CS 148

NOTES ON AVOIDING "GO TO" STATEMENTS

BY

D. E. KNUTH
R. W. FLOYD

TECHNICAL REPORT NO. CS 148
JANUARY 1970

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

NOTES ON AVOIDING "GO TO" STATEMENTS

By

D. E. Knuth and R. W. Floyd

The research reported here was supported by IBM Corporation.

NOTES ON AVOIDING "GO TO" STATEMENTS

D. E. Knuth and R. W. Floyd

During the last decade there has been a growing sentiment that the use of "go to" statements is undesirable, or actually harmful. This attitude is apparently inspired by the idea that programs expressed solely in terms of conventional iterative constructions ("for", "while", etc.) are more readable and more easily proved correct. In this note we will make a few exploratory observations about the use and disuse of go to statements, based on two typical programming examples (from "symbol table searching" and "backtracking").

In the first place let us consider systematic ways for eliminating go to statements. There are two apparent ways to achieve this:

(a) Recursive procedure method. Suppose that each statement of a program is labeled. Replace each labeled statement

L: S

by

procedure L; begin S; L' end

where L' is the static successor of the statement S . A go to statement becomes simply a procedure call. The program ends by calling a null procedure. This construction shows that the mere elimination of go to statements does not automatically make a program better or easier to

follow; "go to" is in some sense a special case of the procedure calling mechanism. (It is instructive in fact to consider this construction in reverse, realizing that it is sometimes more efficient to replace procedure calls by go to statements!)

(b) Regular expression method. For convenience, imagine a program expressed in flowchart form, as a directed graph. It is well known that all paths through this graph can be represented by "regular expressions" involving the operations of concatenation, alternation, and "star"; these latter correspond to familiar constructions in programming languages which do not depend on go to statements. Therefore it appears that 'go to' statements can be eliminated, although it may be necessary to duplicate the code for other statements in several places. This process is essentially what John Cocke calls "node splitting".

Consider, for example the following well-known programming situation:

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

(Let us assume, for convenience, that $i = n+1$ if the for loop is exhausted.) It is not obvious that the go to statement here is all that unsightly, but let us suppose that we are reactionary enough that we really want to abolish them from programming languages. [See Dijkstra Comm. ACM 11 (1968), 147-148.] One way to avoid the go to is to use a recursive procedure:

```
procedure find;
   if i > n then begin n := i; A[i] := x; B[i] := 0 end
      else if A[i] ≠ x then begin i := i+1; find end;
i := 1; find; B[i] := B[i]+1;
```

An optimizing compiler could perhaps produce the same code for both programs, but again it is debatable which program is most readable and simple.

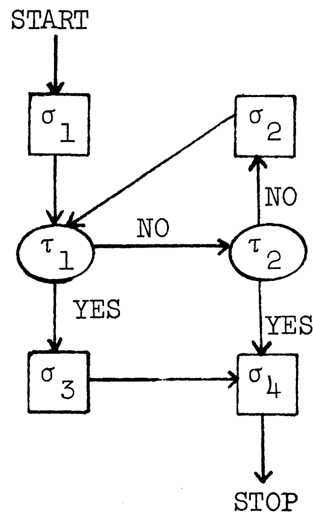
Other solutions change the structure of the program slightly:

(a)
```
i := 1;
while i ≤ n and A[i] ≠ x do i := i+1;
if i > n then begin n := i; A[i] := x; B[i] := 0 end;
B[i] := B[i]+1;
```

(b)
```
i := 1;
while A[i] ≠ x do
begin i := i+1;
   if i > n then begin n := i; A[i] := x; B[i] := 0 end
end;
B[i] := B[i]+1;
```

Solution (b) assumes that $n > 0$. Both solutions increase the amount of calculation that is specified: (a) tests "$i > n$" twice, while (b) tests "$A[i] \neq x$" after n has been increased.

The flowchart of the original program is:



(*)

$\sigma_1 \equiv$ i := 1

$\tau_1 \equiv$ i > n ?

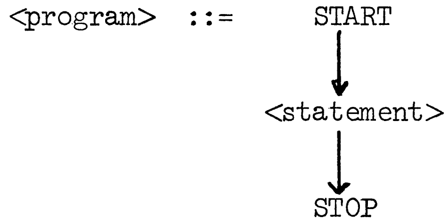
$\tau_2 \equiv$ A[i] = x ?

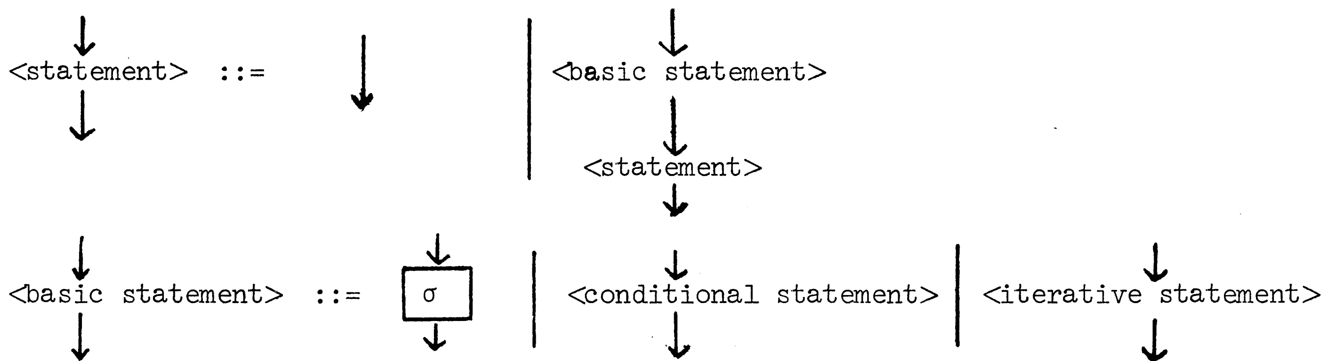
$\sigma_2 \equiv$ i := i+1

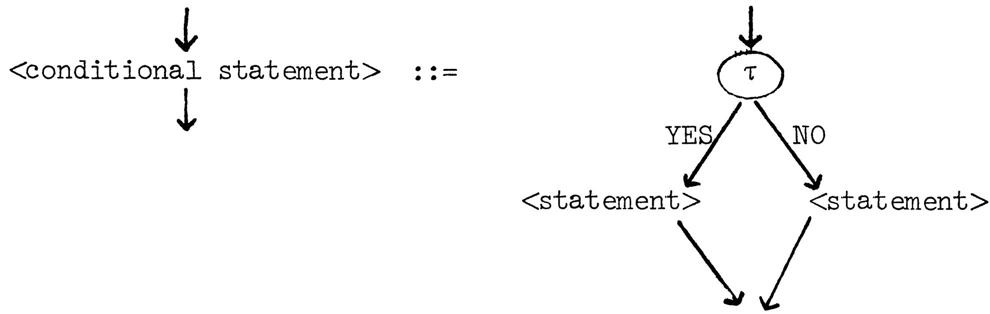
$\sigma_3 \equiv$ n := i; A[i] := x; B[i] := 0

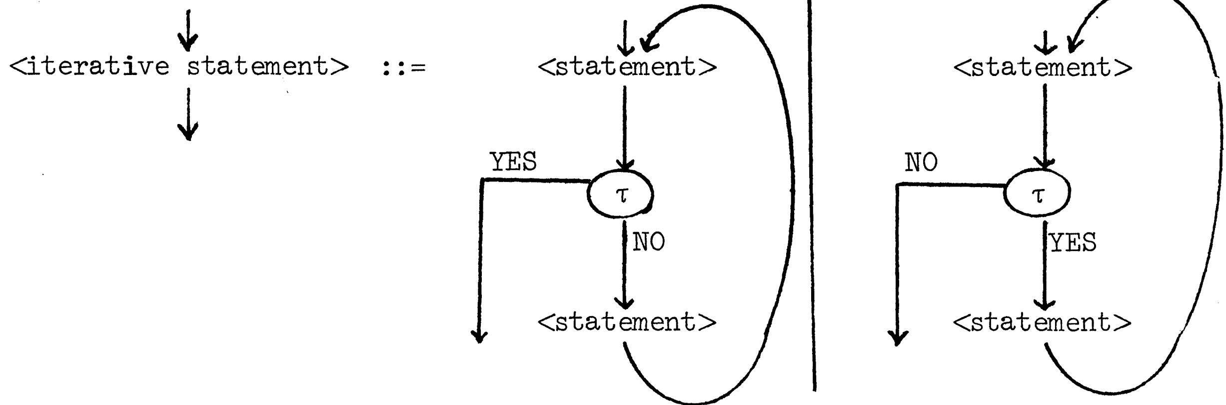
$\sigma_4 \equiv$ B[i] := B[i]+1

By a suitable extension of BNF we can write a grammar for all flowcharts producible by a language without procedure calls or go to statements:









Here σ denotes a "statement" and τ denotes a "test".

We have not completely analyzed this grammar, although it appears to be unambiguous; there is probably an efficient parsing algorithm which will decide whether or not a given flowchart is derivable from the grammar, constructing a derivation when one exists. But we can easily prove that the above flowchart is <u>not</u> producible by this grammar. In fact, a stronger result is true:

<u>Theorem</u>. No flowchart producible by the above grammar specifies precisely the computations of the above example flowchart (*).

This theorem contradicts our observations above about regular expressions being reducible to concatenation, alternation, and iteration;

for our flowcharts provide each of these operations, yet they cannot reproduce the computations in (*). What went wrong? Perhaps it is that regular expressions are nondeterministic, while computations are inherently deterministic; but no, it is well known that regular expressions may be considered to be deterministic. The difference really lies in the nature of computational tests.

Thus, let us consider a special class $\mathfrak{R}$ of regular expressions; $\mathfrak{R}$ describes all computational sequences (paths in the flowchart) producible by flowcharts corresponding to a language without go-to statements:

the empty sequence is in $\mathfrak{R}$.

$\sigma \in \mathfrak{R}$, for all statements σ.

$\mathfrak{R}_1 \mathfrak{R}_2 \in \mathfrak{R}$, for all $\mathfrak{R}_1$ and $\mathfrak{R}_2 \in \mathfrak{R}$.

$(\tau_Y \mathfrak{R}_1 | \tau_N \mathfrak{R}_2)$, for all $\mathfrak{R}_1$ and $\mathfrak{R}_2 \in \mathfrak{R}$ and all tests τ.

$(\tau_Y \mathfrak{R}_1)^* \tau_N \in \mathfrak{R}$, $(\tau_N \mathfrak{R}_1)^* \tau_Y \in \mathfrak{R}$, for all $\mathfrak{R}_1 \in \mathfrak{R}$ and all tests τ.

Here the subscripts Y and N denote the "YES" or "NO" branches in the flowchart.

To prove the theorem, consider the computational sequences producible by the flowchart (*); they may be described by the regular expression

$$\sigma_1 (\tau_{1N} \tau_{2N} \sigma_2)^* (\tau_{1Y} \sigma_3 | \tau_{1N} \tau_{2Y}) \sigma_4 \quad . \qquad (**)$$

We will show that the corresponding regular event (the sequences defined by this regular expression) cannot be defined by any of the regular expressions in $\mathfrak{R}$.

Every regular expression in $\mathfrak{R}$ which specifies infinitely many sequences includes some test τ with one of the following two properties:

(i) Every occurrence of τ_Y is followed by at least one occurrence of τ_N.

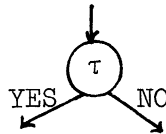
or (ii) Every occurrence of τ_N is followed by at least one occurrence of τ_Y.

The infinitely many sequences specified by (**) do not have any such test since the sequences include

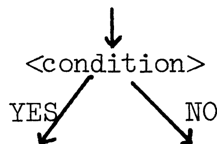
$$\sigma_1 \tau_{1Y} \sigma_3 \sigma_4 \ , \ \sigma_1 \tau_{1N} \tau_{2Y} \sigma_4 \ , \ \sigma_1 \tau_{1N} \tau_{2N} \sigma_2 \tau_{1Y} \sigma_3 \sigma_4 \ .$$

Hence no regular expression in $\mathfrak{R}$ can produce the regular event (**), and the theorem is proved.

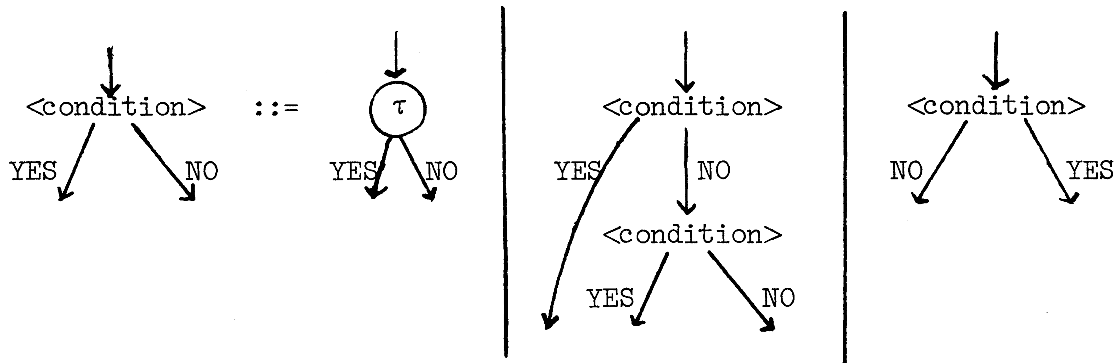
Perhaps the reader feels that the above proof is too "slick", or that something has been concealed. In fact, this is quite true; we have penalized the class of flowcharts too severely! Compound tests such as "τ_1 and τ_2" have not been allowed sufficient latitude. Our flowchart grammar should be extended as follows: Replace



in the definitions of <conditional statement> and <iterative statement> by



and add the new definition



The grammar now becomes ambiguous in several cases, although the ambiguity can be removed at the expense of some complications which are irrelevant here. More important is the change to grammar $\mathfrak{R}$, where we are allowed to substitute

$$\tau'_Y \quad \text{for} \quad \tau_N \, , \qquad \tau'_N \quad \text{for} \quad \tau_Y$$

or

$$\tau'_N\tau''_N \quad \text{for} \quad \tau_N \, , \qquad (\tau'_N|\tau'_N\tau''_Y) \quad \text{for} \quad \tau_Y$$

whenever τ, τ', τ'' are tests. Thus since $\sigma_1(\tau_N\sigma_2)^*\tau_Y\sigma_4 \in \mathfrak{R}$, so is

$$\sigma_1(\tau_{1N}\tau_{2N}\sigma_2)^*(\tau_{1Y}|\tau_{1N}\tau_{2Y})\sigma_4 \, ,$$

and this is the same as (**) with σ_3 deleted. The theorem above is almost false! But we can still prove it by an exhaustive case analysis, considering all possible substitutions of compound tests and showing that none are permissible because of the presence of σ_3.

The theorem becomes almost false in another sense too, when compound conditions are considered, since the expression

$$\sigma_1(\tau_{1N}\tau_{2N}\sigma_2)^*(\tau_{1Y}|\tau_{1N}\tau_{2Y})(\tau_{1Y}\sigma_3|\tau_{1N})\sigma_4$$

is in $\mathfrak{R}$ and it differs from (**) only in that τ_{1Y} becomes $\tau_{1Y}\tau_{1Y}$ and $\tau_{1N}\tau_{2Y}$ becomes $\tau_{1N}\tau_{2Y}\tau_{1N}$. The sequences are essentially the same except that redundant tests are made. We could therefore consider equivalence operations on regular expressions, allowing commutativity of successive tests, and an idempotent law $\tau_Y\tau_Y = \tau_Y$. In that case our theorem would become false; but we can easily find another flowchart for which the theorem still applies: Simply put another statement box σ_5 between τ_1 and τ_2 . Then no two tests are adjacent, and our original "slick" proof immediately shows that the regular event defined by

$$\sigma_1(\tau_{1N}\sigma_5\tau_{2N}\sigma_2)^*(\tau_{1Y}\sigma_3|\tau_{1N}\sigma_5\tau_{2Y})\sigma_4$$

is not equivalent to any regular event definable with $\mathfrak{R}$. (When no two tests are adjacent compound conditions cannot appear, nor do any of the equivalences apply, so none of the extensions affect the original proof of the theorem.)

Therefore our "slick" proof is vindicated, and _we have proved the existence of programs whose go to statements cannot be eliminated without introducing procedure calls_.

Let us now consider a second example program, taken this time from a typical "backtracking" or exhaustive enumeration application. Most backtrack problems can be abstracted into the following form:

```
start:  m[1] := 0; k := 0;
 up:    k := k+1; list(k); a[k] := m[k];
 try:   if a[k] < m[k+1] then begin move (a[k]); go to up end;
 down:  k := k-1;
        if k = 0 then go to done;
        unmove (a[k]);
        a[k] := a[k]+1; go to try;
 done:
```

Here the procedures list, move, unmove may be regarded as manipulating a variable-width stack s[0],s[1],... of possible choices in this abstracted algorithm. Procedure list(k) determines all possible choices at the k-th level of backtracking, based on the previously made choices a[1],...,a[k-1] . If there are c choices now possible, list(k) will set m[k+1] := m[k]+c , and it will also set the stack entries s[m[k]+1],...,s[m[k]+c] to identify the choices. (Note that c can be zero. The choices might be, for example, where to place the k-th queen on a chessboard, given positions of k-1 other queens, if we are trying to solve the queens' problem.) Procedure move(t) makes the decision to choose alternative s[t] ; this usually means that some internal tables need to be updated. Procedure unmove(t) reverses the decisions made by move(t) .

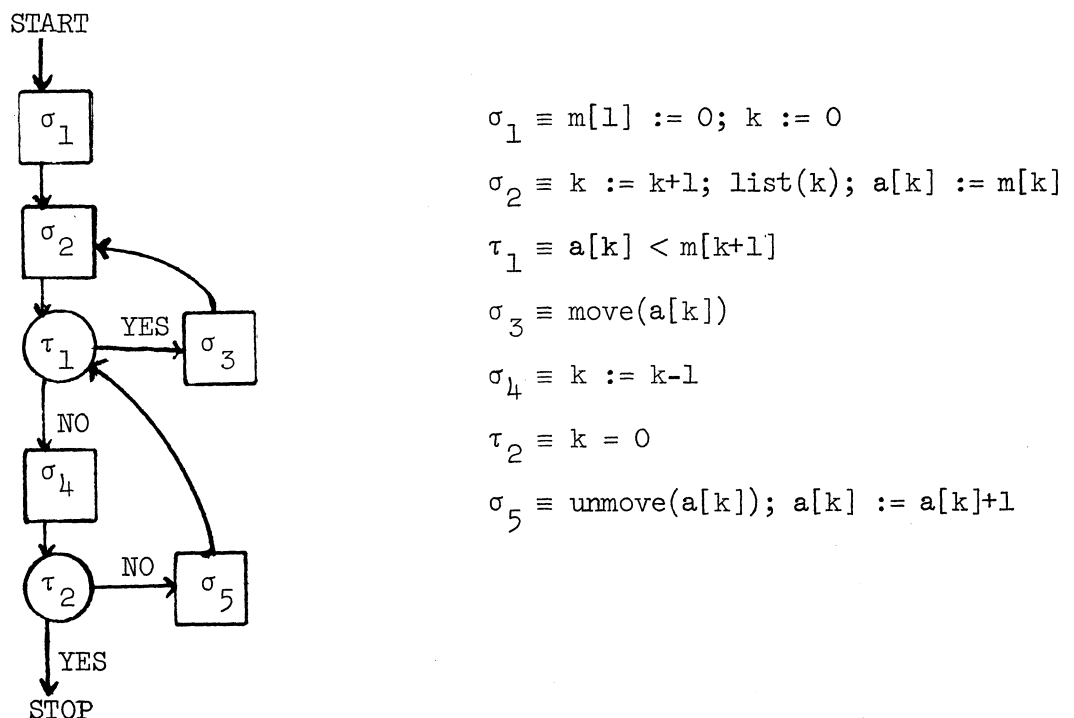
It is not necessary to understand the exact mechanism of this construction, although people familiar with backtracking should find the previous paragraph self-explanatory; the main point is that essentially all backtracking programs have the form of the above program, when appropriate sequences of code are substituted for list(k) , move(a[k]) , and unmove(a[k]) , hence the program is worth considering from the standpoint of go-to elimination.

First we can eliminate go-to's by introducing a procedure:

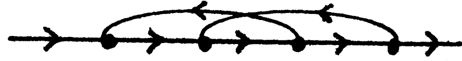
```
procedure backtrack(k); value k; integer k;
   begin list(k); a[k] := m[k];
      while a[k] < m[k+1] do
         begin move(a[k]); backtrack(k+1); unmove(a[k]);
            a[k] := a[k]+1
         end
   end backtrack;
m[1] := 0; backtrack(1);
```

This use of recursion is rather clean, so the above program is attractive except for the procedure-calling overhead (which is important since backtrack programs typically involve many millions of iterations). It is an interesting exercise to prove this program equivalent to our first version.

Now let's try to eliminate the go to statements without introducing a new procedure. The flowchart is:

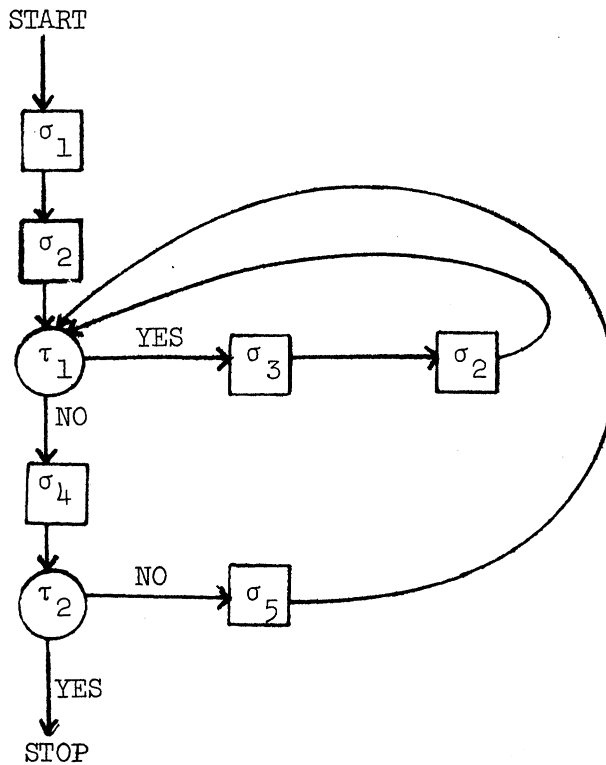


Here we have the basic flowchart structure



instead of the previous situation when we had

It turns out that node-splitting works in this case but not the other; we can make two copies of node σ_2 in the above flowchart and we obtain



This diagram obviously satisfies the conditions of our flowchart grammar above, so we can avoid the go to statements.

What is the resulting program? Our flowchart grammar above allows more general iterative statements than present-day programming languages will admit. A general iterative construction might be written

begin loop σ_1; exit loop if τ_1; σ_2 end loop; (***)

but today's languages only consider the case that σ_1 is empty:

while $> \tau_1$ do σ_2;

or if σ_2 is empty:

do σ_1 until τ_1;

We can always rewrite (***) in the equivalent form

σ_1; while $\neg \tau_1$ do begin σ_2; σ_1 end;

but this is quite unattractive when σ_1 is long, so a programmer will certainly prefer to use go to statements in that case. If we want to teach programmers to avoid go to statements, we must provide them with a sufficiently rich syntax of iterative statements to serve as a substitute.

Using (***) leads to the following program for backtracking without go to statements:

```
m[1] := 0; k := 1; list(1); a[1] := 0;
begin loop
   while a[k] < m[k+1] do
      begin move(a[k]);
         k := k+1; list(k); a[k] := m[k]
      end;
   k := k-1;
exit loop if k = 0;
   unmove(a[k]); a[k] := a[k]+1
end loop;
```

This code, although free of "go to statements", involves an uncomfortable element which may not make it very palatable: the "*while* a[k] < m[k+1]" is a rather peculiar condition since k varies and the test involves different variables each time. This is quite different in effect from the appearance of the same clause in our recursive procedure backtrack(k) . It is possible to think of the program in a fairly natural way nevertheless, for example (in tree language) as follows:

```
start at root of search tree;
begin loop
   while possible to go down and left in tree do so;
   move up one level in the tree;
exit loop if at the root;
   move to the right in the tree;
end loop;
```

this is a typical tree traversal algorithm. Yet it is debatable whether or not the elimination of go to statements was an improvement.

The syntax in (***) is perhaps not the best way to improve iteration statements. An alternative proposal, based on some unpublished ideas of Wirth, has just been implemented as an extension to Stanford's ALGOL W compiler: The statement

```
repeat <block>
```

has the effect of

```
L1: <block>; go to L1; L2:
```

and the statement

```
exit
```

has the effect of

go to L_2

where L_2 is the second implicit label corresponding to the smallest repeat block statically enclosing the exit statement. Thus, (***) becomes

repeat begin σ_1; if τ_1 then exit; σ_2 end;

and we can even write our symbol table search routine without go to statements:

```
i := 1;
repeat begin
   while i ≤ n do if A[i] = x then exit else i := i+1;
   n := i; A[i] := x; B[i] := 0; exit
end;
B[i] := B[i]+1;
```

Here the "repeat loop" is never repeated, but the desired effect has been achieved. It appears doubtful that this repeat-exit mechanism will be able to eliminate go to statements in general, since it only allows a "one-level exit"; further study of these issues is indicated.