last year; several of my blind spots were removed as I was learning these things, and I hope I can convey some of my excitement to the reader. Hardly any of the ideas I will discuss are my own; they are nearly all the work of others, but perhaps I may be presenting them in a new light. I write this article in the first person to emphasize the fact that what I'm saying is just one man's opinion; I don't expect to persuade everyone that my present views are correct.

Before beginning a more technical discussion. I should confess that the title of this article was chosen primarily to generate attention. There are doubtless some readers who are convinced that abolition of **go to** statements is merely a fad, and they may see this title and think, "Aha! Knuth is rehabilitating the **go to** statement, and we can go back to our old ways of programming again." Another class of readers will see the heretical title and think, "When are diehards like Knuth going to get with it?" I hope that both classes of people will read on and discover that what I am really doing is striving for a reasonably well balanced viewpoint about the proper role of **go to** statements. I argue for the elimination of **go to**'s in certain cases, and for their introduction in others.

I believe that by presenting such a view I am not in fact disagreeing sharply with Dijkstra's ideas, since he recently wrote the following: "Please don't fall into the trap of believing that I am terribly dogmatical about [the **go to** statement]. I have the uncomfortable feeling that others are making a religion out of it, as if the conceptual problems of programming could be solved by a single trick, by a simple form of coding discipline!" [29]. In other words, it seems that fanatical advocates of the New Programming are going overboard in their strict enforcement of morality and purity in programs. Sooner or later people are going to find that their beautifully-structured programs are running at only half the speed —or worse—of the dirty old programs they used to write, and they will mistakenly blame the structure instead of recognizing what is probably the real culprit—the system overhead caused by typical compiler implementation of Boolean variables and procedure calls. Then we'll have an unfortunate counterrevolution, something like the current rejection of the "New Mathematics" in reaction to its over-zealous reforms.

It may be helpful to consider a further analogy with mathematics. In 1904, Bertrand Russell published his famous paradox about the set of all sets which aren't members of themselves. This antinomy shook the foundations of classical mathematical reasoning, since it apparently brought very simple and ordinary deductive methods into question. The ensuing crisis led to the rise of "intuitionist logic", a school of thought championed especially by the Dutch mathematician, L. E. J. Brouwer; intuitionism abandoned all deductions that were based on questionable nonconstructive ideas. For a while it appeared that intuitionist logic would cause a revolution in mathematics. But the new approach angered David Hilbert, who was perhaps the leading mathematician of the time; Hilbert said that "Forbidding a mathematician to make use of the principle of the excluded middle is like forbidding an astronomer his telescope or a boxer the use of his fists." He characterized the intuitionist approach as seeking "to save mathematics by throwing overboard all that is troublesome. . . . They would chop up and mangle the science. If we would follow such a reform as they suggest, we could run the risk of losing a great part of our most valuable treasures" [80, pp. 98–99, 148–150, 154–157, 184–185, 268–270].

Something a little like this is happening in computer science. In the late 1960's we witnessed a "software crisis", which many people thought was paradoxical because programming was supposed to be so easy. As a result of the crisis, people are now beginning to renounce every feature of programming that can be considered guilty by virtue of its association with difficulties. Not only **go to** statements are being questioned; we also hear complaints about floating-point calculations, global variables, semaphores, pointer variables, and even assignment statements. Soon we might be restricted to only a dozen or so programs that are sufficiently simple to be allowable; then we will be almost certain that these programs cannot lead us into any trouble, but of course we won't be able to solve many problems.

In the mathematical case, we know what happened: The intuitionists taught the other mathematicians a great deal about deductive methods, while the other mathematicians cleaned up the classical methods and eventually "won" the battle. And a revolution did, in fact, take place. In the computer science case, I imagine that a similar thing will eventually happen: purists will point the way to clean constructions, and others will find ways to purify their use of floating-point arithmetic, pointer variables, assignments, etc., so that these classical tools can be used with comparative safety.

Of course all analogies break down, including this one, especially since I'm not yet conceited enough to compare myself to David Hilbert. But I think it's an amusing coincidence that the present programming revolution is being led by another Dutchman (although he doesn't have extremist views corresponding to Brouwer's); and I do consider assignment statements and pointer variables to be among computer science's "most valuable treasures".

At the present time I think we are on the verge of discovering at last what programming languages should really be like. I look forward to seeing many responsible experiments with language design during the next few years; and my dream is that by 1984 we will see a consensus developing for a really good programming language (or, more likely, a coherent family of languages). Furthermore, I'm guessing that people will become so disenchanted with the languages they are now using—even COBOL and FORTRAN—that this new language, UTOPIA 84, will have a chance to take over. At present we are far from that goal, yet there are indications that such a language is very slowly taking shape.

Will Utopia 84, or perhaps we should call it Newspeak, contain **go to** statements? At the moment, unfortunately, there isn't even a consensus about this apparently trivial issue, and we had better not be hung up on the question too much longer since there are only ten years left.

I will try in what follows to give a reasonably comprehensive survey of the **go to** controversy, arguing both pro and con, without taking a strong stand one way or the other until the discussion is nearly complete. In order to illustrate different uses of **go to** statements, I will discuss many example programs, some of which tend to negate the conclusions we might draw from the others. There are two reasons why I have chosen to present the material in this apparently vacillating manner. First, since I have the opportunity to choose all the examples, I don't think it's fair to load the dice by selecting only program fragments which favor one side of the argument. Second, and perhaps most important, I tried this approach when I lectured on the subject at UCLA in February, 1974, and it worked beautifully: nearly everybody in the audience had the illusion that I was largely supporting his or her views, regardless of what those views were!

1. ELIMINATION OF go to STATEMENTS

Historical Background

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. Here is the history of the subject, as far as I have been able to trace it.

The first programmer who systematically began to avoid all labels and **go to** statements was perhaps D. V. Schorre, then of UCLA. He has written the following account of his early experiences [85]:

> Since the summer of 1960, I have been writing programs in outline form, using conventions of indentation to indicate the flow of control. I have never found it necessary to take exception to these conventions by using *go statements*. I used to keep these outlines as original documentation of a program, instead of using flow charts . . . Then I would code the program in assembly language from the outline. Everyone liked these outlines better than the flow charts I had drawn before, which were not very neat—my flow charts had been nick-named "balloon-o-grams".

He reported that this method made programs easier to plan, to modify and to check out.

When I met Schorre in 1963, he told me of his radical ideas, and I didn't believe they would work. In fact, I suspected that it was really his rationalization for not finding an easy way to put labels and **go to** statements into his Meta-II subset of Algol [84], a language which I liked very much except for this omission. In 1964 I challenged him to write a program for the eight-queens problem without using **go to** statements, and he responded with a program using recursive procedures and Boolean variables, very much like the program later published independently by Wirth [96].

I was still not convinced that all **go to** statements could or should be done away with, although I fully subscribed to Peter Naur's observations which had appeared about the same time [73]. Since Naur's comments were the first published remarks about harmful **go to**'s, it is instructive to quote some of them here:

> If you look carefully you will find that surprisingly often a **go to** statement which looks back really is a concealed **for** statement. And you will be pleased to find how the clarity of the algorithm improves when you insert the **for** clause where it belongs. . . . If the purpose [of a programming course] is to teach Algol programming, the use of flow diagrams will do more harm than good, in my opinion.

The next year we find George Forsythe also purging **go to** statements from algorithms submitted to *Communications of the ACM* (cf. [53]). Incidentally. the second example program at the end of the original Algol 60 report [72] contains four **go to** statements, to labels named AA, BB, CC, and DD, so it is clear that the advantages of Algol's control structures weren't fully perceived in 1960.

In 1965, Edsger Dijkstra published the following instructive remarks [21]:

> Two programming department managers from

different countries and different backgrounds —the one mainly scientific, the other mainly commercial—have communicated to me, independently of each other and on their own initiative, their observation that the quality of their programmers was inversely proportional to the density of goto statements in their programs. . . . I have done various programming experiments . . . in modified versions of ALGOL 60 in which the goto statement was abolished. . . . The latter versions were more difficult to make: we are so familiar with the jump order that it requires some effort to forget it! In all cases tried, however, the program without the goto statement turned out to be shorter and more lucid.

A few months later, at the ACM Programming Languages and Pragmatics Conference, Peter Landin put it this way [59]:

There is a game sometimes played with ALGOL 60 programs—rewriting them so as to avoid using **go to** statements. It is part of a more embracing game—reducing the extent to which the program conveys its information by explicit sequencing. . . . The game's significance lies in that it frequently produces a more "transparent" program—easier to understand, debug, modify, and incorporate into a larger program.

Peter Naur reinforced this opinion at the same meeting [74, p. 179].

The next chapter in the story is what many people regard as the first, because it made the most waves. Dijkstra submitted a short article to *Communications of the ACM*, devoted entirely to a discussion of **go to** statements. In order to speed publication, the editor decided to publish Dijkstra's article as a letter, and to supply a new title, "Go to statement considered harmful". This note [23] rapidly became well-known; it expressed Dijkstra's conviction that **go to**'s "should be abolished from all 'higher level' programming languages (i.e., everything except, perhaps, plain machine code). . . . 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." He encouraged looking for alternative constructions which may be necessary to satisfy all needs. Dijkstra also recalled that Heinz Zemanek had expressed doubts about **go to** statements as early as 1959; and that Peter Landin, Christopher Strachey, C. A. R. Hoare and others had been of some influence on his thinking.

By 1967, the entire XPL compiler had been written by McKeeman, Horning, and Wortman, using **go to** only once ([65], pp. 365–458; the **go to** is on page 385). In 1971, Christopher Strachey [87] reported that "It is my aim to write programs with no labels. I am doing quite well. I have got the operating system down to 5 labels and I am planning to write a compiler with no labels at all." In 1972, an entire session of the ACM National Conference was devoted to the subject [44; 60; 100]. The December, 1973, issue of *Datamation* featured five articles about structured programming and elimination of **go to**'s [3; 13; 32; 64; 67]. Thus, it is clear that sentiments against **go to** statements have been building up. In fact, the discussion has apparently caused some people to feel threatened; Dijkstra once told me that he actually received "a torrent of abusive letters" after publication of his article.

The tide of opinion first hit me personally in 1969, when I was teaching an introductory programming course for the first time. I remember feeling frustrated on several occasions, at not seeing how to write programs in the new style; I would run to Bob Floyd's office asking for help, and he usually showed me what to do. This was the genesis of our article [52] in which we presented two types of programs which did not submit gracefully to the new prohibition. We found that there was no way to implement certain simple constructions with **while** and conditional statements substituted for **go to**'s, unless extra computation was specified.

During the last few years several languages have appeared in which the designers proudly announced that they have abolished the **go to** statement. Perhaps the most prominent of these is BLISS [98], which originally replaced **go to**'s by eight so-called "escape" statements. And the eight weren't even enough; the authors wrote, "Our mistake was in assuming that there is no need for a label once the **go to** is removed," and they later [99, 100] added a new statement "**leave** ⟨label⟩ **with** ⟨expression⟩" which goes to the place *after* the statement identified by the ⟨label⟩. Other **go to**-less languages for systems programming have similarly introduced other statements which provide "equally powerful" alternative ways to jump.

In other words, it seems that there is widespread agreement that **go to** statements are harmful, yet programmers and language designers still feel the need for some euphemism that "goes to" without saying **go to.**

A Searching Example

What are the reasons for this? In [52], Floyd and I gave the following example of a typical program for which the ordinary capabilities of **while** and **if** statements are inadequate. Let's suppose that we want to search a table A[1] $\cdots$ A[m] of distinct values, in order to find where a given value x appears; if x is not present in the table, we want to insert it as an additional entry. Let's suppose further that there is another array B, where B[i] equals the number of times we have searched for the value A[i]. We might solve such a problem as follows:

Example 1:

```
for i := 1 step 1 until m do
  if A[i] = x then go to found fi;
not found: i := m+1; m := i;
  A[i] := x; B[i] := 0;
found: B[i] := B[i]+1;
```

(In the present article I shall use an ad hoc programming language that is very similar to ALGOL 60, with one exception: the symbol **fi** is required as a closing bracket for all **if** statements, so that **begin** and **end** aren't needed between **then** and **else.** I don't really like the looks of **fi** at the moment; but it is short, performs a useful function, and connotes finality, so I'm confidently hoping that I'll get used to it. Alan Perlis has remarked that **fi** is a perfect example of a cryptic notation that can make programming unnecessarily complicated for beginners; yet I'm more comfortable with **fi** every time I write it. I still balk at spelling *other* basic symbols backwards, and so do most of the people I know; a student's paper containing the code fragment "**esac; comment** bletch **tnemmoc;**" is a typical reaction to this trend!)

There are ways to express Example 1 without **go to** statements, but they require more computation and aren't really more perspicuous. Therefore, this example has been widely quoted in defense of the **go to** statement, and it is appropriate to scrutinize the problem carefully.

Let's suppose that we've been forbidden to use **go to** statements, and that we want to do *precisely* the computation specified in Example 1 (using the obvious expansion of such a **for** statement into assignments and a **while** iteration). If this means not only that we want the same results, but also that we want to do the same operations in the same order, the mission is impossible. But if we are allowed to weaken the conditions just slightly, so that a relation can be tested twice in succession (assuming that it will yield the same result each time, i.e., that it has no side-effects), we can solve the problem as follows:

Example 1a:

```
i := 1;
while i ≤ m and A[i] ≠ x do i := i+1;
if i > m then m := i; A[i] := x; B[i] := 0 fi;
B[i] := B[i]+1;
```

The **and** operation used here stands for McCarthy's sequential conjunction operator [62, p. 185]; i.e., "p **and** q" means "**if** p **then** q **else false fi**", so that q is not evaluated when p is false. Example 1a will do exactly the same sequence of computations as Example 1, except for one extra comparison of i with m (and occasionally one less computation of $m+1$). If the iteration in this **while** loop is performed a large number of times, the extra comparison has a negligible effect on the running time.

Thus, we can live without the **go to** in Example 1. But Example 1a is slightly less readable, in my opinion, as well as slightly slower; so it isn't clear what we have gained. Furthermore, if we had made Example 1 more complicated, the trick of going to Example 1a would no longer work. For example, suppose we had inserted another statement into the **for** loop, just before the **if** clause; then the relations $i \leq m$ and A[i] $= x$ wouldn't have been tested consecutively, and we couldn't in general have combined them with **and.**

John Cocke told me an instructive story

relating to Example 1 and to the design of languages. Some PL/I programmers were asked to do the stated search problem without using jumps, and they came up with essentially the following two solutions:

```
a)     DO I = 1 to M WHILE A(I) ¬ = X;
       END;
       IF I > M THEN
            DO; M = I; A(I) = X; B(I) = 0; END;
       B(I) = B(I) + 1;
b)     FOUND = 0;
       DO I = 1 TO M WHILE FOUND = 0;
              IF A(I) = X THEN FOUND = 1;
       END;
       IF FOUND = 0 THEN
              DO; M = I; A(I) = X; B(I) = 0; END;
       B(I) = B(I) = 1;
```

Solution (a) is best, but since it involves a null iteration (with no explicit statements being iterated) most people came up with Solution (b). The instructive point is that Solution (b) doesn't work; there is a serious bug which caused great puzzlement before the reason was found. Can the reader spot the difficulty? (The answer appears on page 298.)

As I've said, Example 1 has often been used to defend the **go to** statement. Unfortunately, however, the example is totally unconvincing in spite of the arguments I've stated so far, because the method in Example 1 is almost *never* a good way to search an array for x! The following modification to the data structure makes the algorithm much better:

Example 2:

```
A[$m$+1] := $x$; $i$ := 1;
while A[$i$] $\neq$ $x$ do $i$ := $i$+1;
if $i > m$ then $m$ := $i$; B[$i$] := 1;
else B[$i$] := B[$i$]+1 fi;
```

Example 2 beats Example 1 because it makes the inner loop considerably faster. If we assume that the programs have been handcoded in assembly language, so that the values of i, m, and x are kept in registers, and if we let n be the final value of i at the end of the program, Example 1 will make $6n + 10$ ($+3$ if not found) references to memory for data and instructions on a typical computer, while the second program will make only $4n + 14$ ($+6$ if not found). If, on the other hand, we assume that these programs are translated by a typical "90% efficient compiler" with bounds-checking suppressed, the corresponding run-time figures are respectively about $14n + 5$ and $11n + 21$. (The appendix to this paper explains the ground rules for these calculations.) Under the first assumption we save about 33% of the run-time, and under the second assumption we save about 21%, so in both cases the elimination of the **go to** has also eliminated some of the running time.

Efficiency

The ratio of running times (about 6 to 4 in the first case when n is large) is rather surprising to people who haven't studied program behavior carefully. Example 2 doesn't look *that* much more efficient, but it is. Experience has shown (see [46], [51]) that most of the running time in non-IO-bound programs is concentrated in about 3% of the source text. We often see a short inner loop whose speed governs the overall program speed to a remarkable degree; speeding up the inner loop by 10% speeds up everything by almost 10%. And if the inner loop has 10 instructions, a moment's thought will usually cut it to 9 or fewer.

My own programming style has of course changed during the last decade, according to the trends of the times (e.g., I'm not quite so tricky anymore, and I use fewer **go to**'s), but the major change in my style has been due to this inner loop phenomenon. I now look with an extremely jaundiced eye at every operation in a critical inner loop, seeking to modify my program and data structure (as in the change from Example 1 to Example 2) so that some of the operations can be eliminated. The reasons for this approach are that: a) it doesn't take long, since the inner loop is short; b) the payoff is real; and c) I can then afford to be less efficient in the other parts of my programs, which therefore are more readable and more easily written and debugged. Tools are being developed to make this critical-loop identification job easy (see for example [46] and [82]).

Thus, if I hadn't seen how to remove one of the operations from the loop in Example 1 by changing to Example 2. I would probably (at least) have made the **for** loop run from m to 1 instead of from 1 to m, since it's usually easier to test for zero than to compare with m. And if Example 2 were really critical, I would improve on it still more by "doubling it up" so that the machine code would be essentially as follows.

Example 2a:

```
        A[m+1] := x; i := 1; go to test;
        loop:  i := i+2;
        test:  if A[i] = x then go to found fi;
               if A[i+1] ≠ x then go to loop fi;
               i := i+1;
        found: if i > m then m := i; B[i] := 1;
               else B[i] := B[i]+1 fi;
```

Here the loop variable i increases by 2 on each iteration, so we need to do that operation only half as often as before; the rest of the code in the loop has essentially been duplicated to make this work. The running time has now been reduced to about $3.5n + 14.5$ or $8.5n + 23.5$ under our respective assumptions—again this is a noticeable saving in the overall running speed, if, say, the average value of n is about 20, and if this search routine is performed a million or so times in the overall program. Such loop-optimizations are not difficult to learn and, as I have said, they are appropriate in just a small part of a program, yet they very often yield substantial savings. (Of course if we want to improve on Example 2a still more, especially for large m, we'll use a more sophisticated search technique; but let's ignore that issue, at the moment, since I want to illustrate loop optimization in general, not searching in particular.)

The improvement in speed from Example 2 to Example 2a is only about 12%, and many people would pronounce that insignificant. The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being practiced by penny-wise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs. In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal; and I believe the same viewpoint should prevail in software engineering. Of course I wouldn't bother making such optimizations on a one-shot job, but when it's a question of preparing quality programs, I don't want to restrict myself to tools that deny me such efficiencies.

There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only *after* that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail. After working with such tools for seven years, I've become convinced that all compilers written from now on should be designed to provide all programmers with feedback indicating what parts of their programs are costing the most; indeed, this feedback should be supplied automatically unless it has been specifically turned off.

After a programmer knows which parts of his routines are really important, a transformation like doubling up of loops will be worthwhile. Note that this transformation introduces **go to** statements—and so do several other loop optimizations; I will return to this point later. Meanwhile I have to admit that the presence of **go to** statements in Example 2a has a negative as well as a positive effect on efficiency; a non-optimizing compiler will tend to produce awkward code, since the contents of registers can't be assumed known when a label is passed. When I computed the running times cited above by looking at a typical compiler's output for this example, I found that the improvement in performance was not quite as much as I had expected.

Error Exits

For simplicity I have avoided a very important issue in the previous examples, but it must now be faced. All of the programs we have considered exhibit bad programming practice, since they fail to make the necessary check that m has not gone out of range. In each case before we perform "$m := i$" we should precede that operation by a test such as

```
    if m = max then go to memory overflow;
```

where *max* is an appropriate threshold value. I left this statement out of the examples since it would have been distracting, but we need to look at it now since it is another important class of **go to** statements: an *error exit*. Such checks on the validity of data are very important, especially in software, and it seems to be the one class of **go to**'s that still is considered ugly but necessary by today's leading reformers. (I wonder how Val Schorre has managed to avoid such **go to**'s during all these years.)

Sometimes it is necessary to exit from several levels of control, cutting across code that may even have been written by other programmers; and the most graceful way to do this is a direct approach with a **go to** or its equivalent. Then the intermediate levels of the program can be written under the assumption that nothing will go wrong.

I will return to the subject of error exits later.

Subscript Checking

In the particular examples given above we can, of course, avoid testing m vs. *max* if we have dynamic range-checking on all subscripts of A. But this usually aborts the program, giving us little or no control over the error recovery; so we probably want to test m anyway. And ouch, what subscript checking does to the inner loop execution times! In Example 2, I will certainly want to suppress range-checking in the **while** clause since its subscript can't be out of range unless A[$m+1$] was already invalid in the previous line. Similarly, in Example 1 there can be no range error in the **for** loop unless a range error occurred earlier. It seems senseless to have expensive range checks in those parts of my programs that I *know* are clean.

In this respect I should mention Hoare's almost persuasive arguments to the contrary [40, p. 18]. He points out quite correctly that the current practice of compiling subscript range checks into the machine code while a program is being tested, then suppressing the checks during production runs, is like a sailor who wears his life preserver while training on land but leaves it behind when he sails! On the other hand, that sailor isn't so foolish if life vests are extremely expensive, and if he is such an excellent swimmer that the chance of needing one is quite small compared with the other risks he is taking. In the foregoing examples we typically are much more certain that the subscripts will be in range than that other aspects of our overall program will work correctly. John Cocke observes that time-consuming range checks can be avoided by a smart compiler which first compiles the checks into the program then moves them out of the loop. Wirth [94] and Hoare [39] have pointed out that a well-designed **for** statement can permit even a rather simple-minded compiler to avoid most range checks within loops.

I believe that range checking should be used far more often than it currently is, but not everywhere. On the other hand I am really assuming infallible hardware when I say this; surely I wouldn't want to remove the parity check mechanism from the hardware, even under a hypothetical assumption that it was slowing down the computation. Additional memory protection is necessary to prevent my program from harming someone else's, and theirs from clobbering mine. My arguments are directed towards compiled-in tests, not towards the hardware mechanisms which are really needed to ensure reliability.

Hash Coding

Now let's move on to another example, based on a standard hashing technique but otherwise designed for the same application as the above. Here $h(x)$ is a hash function which takes on values between 1 and m; and $x \neq 0$. In this case m is somewhat larger than the number of items in the table, and "empty" positions are represented by 0.

Example 3:

```
i := h(x);
while A[i] ≠ 0 do
  begin if A[i] = x then go to found fi;
    i := i−1; if i = 0 then i := m fi;
  end;
not found: A[i] := x; B[i] := 0;
found: B[i] := B[i]+1;
```

If we analyze this as we did Example 1, we see that the trick which led to Example 2 doesn't work any more. Yet if we want to eliminate the **go to** we can apply the idea of Example 1a by writing

while A[i] $\neq$ 0 **and** A[i] $\neq x$ **do** . . .

and by testing afterwards which condition caused termination. This version is perhaps a little bit easier to read; unfortunately it makes a redundant test, which we would like to avoid if we were in a critical part of the program.

Why should I worry about the redundant test in this case? After all, the extra test whether A[i] was $\neq 0$ or $\neq x$ is being made outside of the **while** loop, and I said before that we should generally confine our optimizations to inner loops. Here, the reason is that this **while** loop *won't* usually be a loop at all; with a proper choice of h and m, the operation $i := i-1$ will tend to be executed very infrequently, often less than once per search on the average [54, Section 6.4]. Thus, the entire program of Example 3, except perhaps for the line labeled "not found", must be considered as part of the inner loop, if this search process is a dominant part of the overall program (as it often is). The redundant test will therefore be significant in this case.

Despite this concern with efficiency, I should actually have written the first draft of Example 3 without that **go to** statement, probably even using a **while** clause written in an extended language, such as

while A[i] $\notin \{0, x\}$ **do** . . .

since this formulation abstracts the *real* meaning of what is happening. Someday there may be hardware capable of testing membership in small sets more efficiently than if we program the tests sequentially, so that such a program would lead to better code than Example 3. And there is a much more important reason for preferring this form of the **while** clause: it reflects a symmetry between 0 and x that is not present in Example 3. For example, in most software applications it turns out that the condition A[i] $= x$ terminates the loop far more frequently than A[i] $= 0$; with this knowledge, my second draft of the program would be the following.

Example 3a:

```
i := h(x);
while A[i] ≠ x do
  begin if A[i] = 0
    then A[i] := x; B[i] := 0;
      go to found;
    fi;
    i := i−1; if i = 0 then i := m fi;
  end;
found: B[i] := B[i]+1;
```

This program is easy to derive from the **go to**-less form, but not from Example 3; and it is better than Example 3. So, again we see the advantage of delaying optimizations until we have obtained more knowledge of a program's behavior.

It is instructive to consider Example 3a further, assuming now that the **while** loop is performed many times per search. Although this should not happen in most applications of hashing, there are other programs in which a loop of the above form is present, so it is worth examining what we should do in such circumstances. If the **while** loop becomes an inner loop affecting the overall program speed, the whole picture changes; that redundant test outside the loop becomes utterly negligible, but the test "**if** $i = 0$" suddenly looms large. We generally want to avoid testing conditions that are almost always false, inside a critical loop. Therefore, under these new assumptions I would change the data structure by adding a new element A[0] $= 0$ to the array and eliminating the test for $i = 0$ as follows.

Example 3b:

```
i := h(x);
while A[i] ≠ x do
  if A[i] ≠ 0
  then i := i−1
  else if i = 0
    then i := m;
    else A[i] := x; B[i] := 0;
      go to found;
    fi;
  fi;
found: B[i] := B[i]+1;
```

The loop now is noticeably faster. Again, I would be unhappy with slow subscript range checks if this loop were critical. Incidentally, Example 3b was derived from Example 3a, and a rather different program would have emerged if the same idea had been applied to Example 3; then a test "**if** $i = 0$" would have been inserted *outside* the loop, at label "not found", and another **go to** would have been introduced by the optimization process.

As in the first examples, the program in Example 3 is flawed in failing to test for memory overflow. I should have done this, for example by keeping a count, n, of how many items are nonzero. The "not found" routine should then begin with something like "$n := n+1$; **if** $n = m$ **then go to** memory overflow".

Text Scanning

The first time I consciously applied the top-down structured programming methodology to a reasonably complex job was in the late summer of 1972, when I wrote a program to prepare the index to my book *Sorting and Searching* [54]. I was quite pleased with the way that program turned out (there was only one serious bug), but I did use one **go to** statement. In this case the reason was somewhat different, having nothing to do with exiting from loops; I was exiting, in fact, from an **if-then-else** construction.

The following example is a simplified version of the situation I encountered. Suppose we are processing a stream of text, and that we want to read and print the next character from the input; however, if that character is a slash ("/") we want to "tabulate" instead (i.e., to advance in the output to the next tab-stop position on the current line); however, two consecutive slashes means a "carriage return" (i.e., to advance in the output to the beginning of the next line). After printing a period (".") we also want to insert an additional space in the output. The following code clearly does the trick.

Example 4:

```
x := read char;
if x = slash
then x := read char;
  if x = slash
  then return the carriage;
    go to char processed;
  else tabulate;
  fi;
fi;
write char (x);
if x = period then write char (space) fi;
char processed:
```

An abstract program with similar characteristics has been studied by Peterson et al. [77; Fig. 1(a)]. In practice we occasionally run into situations where a sequence of decisions is made via nested **if-then-else**'s, and then two or more of the branches merge into one. We can manage such decision-table tasks without **go to**'s by copying the common code into each place, or by defining it as a **procedure**, but this does not seem conceptually simpler than to make **go to** a common part of the program in such cases. Thus in Example 4 I could avoid the **go to** by copying "*write char* (x); **if** x = *period* **then** *write char* (*space*) **fi**" into the program after "*tabulate*;" and by making corresponding changes. But this would be a pointless waste of energy just to eliminate a perfectly understandable **go to** statement: the resulting program would actually be harder to maintain than the former, since the action of printing a character now appears in two different places. The alternative of declaring procedures avoids the latter problem, but it is not especially attractive either. Still another alternative is:

Example 4a:

```
x := read char;
double slash := false;
if x = slash
then x := read char;
  if x = slash
  then double slash := true;
  else tabulate;
  fi;
```

```
fi;
if double slash
then return the carriage;
else write char(x);
  if x = period then write char (space) fi;
fi;
```

I claim that this is conceptually no simpler than Example 4; indeed, one can argue that it is actually more difficult, because it makes the *entire* routine aware of the "double slash" exception to the rules, instead of dealing with it in one exceptional place.

A Confession

Before we go on to another example, I must admit what many readers already suspect, namely, that I'm subject to substantial bias because I actually have a vested interest in **go to** statements! The style for the series of books I'm writing was set in the early 1960s, and it would be too difficult for me to change it now; I present algorithms in my books using informal English language descriptions, and **go to** or its equivalent is almost the only control structure I have. Well, I rationalize this apparent anachronism by arguing that: a) an informal English description seems advantageous because many readers tell me they automatically read English, but skip over formal code; b) when **go to** statements are used judiciously together with comments stating nonobvious loop invariants, they are semantically equivalent to **while** statements, except that indentation is missing to indicate the structure; c) the algorithms are nearly always short, so that accompanying flowcharts are able to illustrate the structure; d) I try to present algorithms in a form that is most efficient for implementation, and high-level structures often don't do this; e) many readers will get pleasure from converting my semiformal algorithms into beautifully structured programs in a formal programming language; and f) we are still learning much about control structures, and I can't afford to wait for the final consensus.

In spite of these rationalizations, I'm uncomfortable about the situation, because I find others occasionally publishing examples of algorithms in "my" style but without the important parenthesized comments and/or with unrestrained use of **go to** statements. In addition, I also know of places where I have myself used a complicated structure with excessively unrestrained **go to** statements, especially the notorious Algorithm 2.3.3A for multivariate polynomial addition [50]. The original program had at least three bugs; exercise 2.3.3-14, "Give a formal proof (or disproof) of the validity of Algorithm A", was therefore unexpectedly easy. Now in the second edition, I believe that the revised algorithm is correct, but I still don't know any good way to prove it; I've had to raise the difficulty rating of exercise 2.3.3-14, and I hope someday to see the algorithm cleaned up without loss of its efficiency.

My books emphasize efficiency because they deal with algorithms that are used repeatedly as building blocks in a large variety of applications. It is important to keep efficiency in its place, as mentioned above, but when efficiency counts we should also know how to achieve it.

In order to make it possible to derive quantitative assessments of efficiency, my books show how to analyze machine language programs; and these programs are expressed in MIXAL, a symbolic assembly language that explicitly corresponds one-for-one to machine language. This has its uses, but there is a danger of placing too much stress on assembly code. Programs in MIXAL are like programs in machine language, devoid of structure; or, more precisely, it is difficult for our eyes to perceive the program structure. Accompanying comments explain the program and relate it to the global structure illustrated in flowcharts, but it is not so easy to understand what is going on; and it is easy to make mistakes, partly because we rely so much on comments which might possibly be inaccurate descriptions of what the program really does. It is clearly better to write programs in a language that reveals the control structure, even if we are intimately conscious of the hardware at each step; and therefore I will be discussing a structured assembly language called PL/MIX in the fifth volume of *The art of computer programming*. Such a language (analogous to Wirth's PL360 [95]) should really be supported by each manufacturer for each machine in place of the old-fashioned structureless assemblers that still proliferate.

On the other hand I'm not really unhappy that MIXAL programs appear in my books, because I believe that MIXAL is a good example of a "quick and dirty assembler", a genre of software which will always be useful in its proper role. Such an assembler is characterized by language restrictions that make simple one-pass assembly possible, and it has several noteworthy advantages when we are first preparing programs for a new machine: a) it is a great improvement over numeric machine code; b) its rules are easy to state; and c) it can be implemented in an afternoon or so, thus getting an efficient assembler working quickly on what may be very primitive equipment. So far I have implemented six such assemblers, at different times in my life, for machines or interpretive systems or microprocessors that had no existing software of comparable utility; and in each case other constraints made it impractical for me to take the extra time necessary to develop a good, structured assembler. Thus I am sure that the concept of quick-and-dirty-assembler is useful, and I'm glad to let MIXAL illustrate what one is like. However, I also believe strongly that such languages should never be improved to the point where they are too easy or too pleasant to use; one must restrict their use to primitive facilities that are easy to implement efficiently. I would never switch to a two-pass process, or add complex pseudo-operations, macro-facilities, or even fancy error diagnostics to such a language, nor would I maintain or distribute such a language as a standard programming tool for a real machine. All such ameliorations and refinements should appear in a structured assembler. Now that the technology is available, we can condone unstructured languages only as a bootstrap-like means to a limited end, when there are strong economic reasons for not implementing a better system.

Tree Searching

But, I'm digressing from my subject of **go to** elimination in higher level languages. A few weeks ago I decided to choose an algorithm at random from my books, to study its use of **go to** statements. The very first example I encountered [54, Algorithm 6.2.3C] turned out to be another case where existing programming languages have no good substitute for **go to**'s. In simplified form, the loop where the trouble arises can be written as follows.

Example 5:

```
compare:
if A[i] < x
then if L[i] ≠ 0
  then i := L[i]; go to compare;
  else L[i] := j; go to insert fi;
else if R[i] ≠ 0
  then i := R[i]; go to compare;
  else R[i] := j; go to insert fi;
fi;
insert: A[j] := x;
L[j] := 0; R[j] := 0; j := j+1;
```

This is part of the well-known "tree search and insertion" scheme, where a binary search tree is being represented by three arrays: A[i] denotes the information stored at node number i, and L[i], R[i] are the respective node numbers for the roots of that node's left and right subtrees; empty subtrees are represented by zero. The program searches down the tree until finding an empty subtree where x can be inserted; and variable j points to an appropriate place to do the insertion. For convenience, I have assumed in this example that x is not already present in the search tree.

Example 5 has four **go to** statements, but the control structure is saved from obscurity because the program is so beautifully symmetric between L and R. I know that these **go to** statements can be eliminated by introducing a Boolean variable which becomes true when L[i] or R[i] is found to be zero. But I don't want to test this variable in the inner loop of my program.

Systematic Elimination

A good deal of theoretical work has been addressed to the question of **go to** elimination, and I shall now try to summarize the findings and to discuss their relevance.

S. C. Kleene proved a famous theorem in 1956 [48] which says, in essence, that the set of all paths through any flowchart can be represented as a "regular expression" R built up from the following operations:

s	the single arc s of the flowchart
R_1; R_2	concatenation (all paths consisting of a path of R_1 followed by a path of R_2)
$R_1 \cup R_2$	union (all paths of either R_1 or R_2)
R^+	iteration (all paths of the form p_1; p_2; $\cdots$; p_n for some $n \geq 1$, where each p_i is a path of R)

These regular expressions correspond loosely to programs consisting of statements in a programming language related by the three operations of sequential composition, conditionals (**if-then-else**), and iterations (**while** loops). Thus, we might expect that these three program control structures would be sufficient for all programs. However, closer analysis shows that Kleene's theorem does not relate directly to control structures; the problem is only superficially similar. His result is suggestive but not really applicable in this case.

The analogous result for control structures was first proved by G. Jacopini in 1966, in a paper written jointly with C. Böhm [8]. Jacopini showed, in effect, that any program given, say, in flowchart form can be transformed systematically into another program, which computes the same results and which is built up from statements in the original program using only the three basic operations of composition, conditional, and iteration, plus possible assignment statements and tests on auxiliary variables. Thus, in principle, **go to** statements can always be removed. A detailed exposition of Jacopini's construction has been given by H. D. Mills [69].

Recent interest in structured programming has caused many authors to cite Jacopini's result as a significant breakthrough and as a cornerstone of modern programming technique. Unfortunately, these authors are unaware of the comments made by Cooper in 1967 [16] and later by Bruno and Steiglitz [10], namely, that from a practical standpoint the theorem is meaningless. Indeed, any program can obviously be put into the "beautifully structured" form

```
p := 1;
while p > 0 do
  begin if p = 1 then perform step 1;
      p := successor of step 1 fi;
    if p = 2 then perform step 2;
      p := successor step 2 fi;
    ...
    if p = n then perform step n;
      p := successor of step n fi;
  end.
```

Here the auxiliary variable p serves as a program counter representing which box of the flowchart we're in, and the program stops when p is set to zero. We have eliminated all **go to**'s, but we've actually lost all the structure.

Jacopini conjectured in his paper that auxiliary variables are necessary in general, and that the **go to**'s in a program of the form

```
L1: if B1 then go to L2 fi;
      S1;
    if B2 then go to L2 fi;
      S2;
    go to L1;
L2: S3;
```

cannot always be removed unless additional computation is done. Floyd and I proved this conjecture with John Hopcroft's help [52]. Sharper results were later obtained by Ashcroft and Manna [1], Bruno and Steiglitz [10], Kosaraju [57], and Peterson, Kasami, and Tokura [77].

Jacopini's original construction was not merely the trivial flowchart emulation scheme indicated above; he was able to salvage much of the given flowchart structure if it was reasonably well-behaved. A more general technique of **go to** elimination, devised by Ashcroft and Manna [1], made it possible to capture still more of a given program's natural flow; for example, their technique applied to Example 5 yields

Example 5a:

```
t := true;
while t do
  begin if A[i] < x
    then if L[i] ≠ 0 then i := L[i];
      else L[i] := j; t := false fi;
    else if R[i] ≠ 0 then i := R[i];
      else R[i] := j; t := false fi;
  end;
A[j] := x;
```

But, in general, their technique may cause a program to grow exponentially in size; and when error exits or other recalcitrant **go to**'s are present, the resulting programs will indeed look rather like the flowchart emulator sketched above.

If such automatic **go to** elimination procedures are applied to badly structured programs, we can expect the resulting programs to be at least as badly structured. Dijkstra pointed this out already in [23], saying:

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.

In other words, we shouldn't merely remove **go to** statements because it's the fashionable thing to do; the presence or absence of **go to** statements is not really the issue. The underlying structure of the program is what counts, and we want only to avoid usages which somehow clutter up the program. Good structure can be expressed in FORTRAN or COBOL, or even in assembly language, although less clearly and with much more trouble. The real goal is to formulate our programs in such a way that they are easily understood.

Program structure refers to the way in which a complex algorithm is built up from successively simpler processes. In most situations this structure can be described very nicely in terms of sequential composition, conditionals, simple iterations, and with **case** statements for multiway branches; undisciplined **go to** statements make program structure harder to perceive, and they are often symptoms of a poor conceptual formulation. But there has been far too much emphasis on **go to** elimination instead of on the really important issues; people have a natural tendency to set up an easily understood quantitative goal like the abolition of jumps, instead of working directly for a qualitative goal like good program structure. In a similar way, many people have set up "zero population growth" as a goal to be achieved, when they really desire living conditions that are much harder to quantify.

Probably the worst mistake any one can make with respect to the subject of **go to** statements is to assume that "structured programming" is achieved by writing programs as we always have and then eliminating the **go to**'s. Most **go to**'s shouldn't be there in the first place! What we really want is to conceive of our program in such a way that we rarely even *think* about **go to** statements, because the real need for them hardly ever arises. The language in which we express our ideas has a strong influence on our thought processes. Therefore, Dijkstra [23] asks for more new language features—structures which encourage clear thinking—in order to avoid the **go to**'s temptations toward complications.

Event Indicators

The best such language feature I know has recently been proposed by C. T. Zahn [102]. Since this is still in the experimental stage, I will take the liberty of modifying his "syntactic sugar" slightly, without changing his basic idea. The essential novelty in his approach is to introduce a new quantity into programming languages, called an *event indicator* (not to be confused with concepts from PL/I or SIMSCRIPT). My current preference is to write his event-driven construct in the following two general forms.

```
A)  loop until ⟨event⟩₁ or ··· or ⟨event⟩ₙ:
          ⟨statement list⟩₀;
    repeat;
    then ⟨event⟩₁ => ⟨statement list⟩₁;
          ⋮
          ⟨event⟩ₙ => ⟨statement list⟩ₙ;
    fi;

B)  begin until ⟨event⟩₁ or ··· or ⟨event⟩ₙ;
          ⟨statement list⟩₀;
    end;
    then ⟨event⟩₁ => ⟨statement list⟩₁;
          ⋮
          ⟨event⟩ₙ => ⟨statement list⟩ₙ;
    fi;
```

There is also a new statement, "⟨event⟩", which means that the designated event has occurred: such a statement is allowed only within ⟨statement list⟩$_0$ of an **until** construct which declares that event.

In form (A), ⟨statement list⟩$_0$ is executed repeatedly until control leaves the construct entirely or until one of the named events occurs; in the latter case, the statement list corresponding to that event is executed. The behavior in form (B) is similar, except that no iteration is implied; one of the named events must have occurred before the **end** is reached. The **then** ··· **fi** part may be omitted when there is only one event name.

The above rules should become clear after looking at what happens when Example 5 above is recoded in terms of this new feature:

Example 5b:

```
loop until left leaf hit or
            right leaf hit:
  if A[i] < x
  then if L[i] ≠ 0 then i := L[i];
    else left leaf hit fi;
  else if R[i] ≠ 0 then i := R[i];
    else right leaf hit fi;
  fi;
repeat;
then left leaf hit => L[i] := j;
      right leaf hit => R[i] := j;
fi;
A[j] := x; L[j] := 0; R[j] := 0; j := j+1;
```

Alternatively, using a single event,

Example 5c:

```
loop until leaf replaced:
  if A[i] < x
  then if L[i] ≠ 0 then i := L[i]
    else L[i] := j; leaf replaced fi;
  else if R[i] ≠ 0 then i := R[i]
    else R[i] := j; leaf replaced fi;
  fi;
repeat;
A[j] := x; L[j] := 0; R[j] := 0; j := j+1;
```

For reasons to be discussed later, Example 5b is preferable to 5c.

It is important to emphasize that the first line of the construct merely declares the event indicator names, and that event indicators are *not* conditions which are being tested continually; ⟨event⟩ statements are simply transfers of control which the compiler can treat very efficiently. Thus, in Example 5c the statement "leaf replaced" is essentially a **go to** which jumps out of the loop.

This use of events is, in fact, semantically equivalent to a restricted form of **go to** statement, which Peter Landin discussed in 1965 [58] before most of us were ready to listen. Landin's device has been reformulated by Clint and Hoare [14] in the following way: Labels are declared at the beginning of each block, just as procedures normally are, and each label also has a ⟨label body⟩ just as a procedure has a ⟨procedure body⟩. Within the block whose heading contains such a declaration of label L, the statement **go to** L according to this scheme means "execute the body of L, then leave the block". It is easy to see that this is exactly the form of control provided by Zahn's event mechanism, with the ⟨label body⟩s replaced by ⟨statement list⟩s in the **then** ··· **fi** postlude and with ⟨event⟩ statements corresponding to Landin's **go to.** Thus, Clint and Hoare would have written Example 5b as follows.

```
while true do
  begin label left leaf hit; L[i] := j;
    label right leaf hit; R[i] := j;
    if A[i] < x
    then if L[i] ≠ 0 then i := L[i];
      else go to left leaf hit fi;
    else if R[i] ≠ 0 then i := R[i];
      else go to right leaf hit fi;
  end;
A[j] := x; L[j] := 0; R[j] := 0; j := j+1;
```

I believe the program reads much better in Zahn's form, with the ⟨label body⟩s set in the code between that which logically precedes and follows.

Landin also allowed his "labels" to have parameters like any other procedures; this is a valuable extension to Zahn's proposal, so I shall use events with value parameters in several of the examples below.

As Zahn [102] has shown, event-driven statements blend well with the ideas of structured programming by stepwise refinement. Thus, Examples 1 to 3 can all be cast into the following more abstract form, using an event "found" with an integer parameter:

```
begin until found:
  search table for x and
  insert it if not present;
end;
then found (integer j) => B[j] := B[j]+1;
fi;
```

This much of the program can be written before we have decided how to maintain the table. At the next level of abstraction, we might decide to represent the table as a sequential list, as in Example 1, so that "search table ⋯ " would expand into

```
for i := 1 step 1 until m do
  if A[i] = x then found(i) fi;
m := m+1; A[m] := x; found(m);
```

Note that this **for** loop is more disciplined than the one in our original Example 1, because the iteration variable is not used outside the loop; it now conforms to the rules of ALGOL W and ALGOL 68. Such **for** loops provide convenient documentation and avoid common errors associated with global variables; their advantages have been discussed by Hoare [39].

Similarly, if we want to use the idea of Example 2 we might write the following code as the refinement of "search table ⋯":

```
begin integer i;
  A[m+1] := x; i := 1;
  while A[i] ≠ x do i := i+1;
  if i > m then m := i; B[m] := 0 fi;
  found(i);
end;
```

And finally, if we decide to use hashing, we obtain the equivalent of Example 3, which might be written as follows.

```
begin integer i;
  i := h(x);
  loop until present or absent:
    if A[i] = x then present fi;
    if A[i] = 0 then absent fi;
    i := i − 1;
    if i = 0 then i := m fi;
  repeat;
  then present => found(i);
       absent => A[i] := x; found(i);
  fi;
end;
```

The **begin until** ⟨event⟩ construct also provides a natural way to deal with decision-table constructions such as the text-scanning application we have discussed.

Example 4b:

```
begin until normal character input
    or double slash:
  char x;
  x := read char;
  if x = slash
  then x := read char;
       if x = slash
       then double slash;
       else tabulate;
         normal character input (x);
       fi;
     else normal character input (x);
     fi;
  end;
  then normal character input (char x) =>
       write char (x);
       if x = period then write char (space) fi;
     double slash => return the carriage,
  fi;
```

This program states the desired actions a bit more clearly than any of our previous attempts were able to do.

Event indicators handle error exits too. For example, we might write a program as follows.

```
begin until error or normal end:
  ...
  if m = max then error ('symbol table full') fi;
  ...
  normal end;
end;
then error (string S) =>
     print ('unrecoverable error,'; S);
   normal end =>
     print ('computation complete');
fi;
```

Comparison of Features

Of course, event indicators are not the only decent alternatives to **go to** statements that have been proposed. Many authors have suggested language features which provide roughly equivalent facilities, but which are expressed in terms of **exit, jumpout, break**, or **leave** statements. Kosaraju [57] has proved that such statements are sufficient to express all programs without **go to**'s and without any extra computation, but only if an exit from arbitrarily many levels of control is permitted.

The earliest language features of this kind (besides Landin's proposal) provided essentially only one exit from a loop; this means that the code appearing in the **then** ⋯ **fi** postlude of our examples would be inserted into the body itself before branching. (See Example 5c.) The separation of such code as in Zahn's proposal is better, mainly because the body of the construct corresponds to code that is written under different "invariant assumptions" which are inoperative after a particular event has occurred.

Thus, each event corresponds to a particular set of assertions about the state of the program, and the code which follows that event takes cognizance of these assertions, which are rather different from the assertions in the main body of the construct. (For this reason I prefer Example 5b to Example 5c.)

Language features allowing multiple exits have been proposed by G. V. Bochmann [7], and independently by Shigo et al. [86]. These are semantically equivalent to Zahn's proposals, with minor variations; but they express such semantics in terms of statements that say "exit to ⟨label⟩". I believe Zahn's idea of event indicators is an improvement on the previous schemes, because the specification of events instead of labels encourages a better *conception* of the program. The identifier given to a label is often an imperative verb like "insert" or "compare", saying what action is to be done next, while the appropriate identifier for an event is more likely to be an adjective like "found". The names of events are very much like the names of Boolean variables, and I believe this accounts for the popularity of Boolean variables as documentation aids, in spite of their inefficiency.

Putting this another way, it is much better from a psychological standpoint to write

```
    loop until found ···; found; ··· repeat
```

than to write

```
search: while true do
  begin ··· ; leave search; ··· end.
```

The **leave** or **exit** statement is operationally the same, but intuitively different, since it talks more about the program than about the problem.

The PL/I language allows programmer-defined ON-conditions, which are similar in spirit to event indicators. A programmer first *executes* a statement "ON CONDITION (identifier) block" which specifies a block of code that is to be executed when the identified event occurs, and an occurrence of that event is indicated by writing SIGNAL CONDITION (identifier). However, the analogy is not very close, since control returns to the statement following the SIGNAL statement after execution of the specified block of code, and the block may be dynamically respecified.

Some people have suggested to me that events should be called "conditions" instead, by analogy with Boolean expressions. However, that terminology would tend to imply a relation which is continually being monitored, instead of a happening. By writing "**loop until** *yprime* is near *y*: ···" we seem to be saying that the machine should keep track of whether or not y and *yprime* are nearly equal; a better choice of words would be an event name like "**loop until** convergence established: ···" so that we can write "**if** $abs(yprime - y) < epsilon \times y$ **then** convergence established". An event occurs when the program has *discovered* that the state of computation has changed.

Simple Iterations

So far I haven't mentioned what I believe is really the most common situation in which **go to** statements are needed by an ALGOL or PL/I programmer, namely a simple iterative loop with one entrance and one exit. The iteration statements most often proposed as alternatives to **go to** statements have been "**while** B **do** S" and "**repeat** S **until** B". However, in practice, the iterations I encounter very often have the form

```
A: S;
   if B then go to Z fi;
   T; go to A;
Z:
```

where S and T both represent reasonably long sequences of code. If S is empty, we have a **while** loop, and if T is empty we have a **repeat** loop, but in the general case it is a nuisance to avoid the **go to** statements.

A typical example of such an iteration occurs when S is the code to acquire or generate a new piece of data, B is the test for end of data, and T is the processing of that data. Another example is when the code preceding the loop sets initial conditions for some iterative process; then S is a computation of quantities involved in the test for convergence, B is the test for convergence, and T is the adjustment of variables for the next iteration.

Dijkstra [29] aptly named this a loop

which is performed "n and a half times". The usual practice for avoiding **go to**'s in such loops is either to duplicate the code for S, writing

```
S; while B̄ do begin T; S end;
```

where $\bar{B}$ is the negation of relation B; or to figure out some sort of "inverse" for T so that "T^{-1}; T" is equivalent to a null statement, and writing

```
T⁻¹; repeat T; S until B;
```

or to duplicate the code for B and to make a redundant test, writing

```
repeat S; if B̄ then T fi; until B;
```

or its equivalent. The reader who studies **go to**-less programs as they appear in the literature will find that all three of these rather unsatisfactory constructions are used frequently.

I discussed this weakness of ALGOL in a letter to Niklaus Wirth in 1967, and he proposed two solutions to the problem, together with many other instructive ideas in an unpublished report on basic concepts of programming languages [94]. His first suggestion was to write

```
repeat begin S; when B exit; T; end;
```

and readers who remember 1967 will also appreciate his second suggestion,

```
turn on begin S; when B drop out; T; end.
```

Neither set of delimiters was felt to be quite right, but a modification of the first proposal (allowing one or more single-level **exit** statements within **repeat begin** ··· **end**) was later incorporated into an experimental version of the ALGOL W language. Other languages such as BCPL and BLISS incorporated and extended the **exit** idea, as mentioned above. Zahn's construction now allows us to write, for example,

```
loop until all data exhausted:
   S;
   if B then all data exhausted fi;
   T;
repeat;
```

and this is a better syntax for the $n + \frac{1}{2}$ problem than we have had previously.

On the other hand, it would be nicest if our language would provide a single feature which covered all simple iterations without going to a rather "big" construct like the event-driven scheme. When a programmer uses the simpler feature he is thereby making it clear that he has a simple iteration, with exactly one condition which is being tested exactly once each time around the loop. Furthermore, by providing special syntax for this common case we make it easier for a compiler to produce more efficient code, since the compiler can rearrange the machine instructions so that the test appears physically at the end of loop. (Many hours of computer time are now wasted each day executing unconditional jumps to the beginning of loops.)

Ole-Johan Dahl has recently proposed a syntax which I think is the first real solution to the $n + \frac{1}{2}$ problem. He suggests writing the general simple iteration defined above as

```
loop; S; while B̄: T; repeat;
```

where, as before, S and T denote sequences of one or more statements separated by semicolons. Note that as in two of our original **go to**-free examples, the syntax refers to condition $\bar{B}$ which represents staying *in* the iteration, instead of condition B which represents exiting; and this may be the secret of its success.

Dahl's syntax may not seem appropriate at first, but actually it reads well in every example I have tried, and I hope the reader will reserve judgment until seeing the examples in the rest of this paper. One of the nice properties of his syntax is that the word **repeat** occurs naturally at the end of a loop rather than at its beginning, since we read the actions of the program sequentially. As we reach the end, we are instructed to repeat the loop, instead of being informed that the *text* of the loop (not its execution) has ended. Furthermore, the above syntax avoids ALGOL's use of the word **do** (and also the more recent unnatural delimiter **od**); the word **do** as used in ALGOL has never sounded quite right to native speakers of English, it has always been rather quaint for us to say "**do** *read* (A[i])" or "**do begin**"! Another feature of Dahl's proposals is that it is easily axiomatized along the lines proposed by Hoare [37, 41]:

$$\frac{\{P\}S\{Q\} \qquad \{Q \wedge \bar{B}\}T\{P\}}{\{P\}\ \textbf{loop}: S;\ \textbf{while}\ \bar{B}: T;\ \textbf{repeat};\ \{Q \wedge \neg \bar{B}\}}$$

(Here I am using braces around the assertions, as in Wirth's Pascal language [97], instead of following Hoare's original notation "P {S} Q", since assertions are, by nature, parenthetical remarks.)

The nicest thing about Dahl's proposal is that it works also when S or T is empty, so that we have a uniform syntax for all three cases; the **while** and **repeat** statements found in Algol-like languages of the late 1960s are no longer needed. When S or T is empty, it is appropriate to delete the preceding colon. Thus

```
loop while B̄:
  T;
repeat;
```

takes the place of "**while** B̄ **do begin** T **end**;" and

```
loop:
  S
while B̄ repeat;
```

takes the place of "**repeat** S **until** B;". At first glance these may seem strange, but probably less strange than the **while** and **repeat** statements did when we first learned them.

If I were designing a programming language today, my current preference would be to use Dahl's mechanism for simple iteration, plus Zahn's more general construct, plus a **for** statement whose syntax would be perhaps

```
loop for 1 ≤ i ≤ n:
  S;
repeat;
```

with appropriate extensions. These control structures, together with **if** ··· **then** ··· **else** ··· **fi**, will comfortably handle all the examples discussed so far in this paper, without any **go to** statements or loss of efficiency or clarity. Furthermore, none of these language features seems to encourage overly-complicated program structure.

2. INTRODUCTION OF go to STATEMENTS

Now that I have discussed how to remove **go to** statements, I will turn around and show why there are occasions when I actually wish to *insert* them into a **go to**-less program. The reason is that I like well-documented programs very much, but I dislike inefficient ones; and there are some cases where I simply seem to need **go to** statements, despite the examples stated above.

Recursion Elimination

Such cases come to light primarily when I'm trying to optimize a program (originally well-structured), often involving the removal of implicit or explicit recursion. For example, consider the following recursive procedure that prints the contents of a binary tree in symmetric order. The tree is represented by L, A, and R arrays as in Example 5, and the recursive procedure is essentially the *definition* of symmetric order.

Example 6:

```
procedure treeprint(t); integer t; value t;
  if t ≠ 0
  then treeprint(L[t]);
    print(A[t]);
    treeprint(R[t]);
  fi;
```

This procedure may be regarded as a model for a great many algorithms which have the same structure, since tree traversal occurs in so many applications; we shall assume for now that printing is our goal, with the understanding that this is only one instance of a general family of algorithms.

It is often useful to remove recursion from an algorithm, because of important economies of space or time, even though this tends to cause some loss of the program's basic clarity. (And, of course, we might also have to state our algorithm in a language like Fortran or in a machine language that doesn't allow recursion.) Even when we use Algol or PL/I, every compiler I know imposes considerable overhead on procedure calls; this is to a certain extent inevitable because of the generality of the parameter mechanisms, especially call by name and the maintenance of proper dynamic environments. When procedure calls occur in an inner loop the overhead can slow a program down by a factor of two or more. But if we hand tailor our own implementation of recursion instead of relying on a general mechanism we can usually find worthwhile simplifications, and in the process we occasionally get a deeper insight into the original algorithm.

There has been a good deal published about recursion elimination (especially in the work of Barron [4], Cooper [15], Manna and Waldinger [61], McCarthy [62], and Strong [88; 91]); but I'm amazed that very little of this is about "down to earth" problems. I have always felt that the transformation from recursion to iteration is one of the most fundamental concepts of computer science, and that a student should learn it at about the time he is studying data structures. This topic is the subject of Chapter 8 in my multivolume work; but it's only by accident that recursion wasn't Chapter 3, since it conceptually belongs very early in the table of contents. The material just wouldn't fit comfortably into any of the earlier volumes; yet there are many algorithms in Chapters 1–7 that are recursions in disguise. Therefore it surprises me that the literature on recursion removal is primarily concerned with "baby" examples like computing factorials or reversing lists, instead of with a sturdy toddler like Example 6.

Now let's go to work on the above example. I assume, of course, that the reader knows the standard way of implementing recursion with a stack [20], but I want to make simplifications beyond this. Rule number one for simplifying procedure calls is:

> If the last action of procedure p before it returns is to call procedure q, simply **go to** the beginning of procedure q instead.

(We must forget for the time being that we don't like **go to** statements.) It is easy to confirm the validity of this rule, if, for simplicity, we assume parameterless procedures. For the operation of calling q is to put a return address on the stack, then to execute q, then to resume p at the return address specified, then to resume the caller of p. The above simplification makes q resume the caller of p. When $q = p$ the argument is perhaps a bit subtle, but it's all right. (I'm not sure who originated this principle; I recall learning it from Gill's paper [34, p. 183], and then seeing many instances of it in connection with top-down compiler organization. Under certain conditions the BLISS/11 compiler [101] is capable of discovering this simplification. Incidentally, the converse of the above principle is also true (see [52]): **go to** statements can always be eliminated by declaring suitable procedures, each of which calls another as its last action. This shows that procedure calls include **go to** statements as a special case; it cannot be argued that procedures are conceptually simpler than **go to**'s, although some people have made such a claim.)

As a result of applying the above simplification, and adapting it in the obvious way to the case of a procedure with one parameter, Example 6 becomes

Example 6a:

```
procedure treeprint(t); integer t; value t;
L: if t ≠ 0
  then treeprint(L[t]);
    print(A[t]);
    t := R[t]; go to L;
  fi;
```

But we don't really want that **go to**, so we might prefer to write the code as follows, using Dahl's syntax for iterations as explained above.

Example 6b:

```
procedure treeprint(t); integer t; value t;
loop while t ≠ 0:
  treeprint(L[t]);
  print(A[t]);
  t := R[t];
repeat;
```

If our goal is to impress somebody, we might tell them that we thought of Example 6b first, instead of revealing that we got it by straightforward simplification of the obvious program in Example 6.

There is still a recursive call in Example 6b; and this time it's embedded in the procedure, so it looks as though we have to go to the general stack implementation. However, the recursive call now occurs in only one place, so we need not put a return address on the stack; only the local variable *t* needs to be saved on each call. (This is another simplification which occurs frequently.) The program now takes the following nonrecursive form.

Example 6c:

```
procedure treeprint(t); integer t; value t;
  begin integer stack S; S := empty;
L1: loop while t ≠ 0:
        S <= t; t := L[t]; go to L1;
L2:     t <= S;
        print(A[t]);
        t := R[t];
    repeat;
    if nonempty(S) then go to L2 fi;
    end.
```

Here for simplicity I have extended ALGOL to allow a "stack" data type, where S <= *t* means "push *t* onto S" and *t* <= S means "pop the top of S to *t*, assuming that S is nonempty".

It is easy to see that Example 6c is equivalent to Example 6b. The statement "**go to** L1" initiates the procedure, and control returns to the following statement (labeled L2) when the procedure is finished. Although Example 6c involves **go to** statements, their purpose is easy to understand, given the knowledge that we have produced Example 6c by a mechanical, completely reliable method for removing recursion. Hopkins [44] has given other examples where **go to** at a low level supports high-level constructions.

But if you look at the above program again, you'll probably be just as shocked as I was when I first realized what has happened. I had always thought that the use of **go to** statements was a bit sinful, say a "venial sin"; but there was one kind of **go to** that I certainly had been taught to regard as a mortal sin, perhaps even unforgivable, namely one which goes into the middle of an iteration! Example 6c does precisely that, and it is perfectly easy to understand Example 6c by comparing it with Example 6b. In this particular case we can remove the **go to**'s without difficulty; but in general when a recursive call is embedded in ~~general when a recursive call is embedded in~~ several complex levels of control, there is no equally simple way to remove the recursion without resorting to something like Example 6c. As I say, it was a shock when I first ran across such an example. Later, Jim Horning confessed to me that he also was guilty, in the syntax-table-building program for the XPL system [65, p. 500], because XPL doesn't allow recursion; see also [56]. Clearly a new doctrine about sinful **go to**'s is needed, some sort of "situation ethics".

The new morality that I propose may perhaps be stated thus: "Certain **go to** statements which arise in connection with well-understood transformations are acceptable, provided that the program documentation explains what the transformation was." The use of four-letter words like **goto** can occasionally be justified even in the best of company.

This situation is very similar to what people have commonly encountered when proving a program correct. To demonstrate the validity of a typical program Q, it is usually simplest and best to prove that some rather simple but less efficient program P is correct and then to prove that P can be transformed into Q by a sequence of valid optimizations. I'm saying that a similar thing should be considered standard practice for all but the simplest software programs: A programmer should create a program P which is readily understood and well-documented, and then he should optimize it into a program Q which is very efficient. Program Q may contain **go to** statements and other low-level features, but the transformation from P to Q should be accomplished by completely reliable and well-documented "mechanical" operations.

At this point many readers will say, "But he should only write P, and an optimizing compiler will produce Q." To this I say, "No, the optimizing compiler would have to be so complicated (much more so than anything we have now) that it will in fact be *un*reliable." I have another alternative to propose, a new class of software which will be far better.

Program Manipulation Systems

For 15 years or so I have been trying to think of how to write a compiler that really produces top quality code. For example, most of the Mix programs in my books are considerably more efficient than any of today's most visionary compiling schemes would be able to produce. I've tried to study the various techniques that a hand-coder like myself uses, and to fit them into some systematic and automatic system. A few years ago, several students and I looked at a typical sample of FORTRAN programs [51], and we all tried hard to see how a machine could produce code that would compete with our best hand-optimized object programs. We found ourselves always running up against the same problem: the compiler needs to be in a dialog with the programmer; it needs to know properties of the data, and whether certain cases can arise, etc. And we couldn't think of a good language in which to have such a dialog.

For some reason we all (especially me) had a mental block about optimization, namely that we always regarded it as a behind-the-scenes activity, to be done in the machine language, which the programmer isn't supposed to know. This veil was first lifted from my eyes in the Fall of 1973, when I ran across a remark by Hoare [42] that, ideally, a language should be designed so that an optimizing compiler can describe its optimizations in the *source* language. Of course! Why hadn't I ever thought of it?

Once we have a suitable language, we will be able to have what seems to be emerging as the programming system of the future: an interactive *program-manipulation system*, analogous to the many symbol-manipulation systems which are presently undergoing extensive development and experimentation. We are gradually learning about program transformations, which are more complicated than formula manipulations but really not very different. A program-manipulation system is obviously what we've been leading up to, and I wonder why I never thought of it before. Of course, the idea isn't original with me; when I told Hoare, he said, "Exactly!" and referred me to a recent paper by Darlington and Burstall [18]. Their paper describes a system which removes some recursions from a LISP-like language (curiously, without introducing any **go to**'s), and which also does some conversion of data structures (from sets to lists or bit strings) and some restructuring of a program by combining similar loops. I later discovered that program manipulation is just part of a much more ambitious project undertaken by Cheatham and Wegbreit [12]; another paper about source-code optimizations has also recently appeared [83]. Since LISP programs are easily manipulated as LISP data objects, there has also been a rather extensive development of similar ideas in this domain, notably by Warren Teitelman (see [89, 90]). The time is clearly ripe for program-manipulation systems, and a great deal of further work suggests itself.

The programmer using such a system will write his beautifully-structured, but possibly inefficient, program P; then he will interactively specify transformations that make it efficient. Such a system will be much more powerful and reliable than a completely automatic one. We can also imagine the system manipulating measurement statistics concerning how much of the total running time is spent in each statement, since the programmer will want to know which parts of his program deserve to be optimized, and how much effect an optimization will really have. The original program P should be retained along with the transformation specifications, so that it can be properly understood and maintained as time passes. As I say, this idea certainly isn't my own; it is so exciting I hope that *everyone* soon becomes aware of its possibilities.

A "calculus" of program transformations is gradually emerging, a set of operations which can be applied to programs without rethinking the specific problem each time. I have already mentioned several of these transformations: doubling up of loops (Example 2a), changing final calls to **go to**'s (Example 6a), using a stack for recursions (Example 6c), and combining disjoint loops over the same range [18]. The idea of macro-expansions in general seems to find its most appropriate realization as part of a program manipulation system.

Another well-known example is the removal of invariant subexpressions from loops. We are all familiar with the fact that a program which includes such subexpressions is more readable than the corresponding program with invariant subexpressions moved out of their loops; yet we consciously remove them when the running time of the program is important.

Still another type of transformation occurs when we go from high-level "abstract" data structures to low-level "concrete" ones (see Hoare's chapter in [17] for numerous examples). In the case of Example 6c, we can replace the stack by an array and a pointer, arriving at

Example 6d:

```
procedure treeprint(t); integer t; value t;
  begin integer array S[1:n]; integer k; k := 0;
L1: loop while t ≠ 0:
      k := k+1; S[k] := t;
      t := L[t]; go to L1;
L2:   t := S[k]; k := k−1;
      print(A[t]);
      t := R[t];
    repeat;
    if k ≠ 0 then go to L2 fi;
  end.
```

Here the programmer must specify a safe value for the maximum stack size n, in order to make the transformation legitimate. Alternatively, he may wish to implement the stack by a linked list. This choice can usually be made without difficulty, and it illustrates another area in which interaction is preferable to completely automatic transformations.

Recursion vs. Iteration

Before leaving the *treeprint* example, I would like to pursue the question of **go to** elimination from Example 6c, since this leads to some interesting issues. It is clear that the first **go to** is just a simple iteration, and a little further study shows that Example 6c is just one simple iteration inside another, namely (in Dahl's syntax)

Example 6e:

```
procedure treeprint(t); integer t; value t;
  begin integer stack S; S := empty;
    loop:
      loop while t ≠ 0:
        S <= t;
        t := L[t];
      repeat;
    while nonempty(S):
      t <= S;
      print(A[t]);
      t := R[t];
    repeat;
  end.
```

Furthermore, there is a rather simple way to understand this program, by providing suitable "loop invariants". At the beginning of the first (outer) loop, suppose the stack contents from top to bottom are $t_n, \cdots, t_1$ for some $n \geq 0$; then the procedure's remaining duty is to accomplish the effect of

$$\begin{aligned}&\textit{treeprint}(t);\\ &\textit{print}(A[t_n]);\ \textit{treeprint}(R[t_n]);\\ &\cdots;\\ &\textit{print}(A[t_1]);\ \textit{treeprint}(R[t_1]);\end{aligned} \qquad (*)$$

In other words, the purpose of the stack is to record postponed obligations to print the A's and right subtrees of certain nodes. Once this concept is grasped, the meaning of the program is clear and we can even see how we might have written it without ever thinking of a recursive formulation or a **go to** statement: The innermost loop ensures $t = 0$, and afterwards the program reduces the stack, maintaining (*) as the condition to be fulfilled, at key points in the outer loop.

A careful programmer might notice a source of inefficiency in this program: when $L[t] = 0$, we put t on the stack, then take it off again. If there are n nodes in a binary tree, about half of them, on the average, will have $L[t] = 0$ so we might wish to avoid this extra computation. It isn't easy to do that to Example 6e without major surgery on the structure; but it *is* easy to modify Example 6c (or 6d), by simply bracketing the source of inefficiency, including the **go to**, and the label, and all.

Example 6f:

```
procedure treeprint(t); value t; integer t;
  begin integer stack S; S := empty;
L1: loop while t ≠ 0:
L3:   if L[t] ≠ 0
      then S <= t; t := L[t]; go to L1;
L2:        t <= S;
      fi;
      print(A[t]);
      t := R[t];
    repeat;
    if nonempty(S) then go to L2 fi;
  end.
```

Here we notice that a further simplification is possible: **go to** L1 can become **go to** L3 because t is known to be nonzero.

An equivalent **go to**-free program analogous to Example 6e is

Example 6g:

```
procedure treeprint(t); value t; integer t;
  begin integer stack S; S := empty;
    loop until finished:
      if t ≠ 0
      then
        loop while L[t] ≠ 0:
          S <= t;
          t := L[t];
        repeat;
      else
        if nonempty(S)
        then t <= S;
        else finished;
        fi;
      fi;
      print(A[t]);
      t := R[t];
    repeat;
  end.
```

I derived this program by thinking of the loop invariant (*) in Example 6e and acting accordingly, *not* by trying to eliminate the **go to**'s from Example 6f. So I know this program is well-structured, and I therefore haven't succeeded in finding an example of recursion removal where **go to**'s are strictly necessary. It is interesting, in fact, that our transformations originally intended for efficiency led us to new insights and to programs that still possess decent structure. However, I still feel that Example 6f is easier to understand than 6g, given that the reader is told the recursive program it comes from and the transformations that were used. The recursive program is trivially correct, and the transformations require only routine verification; by contrast, a mental leap is needed to invent (*).

Does recursion elimination help? Clearly there won't be much gain in this example if the *print* routine itself is the bottleneck. But let's replace *print*(A[*t*]) by

$$i := i+1;\ B[i] := A[t];$$

i.e., instead of printing the tree, let's assume that we merely want to transfer its contents to some other array B. Then we can expect to see an improvement.

After making this change, I tried the recursive Example 6 vs. the iterative Example 6d on the two main ALGOL compilers available to me. Normalizing the results so that 6d takes 1.0 units of time per node of the tree, with subscript checking suppressed, I found that the corresponding recursive version took about 2.1 units of time per node using our ALGOL W compiler for the 360/67; and the ratio was 1.16 using the SAIL compiler for the PDP-10. (Incidentally, the relative run-times for Example 6f were 0.8 with ALGOL W, and 0.7 with SAIL. When subscript ranges were dynamically checked, ALGOL W took 1.8 units of time per node for the nonrecursive version, and 2.8 with the recursive version; SAIL's figures were 1.28 and 1.34.)

Boolean Variable Elimination

Another important program transformation, somewhat less commonly known, is the removal of Boolean variables by code duplication. The following example is taken from Dijkstra's treatment [26, pp. 91–93] of Hoare's "Quicksort" algorithm. The idea is to rearrange array elements A[*m*]···A[*n*] so that they are partitioned into two parts: The left part A[*m*] ··· A[*j*−1], for some appropriate *j*, will contain all the elements less than some value, *v*; the right part A[*j*+1]··· A[*n*] will contain all the elements greater than *v*; and the element A[*j*] lying between these parts will be equal to *v*. Partitioning is done by scanning from the left until finding an element greater than *v*, then scanning from the right until finding an element less than *v*, then scanning from the left again, and so on, moving the offending elements to the opposite side, until the two scans come together; a Boolean variable *up* is used to distinguish the left scan from the right.

Example 7:

```
i := m; j := n;
v := A[j]; up := true;
loop:
  if up
  then if A[i] > v
    then A[j] := A[i]; up := false fi;
  else if v > A[j]
    then A[i] := A[j]; up := true fi;
  fi;
  if up then i := i+1 else j := j−1 fi;
while i < j repeat;
A[j] := v;
```

The manipulation and testing of *up* is rather time-consuming here. We can, in general, eliminate a Boolean variable by storing its current value in the program counter, i.e., by duplicating the program, letting one part of the text represent **true** and the other part **false**, with jumps between the two parts in appropriate places. Example 7 therefore becomes

Example 7a:

```
        i := m; j := n;
        v := A[j];
        loop: if A[i] > v
          then A[j] := A[i]; go to upf fi;
    upt: i := i+1;
        while i < j repeat; go to common;
        loop: if v > A[j]
        then A[i] := A[j]; go to upt fi;
    upf: j := j-1;
        while i < j repeat;
    common: A[j] := v;
```

Note that again we have come up with a program which has jumps into the middle of iterations, yet we can understand it since we know that it came from a previously understood program, by way of an understandable transformation.

Of course this program is messier than the first, and we must ask again if the gain in speed is worth this cost. If we are writing a sort procedure that will be used many times, we will be interested in the speed. The average running time of Quicksort was analyzed by Hoare in his 1962 paper on the subject [36], and it turns out that the body of the loop in Example 7 is performed about $2N \ln N$ times while the statement *up* := **false** is performed about $\frac{1}{3}N \ln N$ times, if we are sorting N elements. All other parts of the overall sorting program (not shown here) have a running time of order N or less, so when N is reasonably large the speed of the inner loop governs the speed of the entire sorting process. (Incidentally, a recursive version of Quicksort will run just about as fast, since the recursion overhead is not part of the inner loop. But in this case the removal of recursion is of great value for another reason, because it cuts the auxiliary stack space requirement from order N to order $\log N$.)

Using these facts about inner loop times, we can make a quantitative comparison of Examples 7 and 7a. As with Example 1, it seems best to make two comparisons, one with the assembly code that a decent programmer would write for the examples, and the other with the object code produced by a typical compiler that does only local optimizations. The assembly-language programmer will keep i, j, v, and *up* in registers, while a typical compiler will not keep variables in registers from one statement to another, except if they happen to be there by coincidence. Under these assumptions, the asymptotic running time for an entire Quicksort program based on these routines will be

	assembled	compiled
Example 7	$20\frac{2}{3}N \ln N$	$55\frac{1}{3}N \ln N$
Example 7a	$15\frac{1}{3}N \ln N$	$40N \ln N$

expressed in memory references to data and instructions. So Example 7a saves more than 25% of the sorting time.

I showed this example to Dijkstra, cautioning him that the **go to** leading into an iteration might be a terrible shock. I was extremely pleased to receive his reply [31]:

> Your technique of storing the value of *up* in the order counter is, of course, absolutely safe. I did not faint! I am in no sense "afraid" of a program constructed that way, but I cannot consider it beautiful: it is really the same repetition with the same terminating condition, that "changes color" as the computation proceeds.

He went on to say that he looks forward to the day when machines are so fast that we won't be under pressure to optimize our programs; yet

> For the time being I could not agree more with your closing remarks: if the economies matter, apply "disciplined optimalization" to a nice program, the correctness of which has been established beyond reasonable doubt. Your massaging of the program text is then no longer trickery ad hoc, it is perfectly safe and sound.

It is hard for me to express the joy that this letter gave me; it was like having all my sins forgiven, since I need no longer feel guilty about my optimized programs.

Coroutines

Several of the people who read the first draft of this paper observed that Example 7a can perhaps be understood more easily as the result of eliminating *coroutine* linkage instead of Boolean variables. Consider the following program:

Example 7b:

```
coroutine move i;
  loop: if A[i] > v
          then A[j] := A[i];
            resume move j;
          fi;
          i := i+1;
  while i < j repeat;
coroutine move j;
  loop: if v > A[j]
          then A[i] := A[j];
            resume move i;
          fi;
          j := j-1;
  while i < j repeat;
i := m; j := n; v := A[j];
call move i;
A[j] := v;
```

When a coroutine is "resumed", let's assume that it begins after its own **resume** statement; and when a coroutine terminates, let's assume that the most recent **call** statement is thereby completed. (Actual coroutine linkage is slightly more involved, see Chapter 3 of [17], but this description will suffice for our purposes.) Under these conventions, Example 7b is precisely equivalent to Example 7a. At the beginning of *move i* we know that $A[k] \leq v$ for all $k < i$, and that $i < j$, and that $\{A[m], \cdots, A[j-1], A[j+1], \cdots, A[n]\} \cup v$ is a permutation of the original contents of $\{A[m], \cdots, A[n]\}$; a similar statement holds at the beginning of *move j*. This separation into two coroutines can be said to make Example 7b conceptually simpler than Example 7; but on the other hand, the idea of coroutines admittedly takes some getting used to.

Christopher Strachey once told me about an example which first convinced him that coroutines provided an important control structure. Consider two binary trees represented as in Examples 5 and 6, with their A array information in increasing order as we traverse the trees in symmetric order of their nodes. The problem is to *merge* these two A array sequences into one ordered sequence. This requires traversing both trees more or less asynchronously, in symmetric order, so we'll need two versions of Example 6 running cooperatively. A conceptually simple solution to this problem can be written with coroutines, or by forming an equivalent program which expresses the coroutine linkage in terms of **go to** statements; it appears to be cumbersome (though not impossible) to do the job without using either feature.

Quicksort: A Digression

Dijkstra also sent another instructive example in his letter [30]. He decided to create the program of Example 7 from scratch, as if Hoare's algorithm had never been invented, starting instead with modern ideas of semi-automatic program construction based on the following *invariant* relation:

$$v = A[n] \land \\ \forall k(m \leq k < i \Rightarrow A[k] \leq v) \land \\ \forall k(j < k \leq n \Rightarrow A[k] \geq v).$$

The resulting program is unusual, yet perhaps cleaner than Example 7:

```
i := m; j := n-1; v := A[n];
loop while i ≤ j;
  if A[j] ≥ v then j := j-1;
  else A[i] :=: A[j]; i := i+1;
  fi;
repeat;
if j ≤ m then A[m] :=: A[n]; j := m fi;
```

Here ":=:" denotes the interchange (i.e., swap) operation. At the conclusion of this program, the A array will be different than before, but we will have the array partitioned as desired for sorting (i.e., $A[m] \cdots A[j]$ are $\leq v$ and $A[j+1] \cdots A[n]$ are $\geq v$).

Unfortunately, however, this "pure" program is less efficient than Example 7, and Dijkstra noted that he didn't like it very much himself. In fact, Quicksort is really quick in practice because there is a method that is even better than Example 7a: A good Quicksort routine will have a faster inner loop which avoids most of the "$i < j$" tests. Dijkstra recently [31] sent me another approach to the problem, which leads to a much better solution. First we can abstract the situation by considering any notions "small" and "large" so that: a) an element $A[i]$ is never both small and large simultaneously; b) some elements might be neither small nor large; c) we wish to rearrange an array so that all small elements precede all large ones; and d) there is at least one element which is not small, and at least one which is not large. Then we can write the following program in terms of this abstraction.

Example 8:

```
$i := m; j := n;$
loop:
  loop while A[$i$] is small:
    $i := i+1$; repeat;
  loop while A[$j$] is large:
    $j := j-1$; repeat;
while $i < j$:
  A[$i$] :=: A[$j$];
  $i := i+1; j := j-1$;
repeat;
```

At the beginning of the first (outer) loop we know that A[k] is not large for $m \le k < i$, and that A[k] is not small for $j < k \le n$; also that there exists a k such that $i \le k \le n$ and A[k] is not small, and a k such that $m \le k \le j$ and A[k] is not large. The operations in the loop are easily seen to preserve these "invariant" conditions. Note that the inner loops are now extremely fast, and that they are guaranteed to terminate; therefore the proof of correctness is simple. At the conclusion of the outer loop we know that A[m] $\cdots$ A[$i-1$] and A[j] are not large, that A[i] and A[$j+1$] $\cdots$ A[n] are not small, and that $m \le j \le i \le n$.

Applying this to Quicksort, we can set $v :=$ A[n] and write

"A[i] $< v$" in place of "A[i] is small"
"A[j] $> v$" in place of "A[j] is large"

in the above program. This gives a very pretty algorithm, which is essentially equivalent to the method published by Hoare [38] in his first major application of the idea of invariants, and discussed in his original paper on Quicksort [36]. Note that since $v =$ A[n], we know that the first execution of "**loop while** A[j] $> v$" will be trivial; we could move this loop to the end of the outer loop just before the final **repeat**. This would be slightly faster, but it would make the program harder to understand, so I would hesitate to do it.

The Quicksort partitioning algorithm actually given in my book [54] is better than Example 7a, but somewhat different from the program we have just derived. My version can be expressed as follows (assuming that A[$m-1$] is defined and $\le$A[n]):

```
$i := m-1; j := n; v :=$ A[$n$];
loop until pointers have met:
  loop: $i := i+1$; while A[$i$] $< v$ repeat;
  if $i \ge j$ then pointers have met; fi
  A[$j$] := A[$i$];
  loop: $j := j-1$; while A[$j$] $> v$ repeat;
  if $i \ge j$ then $j := i$; pointers have met; fi
  A[$i$] := A[$j$];
repeat;
A[$j$] := $v$;
```

At the conclusion of this routine, the contents of A[m] $\cdots$ A[n] have been permuted so that A[m] $\cdots$ A[$j-1$] are $\le v$ and A[$j+1$] $\cdots$ A[n] are $\ge v$ and A[j] $= v$ and $m \le j \le n$. The assembled version will make about $11N \ln N$ references to memory on the average, so this program saves 28% of the running time of Example 7a.

When I first saw Example 8 I was chagrined to note that it was easier to prove than my program, it was shorter, and (the crushing blow) it also seemed about 3% faster, because it tested "$i < j$" only half as often. My first mathematical analysis of the average behavior of Example 8 indicated that the asymptotic number of comparisons and exchanges would be the same, even though the partitioned subfiles included all N elements instead of $N-1$ as in the classical Quicksort routine. But suddenly it occurred to me that my new analysis was incorrect because one of its fundamental assumptions breaks down: the elements of the two subfiles after partitioning by Example 8 are not in random order! This was a surprise, because randomness *is* preserved by the usual Quicksort routine. When the N keys are distinct, v will be the largest element in the left subfile, and the mechanism of Example 8 shows that v will tend to be near the left of that subfile. When that subfile is later partitioned, it is highly likely that v will move to the extreme right of the resulting right sub-subfile. So that right sub-subfile will be subject to a trivial partitioning by its largest element; we have a subtle loss of efficiency on the third level of recursion. I still haven't been able to analyze Example 8, but empirical tests have borne out my prediction that it is in fact about 15% slower than the book algorithm.

Therefore, there is no reason for anybody to use Example 8 in a sorting routine;

though it is slightly cleaner looking than the method in my book, it is noticeably slower, and we have nothing to fear by using a slightly more complicated method once it has been proved correct. Beautiful algorithms are, unfortunately, not always the most useful.

This is not the end of the Quicksort story (although I almost wish it was, since I think the preceding paragraph makes an important point). After I had shown Example 8 to my student, Robert Sedgewick, he found a way to modify it, preserving the randomness of the subfiles, thereby achieving both elegance and efficiency at the same time. Here is his revised program.

Example 8a:

```
$i := m-1$; $j := n$; $v := A[n]$;
loop:
  loop: $i := i+1$; while $A[i] < v$ repeat;
  loop: $j := j-1$; while $A[j] > v$ repeat;
while $i < j$:
  $A[i] :=: A[j]$;
repeat;
$A[i] :=: A[n]$;
```

(As in the previous example, we assume that $A[m-1]$ is defined and $\leq A[n]$, since the j pointer might run off the left end.) At the beginning of the outer loop the invariant conditions are now

$$\begin{aligned}&m-1 \leq i < j \leq n;\\ &A[k] \leq v \text{ for } m-1 \leq k \leq i;\\ &A[k] \geq v \text{ for } j \leq k \leq n;\\ &A[n] = v.\end{aligned}$$

It follows that Example 8a ends with

$$A[m]\cdots A[i-1] \leq v = A[i] \leq A[i+1]\cdots A[n]$$

and $m \leq i \leq n$; hence a valid partition has been achieved.

Sedgewick also found a way to improve the inner loop of the algorithm from my book, namely:

```
$i := m-1$; $j := n$; $v := A[n]$;
loop:
  loop: $i := i+1$; while $A[i] < v$ repeat;
  $A[j] := A[i]$;
  loop: $j := j-1$; while $A[j] > v$ repeat;
while $i < j$:
  $A[i] := A[j]$;
repeat;
if $i \neq j$ then $j := j+1$;
$A[j] := v$;
```

Each of these programs leads to a Quicksort routine that makes about $10\frac{2}{3}N \ln N$ memory references on the average; the former is preferable (except on machines for which exchanges are clumsy), since it is easier to understand. Thus I learned again that I should always keep looking for improvements, even when I have a satisfactory program.

Axiomatics of Jumps

We have now discussed many different transformations on programs; and there are more which could have been mentioned (e.g., the removal of trivial assignments as in [50, exercise 1.1-3] or [54, exercise 5.2.1-33]). This should be enough to establish that a program-manipulation system will have plenty to do.

Some of these transformations introduce **go to** statements that cannot be handled very nicely by event indicators, and in general we might expect to find a few programs in which **go to** statements survive. Is it really a formidable job to understand such programs? Fortunately this is not an insurmountable task, as recent work has shown. For many years, the **go to** statement has been troublesome in the definition of correctness proofs and language semantics; for example, Hoare and Wirth have presented an axiomatic definition of PASCAL [41] in which everything but **real** arithmetic and the **go to** is defined formally. Clint and Hoare [14] have shown how to extend this to event-indicator **go to**'s (i.e., those which don't lead into iterations or conditionals), but they stressed that the general case appears to be fraught with complications. Just recently, however, Hoare has shown that there is, in fact, a rather simple way to give an axiomatic definition of **go to** statements; indeed, he wishes quite frankly that it hadn't been quite so simple. For each label L in a program, the programmer should state a logical assertion $\alpha(L)$ which is to be true whenever we reach L. Then the axioms

$$\{\alpha(L)\}\ \textbf{go to}\ L\ \{\textbf{false}\}$$

plus the rules of inference

$$\{\alpha(L)\}\ S\{P\} \vdash \{\alpha(L)\}\ L{:}S\ \{P\}$$

are allowed in program proofs, and all properties of labels and **go to**'s will follow if the $\alpha(L)$ are selected intelligently. One must, of course, carry out the entire proof using the same assertion $\alpha(L)$ for each appearance of the label L, and some choices of assertions will lead to more powerful results than others.

Informally, $\alpha(L)$ represents the desired state of affairs at label L; this definition says essentially that a program is correct if $\alpha(L)$ holds at L and before all "**go to** L" statements, and that control never "falls through" a **go to** statement to the following text. Stating the assertions $\alpha(L)$ is analogous to formulating loop invariants. Thus, it is not difficult to deal formally with tortuous program structure if it turns out to be necessary; all we need to know is the "meaning" of each label.

Reduction of Complication

There is one remaining use of **go to** for which I have never seen a good replacement, and in fact it's a situation where I still think **go to** is the right idea. This situation typically occurs after a program has made a multiway branch to a rather large number of different but related cases. A little computation often suffices to reduce one case to another; and when we've reduced one problem to a simpler one, the most natural thing is for our program to **go to** the routine which solves the simpler problem.

For example, consider writing an interpretive routine (e.g., a microprogrammed emulator), or a simulator of another computer. After decoding the address and fetching the operand from memory, we do a multiway branch based on the operation code. Let's say the operations include no-op, add, subtract, jump on overflow, and unconditional jump. Then the subtract routine might be

```
operand := − operand; go to add;
```

the add routine might be

```
accum := accum + operand;
tyme := tyme + 1;
go to no op;
```

and jump on overflow might be

```
if overflow
then overflow := false; go to jump;
else go to no op;
fi;
```

I still believe that this is the correct way to write such a program.

Such situations aren't restricted to interpreters and simulators, although the foregoing is a particularly dramatic example. Multiway branching is an important programming technique which is all too often replaced by an inefficient sequence of **if** tests. Peter Naur recently wrote me that he considers the use of tables to control program flow as a basic idea of computer science that has been nearly forgotten; but he expects it will be ripe for rediscovery any day now. It is the key to efficiency in all the best compilers I have studied.

Some hints of this situation, where one problem reduces to another, have occurred in previous examples of this paper. Thus, after searching for x and discovering that it is absent, the "not found" routine can insert x into the table, thereby reducing the problem to the "found" case. Consider also our decision-table Example 4, and suppose that each period was to be followed by a carriage return instead of by an extra space. Then it would be natural to reduce the post-processing of periods to the return-carriage part of the program. In each case, a **go to** would be easy to understand.

If we need to find a way to do this without saying **go to**, we could extend Zahn's event indicator scheme so that some events are allowed to happen in the **then** $\cdots$ **fi** part after we have begun to process other events. This accommodates the above-mentioned examples very nicely; but of course it can be dangerous when misused, since it gives us back all the power of **go to**. A restriction which allows $\langle\text{statement list}\rangle_i$ to refer to $\langle\text{event}\rangle_j$ only for $j > i$ would be less dangerous.

With such a language feature, we can't "fall through" a label (i.e., an event indicator) when the end of the preceding code is reached; we must explicitly name each event when we go to its routine. Prohibiting "fall through" means forcing a programmer to write "**go to** common" just before the label "common:" in Example 7a; surprisingly, such a change actually makes that program more readable, since it makes the symmetry plain. Also, the program fragment

```
subtract: operand := − operand; go to add;
add:      accum := accum + operand;
```

seems to be more readable than if "**go to** add" were deleted. It is interesting to ponder why this is so.

3. CONCLUSIONS

This has been a long discussion, and very detailed, but a few points stand out. First, there are several kinds of programming situations in which **go to** statements are harmless, even desirable, if we are programming in ALGOL or PL/I. But secondly, new types of syntax are being developed that provide good substitutes for these harmless **go to**'s, and without encouraging a programmer to create "logical spaghetti".

One thing we haven't spelled out clearly, however, is what makes some **go to**'s bad and others acceptable. The reason is that we've really been directing our attention to the wrong issue, to the objective question of **go to** elimination instead of the important subjective question of program structure. In the words of John Brown [9], "The act of focusing our mightiest intellectual resources on the elusive goal of **go to**-less programs has helped us get our minds off all those really tough and possibly unresolvable problems and issues with which today's professional programmer would otherwise have to grapple." By writing this long article I don't want to add fuel to the controversy about **go to** elimination, since that topic has already assumed entirely too much significance; my goal is to lay that controversy to rest, and to help direct the discussion towards more fruitful channels.

Structured Programming

The real issue is structured programming, but unfortunately this has become a catch phrase whose meaning is rarely understood in the same way by different people. Everybody knows it is a Good Thing, but as McCracken [64] has said, "Few people would venture a definition. In fact, it is not clear that there exists a simple definition as yet." Only one thing is really clear: Structured programming is *not* the process of writing programs and then eliminating their **go to** statements. We should be able to define structured programming without referring to **go to** statements at all; then the fact that **go to** statements rarely need to be introduced as we write programs should follow as a corollary.

Indeed, Dijkstra's original article [25] which gave Structured Programming its name never mentions **go to** statements at all; he directed attention to the critical question, "For what program structures can we give correctness proofs without undue labor, even if the programs get large?" By correctness proofs he explained that he does not mean formal derivations from axioms, he means any sort of proof (formal or informal) that is "sufficiently convincing"; and a proof really means an understanding. By program structure he means data structure as well as control structure.

We understand complex things by systematically breaking them into successively simpler parts and understanding how these parts fit together locally. Thus, we have different levels of understanding, and each of these levels corresponds to an *abstraction* of the detail at the level it is composed from. For example, at one level of abstraction, we deal with an integer without considering whether it is represented in binary notation or two's complement, etc., while at deeper levels this representation may be important. At more abstract levels the precise value of the integer is not important except as it relates to other data.

Charles L. Baker mentioned this principle as early as 1957, as part of his 8-page review [2] of McCracken's first book on programming:

Break the problem into small, self-contained subroutines, trying at all times to isolate the various sections of coding as much as possible . . . [then] the problem is reduced to many much smaller ones. The truth of this seems very obvious to experienced coders, yet it is hard to put across to the newcomer.

Abstraction is easily understood in terms of BNF notation. A metalinguistic category like ⟨assignment statement⟩ is an abstraction which is composed of two abstractions (a ⟨left part list⟩ and an ⟨arithmetic expression⟩), each of which is composed of abstractions such as ⟨identifier⟩ or ⟨term⟩, etc. We understand the program syntax as a whole by knowing the structural details that relate these abstract parts. The most difficult things to understand about a program's syntax are the identifiers, since their meaning is passed across several levels of structure. If all identifiers of an ALGOL program were changed to random meaningless strings of symbols, we would have great difficulty seeing what the type of a variable is and what the program means, but we would still easily recognize the more local features, such as assignment statements, expressions, subscripts, etc. (This inability for our eyes to associate a type or mode with an identifier has led to what I believe are fundamental errors of human engineering in the design of ALGOL 68, but that's another story. My own notation for stacks in Example 6c suffers from the same problem; it works in these examples chiefly because t is lower case and S is upper case.) Larger nested structures are harder for the eye to see unless they are indented, but indentation makes the structure plain.

It would probably be still better if we changed our source language concept so that the program wouldn't appear as one long string. John McCarthy says "I find it difficult to believe that whenever I see a tree I am really seeing a string of symbols." Instead, we should give meaningful names to the larger constructs in our program that correspond to meaningful levels of abstraction, and we should define those levels of abstraction in one place, and merely use their names (instead of including the detailed code) when they are used to build larger concepts. Procedure names do this, but the language could easily be designed so that no action of calling a subroutine is implied.

From these remarks it is clear that sequential composition, iteration, and conditional statements present syntactic structures that the eye can readily assimilate; but a **go to** statement does not. The visual structure of **go to** statements is like that of flowcharts, except reduced to *one* dimension in our source languages. In two dimensions it is possible to perceive **go to** structure in small examples, but we rapidly lose our ability to understand larger and larger flowcharts; some intermediate levels of abstraction are necessary. As an undergraduate, in 1959, I published an octopus flowchart which I sincerely hope is the most horribly complicated that will ever appear in print; anyone who believes that flowcharts are the best way to understand a program is urged to look at this example [49]. (See also [32, p. 54] for a nice illustration of how **go to**'s make a PL/I program obscure, and see R. Lawrence Clark's hilarious spoof about linear representation of flowcharts by means of a "**come from** statement" [13].)

I have felt for a long time that a talent for programming consists largely of the ability to switch readily from microscopic to macroscopic views of things, i.e., to change levels of abstraction fluently. I mentioned this [55] to Dijkstra, and he replied [29] with an excellent analysis of the situation:

> I feel somewhat guilty when I have suggested that the distinction or introduction of "different levels of abstraction" allow you to think about only one level at a time, ignoring completely the other levels. This is not true. You are trying to organize your thoughts; that is, you are seeking to arrange matters in such a way that you can concentrate on some portion, say with 90% of your conscious thinking, while the rest is temporarily moved away somewhat towards the background of your mind. But that is something quite different from "ignoring completely": you allow yourself temporarily to ignore details, but some overall appreciation of what is supposed to be or to come there continues to play a vital role. You remain alert for little red lamps that suddenly start flickering in the corners of your eye.

I asked Hoare for a short definition of structured programming, and he replied that it is "the systematic use of abstraction to control a mass of detail, and also a means of documentation which aids program design."

I hope that my remarks above have made the abstract concept of abstraction clear; the second part of Hoare's definition (which was also stressed by Dijkstra in his original paper [25]) states that a good way to express the abstract properties of an unwritten piece of program often helps us to write that program, and to "know" that it is correct as we write it.

Syntactic structure is just one part of the picture, and BNF would be worthless if the syntactic constructs did not correspond to semantic abstractions. Similarly, a good program will be composed in such a way that each semantic level of abstraction has a reasonably simple relation to its constituent parts. We noticed in our discussion of Jacopini's theorem that every program can trivially be expressed in terms of a simple iteration which simulates a computer; but that iteration has to carry the entire behavior of the program through the loop, so it is worthless as a level of abstraction.

An iteration statement should have a purpose that is reasonably easy to state; typically, this purpose is to make a certain Boolean relation true while maintaining a certain invariant condition satisfied by the variables. The Boolean condition is stated in the program, while the invariant should be stated in a comment, unless it is easily supplied by the reader. For example, the invariant in Example 1 is that $A[k] \neq x$ for $1 \leq k < i$, and in Example 2 it is the same, plus the additional relation $A[m+1] = x$. Both of these are so obvious that I didn't bother to mention them; but in Examples 6e and 8, I stated the more complicated invariants that arose. In each of those cases the program almost wrote itself once the proper invariant was given. Note that an "invariant assertion" actually does vary slightly as we execute statements of the loop, but it comes back to its original form when we repeat the loop.

Thus, an iteration makes a good abstraction if we can assign a meaningful invariant describing the local states of affairs as it executes, and if we can describe its purpose (e.g., to change one state to another). Similarly, an **if** ··· **then** ··· **else** ··· **fi** statement will be a good abstraction if we can state an overall purpose for the statement as a whole.

We also need well-structured *data;* i.e., as we write the program we should have an abstract idea of what each variable means. This idea is also usually describable as an invariant relation, e.g., "m is the number of items in the table" or "x is the search argument" or "$L[t]$ is the number of the root node of node t's left subtree, or 0 if this subtree is empty" or "the contents of stack S are postponed obligations to do such and such".

Now let's consider the slightly more complex case of an event-driven construct. This should also correspond to a meaningful abstraction, and our examples show what is involved: For each event we give an (invariant) assertion which describes the situation which must hold when that event occurs, and for the **loop until** we also give an invariant for the loop. An event statement typically corresponds to an abrupt change in conditions so that a different assertion from the loop invariant is necessary.

An error exit can be considered well-structured for precisely this reason—it corresponds to a situation that is impossible according to the local invariant assertions; it is easiest to formulate assertions that assume nothing will go wrong, rather than to make the invariants cover all contingencies. When we jump out to an error exit we go to another level of abstraction having different assumptions.

As another simple example, consider binary search in an ordered array using the invariant relation $A[i] < x < A[j]$:

```
loop while i+1 < j;
  k := (i+j) ÷ 2;
  if A[k] < x then i := k;
  else if A[k] > x then j := k;
    else cannot preserve the invariant fi;
  fi;
repeat;
```

Upon normal exit from this loop, the conditions $i+1 \geq j$ and $A[i] < x < A[j]$ imply that $A[i] < x < A[i+1]$, i.e., that x is not present. If the program comes to "cannot preserve the invariant" (because $x = A[k]$), it wants to **go to** another set of assumptions. The event-driven construct provides a level at which it is appropriate to specify the other assumptions.

Another good illustration occurs in Example 6g; the purpose of the main **if** statement is to find the first node whose A value should be printed. If there is no such *t*, the event "finished" has clearly occurred; it is better to regard the **if** statement as having the stated abstract purpose without considering that *t* might not exist.

With go to Statements

We can also consider **go to** statements from the same point of view; when do they correspond to a good abstraction? We've already mentioned that **go to**'s do not have a syntactic structure that the eye can grasp automatically; but in this respect they are no worse off than variables and other identifiers. When these are given a meaningful name corresponding to the abstraction (N.B. *not* a numeric label!), we need not apologize for the lack of syntactic structure. And the appropriate abstraction itself is an invariant essentially like the assertions specified for an event.

In other words, we can indeed consider **go to** statements as part of systematic abstraction; all we need is a clearcut notion of exactly what it means to **go to** each label. This should come as no great surprise. After all, a lot of computer programs have been written using **go to** statements during the last 25 years, and these programs haven't all been failures! Some programmers have clearly been able to master structure and exploit it; not as consistently, perhaps, as in modern-day structured programming, but not inflexibly either. By now, many people who have never had any special difficulty writing correct programs have naturally been somewhat upset after being branded as sinners, especially when they know perfectly well what they're doing; so they have understandably been less than enthusiastic about "structured programming" as it has been advertised to them.

My feeling is that it's certainly possible to write well-structured programs with **go to** statements. For example, Dijkstra's 1965 program about concurrent process control [24] used three **go to** statements, all of which were perfectly easy to understand; and I think at most two of these would have disappeared from his code if ALGOL 60 had had a **while** statement. But **go to** is hardly ever the best alternative now, since better language features are appearing. If the invariant for a label is closely related to another invariant, we can usually save complexity by combining those two into one abstraction, using something other than **go to** for the combination.

There is also another problem, namely at what level of abstraction should we introduce a label? This however is like the analogous problem for variables, and the general answer is still unclear in both cases. Aspects of data structure are often postponed, but sometimes variables are defined and passed as "parameters" to other levels of abstraction. There seems to be no clearcut idea as yet about a set of syntax conventions, relating to the definition of variables, which would be most appropriate to structured programming methodology; but for each particular problem there seems to be an appropriate level.

Efficiency

In our previous discussion we concluded that premature emphasis on efficiency is a big mistake which may well be the source of most programming complexity and grief. We should ordinarily keep efficiency considerations in the background when we formulate our programs. We need to be subconsciously aware of the data processing tools available to us, but we should strive most of all for a program that is easy to understand and almost sure to work. (Most programs are probably only run once; and I suppose in such cases we needn't be too fussy about even the structure, much less the efficiency, as long as we are happy with the answers.)

When efficiencies do matter, however, the good news is that usually only a very small fraction of the code is significantly involved. And when it is desirable to sacrifice clarity for efficiency, we have seen that it *is* possible to produce reliable programs that can be maintained over a period of time, if we start with a well-structured program and then use well-understood transformations that can be applied mechanically. We shouldn't attempt to understand the resulting program as it appears in its final form; it should be thought of as the result of the original program modified by specified transformations. We can envision program manipulation systems which will facilitate making and documenting these transformations.

In this regard I would like to quote some observations made recently by Pierre-Arnoul de Marneffe [19]:

> In civil engineering design, it is presently a mandatory concept known as the "Shanley Design Criterion" to collect several functions into one part . . . If you make a cross-section of, for instance, the German V-2, you find external skin, structural rods, tank wall, etc. If you cut across the Saturn-B moon rocket, you find only an external skin which is at the same time a structural component and the tank wall. Rocketry engineers have used the "Shanley Principle" thoroughly when they use the fuel pressure inside the tank to improve the rigidity of the external skin! . . . People can argue that structured programs, even if they work correctly, will look like laboratory prototypes where you can discern all the individual components, but which are not daily usable. Building "integrated" products is an engineering principle as valuable as structuring the design process.

He goes on to describe plans for a prototype system that will automatically assemble integrated programs from well-structured ones that have been written top-down by stepwise refinement.

Today's hardware designers certainly know the advantages of integrated circuitry, but of course they must first understand the separate circuits before the integration is done. The V-2 rocket would never have been airborne if its designers had originally tried to combine all its functions. Engineering has two phases, structuring and integration; we ought not to forget either one, but it is best to hold off the integration phase until a well-structured prototype is working and understood. As stated by Weinberg [93], the former regimen of analysis/coding/debugging should be replaced by analysis/coding/debugging/improving.

The Future

It seems clear that languages somewhat different from those in existence today would enhance the preparation of structured programs. We will perhaps eventually be writing only small modules which are identified by name as they are used to build larger ones, so that devices like indentation, rather than delimiters, might become feasible for expressing local structure in the source language. (See the discussion following Landin's paper [59].) Although our examples don't indicate this, it turns out that a given level of abstraction often involves several related routines and data definitions; for example, when we decide to represent a table in a certain way, we simultaneously want to specify the routines for storing and fetching information from that table. The next generation of languages will probably take into account such related routines.

Program manipulation systems appear to be a promising future tool which will help programmers to improve their programs, and to enjoy doing it. Standard operating procedure nowadays is usually to hand code critical portions of a routine in assembly language. Let us hope such assemblers will die out, and we will see several levels of language instead: At the highest levels we will be able to write abstract programs, while at the lowest levels we will be able to control storage and register allocation, and to suppress subscript range checking, etc. With an integrated system it will be possible to do debugging and analysis of the transformed program using a higher level language for communication. All levels will, of course, exhibit program structure syntactically so that our eyes can grasp it.

I guess the big question, although it really shouldn't be so big, is whether or not the ultimate language will have **go to** statements in its higher levels, or whether **go to** will be confined to lower levels. I personally wouldn't mind having **go to** in the highest level, just in case I really need it; but I probably would never use it, if the general iteration and event constructs suggested in this paper were present. As soon as people learn to apply principles of abstraction consciously, they won't see the need for **go to**, and the issue will just fade away. On the other hand, W. W. Peterson told me about his experience teaching PL/I to beginning programmers: He taught them to use **go to** only in unusual special cases where **if** and **while** aren't right, but he found [78] that "A disturbingly large percentage of the students ran into situations that require **go to**'s, and sure enough, it was often because **while** didn't work well to their plan, but almost invariably because their plan was poorly thought out." Because of arguments like this, I'd say we should, indeed, abolish **go to** from the high-level language, at least as an experiment in training people to formulate their abstractions more carefully. This does have a beneficial effect on style, although I would not make such a prohibition if the new language features described above were not available. The question is whether we should ban it, or educate against it; should we attempt to legislate program morality? In this case I vote for legislation, with appropriate legal substitutes in place of the former overwhelming temptations.

A great deal of research must be done if we're going to have the desired language by 1984. Control structure is merely one simple issue, compared to questions of abstract data structure. It will be a major problem to keep the total number of language features within tight limits. And we must especially look at problems of input/output and data formatting, in order to provide a viable alternative to COBOL.

ACKNOWLEDGMENTS

I've benefited from a truly extraordinary amount of help while preparing this paper. The individuals named provided me with a total of 144 pages of single-spaced comments, plus six hours of conversation, and four computer listings:

Frances E. Allen	Ralph L. London
Forest Baskett	Zohar Manna
G. V. Bochmann	W. M. McKeeman
Per Brinch Hansen	Harlan D. Mills
R. M. Burstall	Peter Naur
Vinton Cerf	Kjell Overholt
T. E. Cheatham, Jr.	James Peterson
John Cocke	W. Wesley Peterson
Ole-Johan Dahl	Mark Rain
Peter J. Denning	John Reynolds
Edsger Dijkstra	Barry K. Rosen
James Eve	E. Satterthwaite, Jr.
K. Friedenbach	D. V. Schorre
Donald I. Good	Jacob T. Schwartz
Ralph E. Gorin	Richard L. Sites
Leo Guibas	Richard Sweet
C. A. R. Hoare	Robert D. Tennent
Martin Hopkins	Niklaus Wirth
James J. Horning	M. Woodger
B. M. Leavenworth	William A. Wulf
Henry F. Ledgard	Charles T. Zahn

These people unselfishly devoted hundreds of man-hours to helping me revise the first draft; and I'm sorry that I wasn't able to reconcile all of their interesting points of view. In many places I have shamelessly used their suggestions without an explicit acknowledgment; this article is virtually a joint paper with 30 to 40 co-authors! However, any mistakes it contains are my own.

APPENDIX

In order to make some quantitative estimates of efficiency, I have counted memory references for data and instructions, assuming a multiregister computer without cache memory. Thus, each instruction costs one unit, plus another if it refers to memory; small constants and base addresses are assumed to be either part of the instruction or present in a register. Here are the code sequences developed for the first two examples, assuming that a typical assembly-language programmer or a very good optimizing compiler is at work.

LABEL	INSTRUCTION	COST	TIMES
Example 1:	$r1 \leftarrow 1$	1	1
	$r2 \leftarrow m$	2	1
	$r3 \leftarrow x$	2	1
	to test	1	1
loop:	A[$r1$]: $r3$	2	$n-a$
	to found **if** =	1	$n-a$
	$r1 \leftarrow r1+1$	1	$n-1$
test:	$r1$: $r2$	1	n
	to loop **if** $\leq$	1	n
notfound:	$m \leftarrow r1$	2	a
	A[$r1$] $\leftarrow r3$	2	a
	B[$r1$] $\leftarrow 0$	2	a
found:	$r4 \leftarrow$ B[$r1$]	2	1
	$r4 \leftarrow r4+1$	1	1
	B[$r1$] $\leftarrow r4$	2	1

LABEL	INSTRUCTION	COST	TIMES
Example 2:	$r2 \leftarrow m$	2	1
	$r3 \leftarrow x$	2	1
	A[$r2+1$] $\leftarrow r3$	2	1
	$r1 \leftarrow 0$	1	1
loop:	$r1 \leftarrow r1+1$	1	n
	A[$r1$]: $r3$	2	n
	to loop **if** $\neq$	1	n
	$r1$: $r2$	1	1
	to found if $\leq$	1	1
notfound:	$m \leftarrow r1$	etc. as in Example 1.	

A traditional "90% efficient compiler" would render the first example as follows:

LABEL	INSTRUCTION	COST	TIMES
Example 1:	$r1 \leftarrow 1$	1	1
	to test	1	1
incr:	$r1 \leftarrow i$	2	$n-1$
	$r1 \leftarrow r1+1$	1	$n-1$
test:	$r1: m$	2	n
	to notfound **if** >	1	n
	$i \leftarrow r1$	2	$n-a$
	$r2 \leftarrow A[r1]$	2	$n-a$
	$r2: x$	2	$n-a$
	to found **if** =	1	$n-a$
	to incr	1	$n-1$
notfound:	$r1 \leftarrow m$	2	a
	$r1 \leftarrow r1 + 1$	1	a
	$i \leftarrow r1$	2	a
	$m \leftarrow r1$	2	a
	$r1 \leftarrow x$	2	a
	$r2 \leftarrow i$	2	a
	$A[r2] \leftarrow r1$	2	a
	$B[r2] \leftarrow 0$	2	a
found:	$r1 \leftarrow i$	2	1
	$r2 \leftarrow B[r1]$	2	1
	$r2 \leftarrow r2+1$	1	1
	$B[r1] \leftarrow r2$	2	1

Answer to PL/I Problem, page 267.

The variable I is increased before FOUND is tested. One way to fix the program is to insert "I = I − FOUND;" before the last statement.

BIBLIOGRAPHY

[1] ASHCROFT, EDWARD, AND MANNA, ZOHAR. "The translation of 'go to' programs to 'while' programs," *Proc. IFIP Congress 1971* Vol. 1, North-Holland Publ. Co., Amsterdam, The Netherlands, 1972, 250–255.

[2] BAKER, CHARLES L. "Review of D. D. McCracken, *Digital computer programming*," *Math. Comput.* 11 (1957), 298–305.

[3] BAKER, F. TERRY, AND MILLS, HARLAN D. "Chief programmer teams," *Datamation* 19, 12 (December 1973), 58–61.

[4] BARRON, D. W. *Recursive techniques in programming*, American Elsevier, New York, 1968, 64 pp.

[5] BAUER, F. L. "A philosophy of programming," University of London Special Lectures in Computer Science (October 1973): Lecture notes published by Math. Inst., Tech. Univ. of Munich, Germany.

[6] BERRY, DANIEL M. "Loops with normal and abnormal exits," *Modeling and Measurement Note 28*, Computer Science Department, Univ. California, Los Angeles, Calif. 1974, 39 pp.

[7] BOCHMANN, G. V. "Multiple exits from a loop without the GOTO," *Comm. ACM* **16**, 7 (July 1973), 443–444.

[8] BÖHM, CORRADO AND JACOPINI, GUISEPPE. "Flow-diagrams, Turing machines, and languages with only two formation rules," *Comm. ACM* **9**, 5 (May 1966), 366–371.

[9] Brown, John R. "In memoriam . . .", unpublished note, January 1974.

[10] Bruno J., and Stieglitz, K. "The expression of algorithms by charts," *J. ACM* **19**, 3 (July 1972), 517–525.

[11] Burkhard, W. A. "Nonrecursive tree traversal algorithms," in *Proc. 7th Annual Princeton Conf. on Information Sciences and Systems*, Princeton Univ. Press, Princeton, N.J., 1973, 403–405.

[12] Cheatham, T. E., Jr., and Wegbreit, Ben. "A laboratory for the study of automating programming," in *Proc. AFIPS 1972 Spring Joint Computer Conf.*, Vol. **40**, AFIPS Press, Montvale, N.J., 1972, 11–21.

[13] Clark, R. Lawrence. "A linguistic contribution to GOTO-less programming," *Datamation* **19**, 12 (December 1973), 62–63.

[14] Clint, M., and Hoare, C. A. R. "Program proving: jumps and functions," *Acta Informatica* **1**, 3 (1972), 214–224.

[15] Cooper, D. C. "The equivalence of certain computations," *Computer J.* **9**, 1 (May 1966), 45–52.

[16] Cooper, D. C. "Böhm and Jacopini's reduction of flow charts," *Comm. ACM* **10**, 8 (August 1967), 463, 473.

[17] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R. *Structured programming*, Academic Press, London, England, 1972, 220 pp.

[18] Darlington, J., and Burstall, R. M. "A system which automatically improves programs," in *Proc. 3rd Interntl. Conf. on Artificial Intelligence*, Stanford Univ., Stanford, Calif., 1973, 479–485.

[19] de Marneffe, Pierre-Arnoul. "Holon programming: A survey," Universite de Liege, Service Informatique, Liege, Belgium, 1973, 135 pp.

[20] Dijkstra, E. W. "Recursive programming," *Numerische Mathematik* **2**, 5 (1960), 312–318.

[21] Dijkstra, E. W. "Programming considered as a human activity," in *Proc. IFIP Congress 1965*, North-Holland Publ. Co., Amsterdam, The Netherlands, 1965, 213–217.

[22] Dijkstra, E. W. "A constructive approach to the problem of program correctness," *BIT* **8**, 3 (1968), 174–186.

[23] Dijkstra, E. W. "Go to statement considered harmful," *Comm. ACM* **11**, 3 (March 1968), 147–148, 538, 541.
[There are two instances of pages 147–148 in this volume; the *second* 147–148 is relevant here.]

[24] Dijkstra, E. W. "Solution of a problem in concurrent programming control," *Comm. ACM* **9**, 9 (September 1968), 569.

[25] Dijkstra, E. W. "Structured programming," in *Software engineering techniques*, J. N. Buxton and B. Randell [Eds.] NATO Scientific Affairs Division, Brussels, Belgium, 1970, 84–88.

[26] Dijkstra, E. W. "EWD316: A short introduction to the art of programming," Technical University Eindhoven, The Netherlands, August 1971, 97 pp.

[27] Dijkstra, E. W. "The humble programmer," *Comm. ACM* **15**, 10 (October 1972), 859–866.

[28] Dijkstra, E. W. "Prospects for a better programming language," in *High level languages*, C. Boon [Ed]., Infotech State of the Art Report 7, 1972, 217–232.

[29] Dijkstra, E. W. personal communication, January 3, 1973.

[30] Dijkstra, E. W. personal communication, November 19, 1973.

[31] Dijkstra, E. W. personal communication, January 30, 1974.

[32] Donaldson, James R. "Structured programming," *Datamation* **19**, 12 (December 1973), 52–54.

[33] Dylan, Bob. *Blonde on blonde*, record album produced by Bob Johnston, Columbia Records, New York, March 1966, Columbia C2S 841.

[34] Gill, Stanley. "Automatic computing: Its problems and prizes," *Computer J.* **8**, 3 (October 1965), 177–189.

[35] Henderson, P. and Snowdon, R. "An experiment in structured programming," *BIT* **12**, 1 (1972), 38–53.

[36] Hoare, C. A. R. "Quicksort," *Computer J.* **5**, 1 (1962), 10–15.

[37] Hoare, C. A. R. "An axiomatic approach to computer programming," *Comm. ACM* **12**, 10 (October 1969), 576–580, 583.

[38] Hoare, C. A. R. "Proof of a program: FIND," *Comm. ACM* **14**, 1 (January 1971), 39–45.

[39] Hoare, C. A. R. "A note on the for statement," *BIT* **12**, 3 (1972), 334–341.

[40] Hoare, C. A. R. "Prospects for a better programming language," in *High level languages*, C. Boon [Ed.], Infotech State of the Art Report 7, 1972, 327–343.

[41] Hoare, C. A. R., and Wirth, N. "An axiomatic definition of the programming language PASCAL," *Acta Informatica* **2**, 4 (1973), 335–355.

[42] Hoare, C. A. R. "Hints for programming language design," Computer Science report STAN-CS-74-403, Stanford Univ., Stanford, Calif., January 1974, 29 pp.

[43] Hopkins, Martin E. "Computer aided software design," in *Software engineering techniques*, J. N. Buxton and B. Randell [Eds.] NATO Scientific Affairs Division, Brussels, Belgium, 1970, 99–101.

[44] Hopkins, Martin E. "A case for the GOTO," *Proc. ACM Annual Conference* Boston, Mass., August 1972, 787–790.

[45] Hull, T. E. "Would you believe structured FORTRAN?" *SIGNUM Newsletter* **8**, 4 (October 1973), 13–16.

[46] Ingalls, Dan. "The execution time profile as a programming tool," in *Compiler optimization*, 2d Courant Computer Science Symposium, Randall Rustin [Ed.], Prentice-Hall, Englewood Cliffs, N. J., 1972, 107–128.

[47] Kelley, Robert A., and Walters, John R. "APLGOL-2, a structured programming system for APL," IBM Palo Alto Scientific Center report 320–3318 (August 1973), 29 pp.

[48] Kleene, S. C. "Representation of events in nerve nets," in *Automata Studies*, C. E. Shannon and J. McCarthy [Eds.], Princeton University Press, Princeton, N.J., 1956, 3–40.

[49] Knuth, Donald E. "RUNCIBLE—Algebraic translation on a limited computer," *Comm. ACM* **2**, 11 (November, 1959), 18–21.

[There is a bug in the flowchart. The arc labeled "2" from the box labeled "θ:" in the upper left corner should go to the box labeled R_M = 8003.]

[50] KNUTH, DONALD E. *Fundamental algorithms, The art of computer programming*, Vol. **1**, Addison-Wesley, Reading, Mass. 1968 2d ed., 1973, 634 pp

[51] KNUTH, DONALD E. "An empirical study of FORTRAN programs," *Software—Practice and Experience* **1**, 2 (April–June 1971), 105–133.

[52] KNUTH, DONALD E., AND FLOYD, ROBERT W. "Notes on avoiding 'go to' statements," *Information Processing Letters* **1**, 1 (February 1971), 23–31, 177.

[53] KNUTH, DONALD E. "George Forsythe and the development of Computer Science," *Comm. ACM* **15**, 8 (August 1972), 721–726.

[54] KNUTH, DONALD E. *Sorting and searching, The art of computer programming*, Vol. **3**, Addison-Wesley, Reading, Mass., 1973, 722 pp.

[55] KNUTH, DONALD E. "A review of '*structured programming*'," Stanford Computer Science Department report STAN-CS-73-371, Stanford Univ., Stanford, Calif., June 1973, 25 pp.

[56] KNUTH, DONALD E., AND SZWARCFITER, JAYME L. "A structured program to generate all topological sorting arrangements," *Information Processing Letters* **2**, 6 (April 1974) 153–157.

[57] KOSARAJU, S. RAO. "Analysis of structured programs," *Proc. Fifth Annual ACM Symp. Theory of Computing*, (May 1973), 240–252; also in *J. Computer and System Sciences*, **9**, 3 (December 1974).

[58] LANDIN, P. J. "A correspondence between ALGOL 60 and Church's lambda-notation: part I," *Comm. ACM* **8**, 2 (February 1965), 89–101.

[59] LANDIN, P. J. "The next 700 programming languages," *Comm. ACM* **9**, 3 (March 1966), 157–166.

[60] LEAVENWORTH, B. M. "Programming with(out) the GOTO," *Proc. ACM Annual Conference*, Boston, Mass., August 1972, 782–786.

[61] MANNA, ZOHAR, AND WALDINGER, RICHARD J. "Towards automatic program synthesis," in Symposium on Semantics of Algorithmic Languages, *Lecture Notes in Mathematics* **188**, E. Engeler [Ed.], Springer-Verlag, New York, 1971, 270–310.

[62] McCARTHY, JOHN. "Recursive functions of symbolic expressions and their computation by machine, part I," *Comm. ACM* **3**, 4 (April 1960), 184–195.

[63] McCARTHY, JOHN. "Towards a mathematical science of computation," in *Proc. IFIP Congress 1962, Munich, Germany*, North-Holland Publ. Co., Amsterdam, The Netherlands, 1963, 21–28.

[64] McCRACKEN, DANIEL D. "Revolution in programming," *Datamation* **19**, 12 (December 1973), 50–52.

[65] McKEEMAN, W. M.; HORNING, J. J.; AND WORTMAN, D. B. *A compiler generator*, Prentice-Hall, Englewood Cliffs, N. J., 1970, 527 pp.

[66] MILLAY, EDNA ST. VINCENT. "Elaine"; cf. Bartlett's *Familiar Quotations*.

[67] MILLER, EDWARD F., JR., AND LINDAMOOD, GEORGE E. "Structured programming: top-down approach," *Datamation* **19**, 12 (December 1973), 55–57.

[68] MILLS, H. D. "Top-down programming in large systems," in *Debugging techniques in large systems*, Randall Rustin [Ed.], Prentice-Hall, Englewood Cliffs, N. J., 1971, 41–55.

[69] MILLS, H. D. "Mathematical foundations for structured programming," report FSC 72-6012, IBM Federal Systems Division, Gaithersburg, Md. (February 1972), 62 pp.

[70] MILLS, H. D. "How to write correct programs and know it," report FSC 73-5008, IBM Federal Systems Division, Gaithersburg, Md. (1973), 26 pp.

[71] NASSI, I. R., AND AKKOYUNLU, E. A. "Verification techniques for a hierarchy of control structures," Tech. report 26, Dept. of Computer Science, State Univ. of New York, Stony Brook, New York (January 1974), 48 pp.

[72] NAUR, PETER [Ed.] "Report on the algorithmic language ALGOL 60," *Comm. ACM* **3**, 5 (May 1960), 299–314.

[73] NAUR, PETER. "Go to statements and good Algol style," *BIT* **3**, 3 (1963), 204–208.

[74] NAUR, PETER. "Program translation viewed as a general data processing problem," *Comm. ACM* **9**, 3 (March 1966), 176–179.

[75] NAUR, PETER. "An experiment on program development," *BIT* **12**, 3 (1972), 347–365.

[76] PAGER, D. "Some notes on speeding up certain loops by software, firmware, and hardware means," in *Computers and automata*, Jerome Fox [Ed.], John Wiley & Sons, New York 1972, 207–213; also in *IEEE Trans. Computers*, **C-21**, 1 (January 1972), 97–100.

[77] PETERSON, W. W.; KASAMI, T.; AND TOKURA, N. "On the capabilities of **while**, **repeat**, and **exit** statements," *Comm. ACM* **16**, 8 (August 1973), 503–512.

[78] PETERSON, W. WESLEY. personal communication, April 2, 1974.

[79] RAIN, MARK AND HOLAGER, PER. "The present most recent final word about labels in MARY," *Machine Oriented Languages Bulletin* **1**, Trondheim, Norway (October 1972), 18–26.

[80] REID, CONSTANCE. *Hilbert*, Springer-Verlag, New York, 1970, 290 pp.

[81] REYNOLDS, JOHN. "Fundamentals of structured programming," Systems and Info. Sci. 555 course notes, Syracuse Univ., Syracuse, N.Y., Spring 1973.

[82] SATTERTHWAITE, E. H. "Debugging tools for high level languages," *Software—Practice and Experience* **2**, 3 (July–September 1972), 197–217.

[83] SCHNECK, P. B., AND ANGEL, ELLINOR. "A FORTRAN to FORTRAN optimizing compiler," *Computer J.* **16**, 4 (1973), 322–330.

[84] SCHORRE, D. V. "META-II—a syntax-directed compiler writing language," *Proc. ACM National Conference*, Philadelphia, Pa., 1964, paper D1.3.

[85] SCHORRE, D. V. "Improved organization for procedural languages," Tech. memo TM 3086/002/00, Systems Development Corp., Santa Monica, Calif., September 8, 1966, 8 pp.

[86] SHIGO, O.; SHIMOMURA, T.; FUJIBAYASHI S.; AND MAEJIMA, T. "SPOT: an experimental system for structured programming" (in Japanese), *Conference Record*, Information Processing Society of Japan, 1973. [Translation available from the authors, Nippon Electric Company Ltd., Kawasaki, Japan.]

[87] STRACHEY, C. "Varieties of programming language," in *High level languages*, C. Boon [Ed.], Infotech State of the Art Report 7, 1972, 345-362.

[88] STRONG, H. R. JR. "Translating recursion equations into flowcharts," *J. Computer and System Sciences* 5, 3 (June 1971), 254-285.

[89] TEITELMAN, W. "Toward a programming laboratory," in *Software Engineering Techniques*, J. N. Buxton and B. Randall [Eds.], NATO Scientific Affairs Division, Brussels, Belgium, 1970, 137-149.

[90] TEITELMAN, W. et al. "INTERLISP reference manual," Xerox Palo Alto Research Center, Palo Alto, Calif., and Bolt Beranek and Newman, Inc., 1974.

[91] WALKER, S. A., AND STRONG, H. R. "Characterizations of flowchartable recursions," *J. Computer and System Sciences* 7, 4 (August 1973), 404-447.

[92] WEGNER, EBERHARD. "Tree-structured programs," *Comm. ACM* 16, 11 (November 1973), 704-705.

[93] WEINBERG, GERALD M. "The psychology of improved programming performance," *Datamation* 17, 11 (November 1972), 82-85.

[94] WIRTH, N. "On certain basic concepts of programming languages," Stanford Computer Science Report CS 65, Stanford, Calif. (May 1967), 30 pp.

[95] WIRTH, N. "PL 360, a programming language for the 360 computers," *J. ACM* 15, 1 (January 1968), 37-74.

[96] WIRTH, N. "Program development by stepwise refinement," *Comm. ACM* 14, 4 (April 1971), 221-227.

[97] WIRTH, N. "The programming language Pascal," *Acta Informatica* 1, 1 (1971), 35-63.

[98] WULF, W. A.; RUSSELL, D. B.; AND HABERMANN, A. N. "BLISS: A language for systems programming," *Comm. ACM* 14, 12 (December 1971), 780-790.

[99] WULF, W. A. "Progamming without the goto," *Information Processing 71, Proc. IFIP Congress*, Vol. 1, North-Holland Publ. Co., Amsterdam, The Netherlands, 1971, 408-413.

[100] WULF, W. A. "A case against the GOTO," *Proc. ACM 1972 Annual Conference*, Boston, Mass. (August 1972), 791-797.

[101] WULF, W. A.; JOHNSON, RICHARD K.; WEINSTOCK, CHARLES P.; AND HOBBS, STEVEN O. "The design of an optimizing compiler," Computer Science Department report, Carnegie-Mellon Univ., Pittsburgh, Pa., (December 1973), 103 pp.

[102] ZAHN, CHARLES T. "A control statement for natural top-down structured programming," presented at Symposium on Programming Languages, Paris, 1974.