A GENEALOGY OF CONTROL STRUCTURES

by

Henry F. Ledgard
Computer and Information Science Department
University of Massachusetts
Amherst, Mass 01002

ABSTRACT

The issue of control structures has had a heated history in programming. To put this issue on a solid footing, this paper reviews numerous theoretical results on control structures and explores their practical implications.

The classic result of Bohm and Jacopini on the theoretical completeness of if-then-else and while-do is discussed. Several recent ideas on control structures are then explored. These include a review of various other control structures, results on time/space limitations, and theorems relating the relative power of control structures under several notions of equivalence.

In conclusion, a case is made against the recent arguments of Knuth [K2] on the utility of the GOTO statement.

Work reported herein was supported by the U.S. Army Research Office and the National Bureau of Standards.

Revised version November 1974

Keywords: Structured Programming, Control Structures, GOTO Statements, Language Design, PASCAL

C.R. Categories: 4.2, 5.24
A GENEALOGY OF CONTROL STRUCTURES

The issue of control structures has had a heated history in programming. To put this issue on a solid footing, this paper reviews numerous theoretical results on control structures and explores their practical implications. The classic result of Bohm and Jacopini on the theoretical completeness of if-then-else and while-do is discussed. Several recent ideas on control structures, results on time/space limitations, and theorems relating the relative power of control structures under several notions of equivalence. In conclusion, a case is made against the recent arguments of Knuth (K2) on the utility of the GOTO statement.
I. INTRODUCTION

In the last decade we have seen the rapidly growing interest in the areas of structured programming and software quality. Although the major attention has been placed on top-down programming techniques and control structures, the concern over the quality of software has also included the definition, modularity, clarity, changeability, and documentation of programs.

This paper focuses on the issue of control structures. While it may well be argued that the control structure issue has been entirely overworked, the debates and polarized opinions remain. At one extreme we have the views of Mills [M1], who has religiously advocated the use of the if-then-else and while-do control structures. At the other extreme, we have the views of Knuth [K2], who has recently given vigorous arguments on the utility of the goto.

Over the years, a number of theoretical results have been presented on the limitations of various control structures. Notable are the works of Bohm and Jacopini [B1, M1], Knuth and Floyd [K3], Bruno and Steiglitz [B2], Peterson, Kasami and Tokura [P1], and importantly, Kosaraju [K4]. These results have placed the control structure issue on a firm foundation. In this paper I present a framework for reviewing these results and discuss their practical implications.

The programming language PASCAL is used here as communication language. Unfortunately, PASCAL omits several constructs that I consider important in contemporary languages. To remedy this situation, I have made a number of extensions, as required by the examples. I believe that these extensions will pose little problem for the reader.
II. CLASSES OF CONTROL STRUCTURES

This section presents various classes of control structures. Aside from minor variants, these classes embrace the control structures found in most algorithmic languages. Readers who are familiar with these control structures may need only a quick reading of this section to become familiar with the terminology given here.

(a) D-structures. We begin with the definition of "D-structures", D for Dijkstra, as in [B2]. A D-structure (see Figure 1) is any program constructed only from the following 1-in, 1-out primitive structures

(i) basic actions (e.g., assignment statements, procedure calls, input/output statements),

(ii) compositions "s_1; s_2" of two D-structures,

(iii) conditional constructs of the form "if p then s_1 else s_2" based on a predicate p (having no side effects) and two D-structures s_1 and s_2, and

(iv) loops of the form "while p do s", where p is a predicate (having no side-effects) and s is a D-structure.

D-structures also include conventional for loop structures. These can be readily defined via basic actions and while-do loops.

D-structures have received prominent attention in the literature. Bruno and Steiglitz [B2], Ashcroft and Manna [A1], and Knuth and Floyd [K3] have explored the reduction of arbitrarily structured programs into D-structure form. Mills [M1] and Dijkstra [D2] have explored programming with these structures, and numerous other researchers [F1,W1,L1,W3] have considered these structures in various ways.

(b) D'-structures. The class of D-structures gives rise to several natural extensions. One class of control structures, here called D'-structures, is shown in Figure 2. This class comprises the class of D-structures, with the addition of the following 1-in, 1-out structures: single branching if statements, n-way branching case statements, and repeat-until loops.
(1a) actions

(1b) compositions

(1c) if-then-else stmts

(1d) while-do loops

Figure (1) Definition of D-structures

(2a) any D-structure, plus

(2b) if-then stmts

(2c) repeat-until loops

(2d) case (1) of case stmts

1 \leq i \leq n

Figure (2) Definition of D'-structures
Aside from the goto, D'-structures comprise the set of PASCAL control structures.

I next turn to some substantive generalizations of the previous control structures. These generalizations stem from the notions of procedure return statements, loop exits arising from exceptional conditions, and repeated loop invocations from within loops.

(c) BJ̵ Structures

First consider a definition of BJ̵ structures (due to Bohm and Jacopini [Bl]), where \( n \geq 1 \). A BJ̵ structure is composed of basic actions, compositions, if-then-else structures, and 1-in, 1-out control structures \( k \) where \( k \geq n \). An \( k \)-structure (see Figure 3) contains \( k \) successive predicates and actions with \( k \) exits, one for each of the \( k \) predicates. An \( \Omega_k \)-structure is equivalent to the following program schema.

\[
\begin{align*}
d & \text{do} \quad \text{if } p_1 \text{ exit;} \\
& \quad s_1; \\
& \quad \text{if } p_2 \text{ exit;} \\
& \quad s_2; \\
& \quad \vdots \\
& \quad \text{if } p_k \text{ exit;} \\
& \quad s_k; \\
& \text{cycle} \\
\end{align*}
\]

If an \( \Omega_k \)-structure is viewed as a procedure, the exit escapes above would be analogous to PL/I-like return statements. Note that \( \Omega_1 \) is a while-do structure. BJ̵ structures are similar to that proposed by Zahn [Z1].

(d) RE̵, REC̵, DRE̵, and DREC̵ structures.

Next consider the definition of RE̵ structures and their variants. An RE̵ structure is composed of basic actions, compositions, if-then-else structures, exit commands of the form exit(\( i \)), where \( i \) is a positive integer constant such that \( 1 \leq i \leq n \), and repeat-end constructs of the form

\[
\begin{align*}
& \text{repeat} \\
& \quad s_1; \\
& \quad s_2; \\
& \quad \vdots \\
& \quad s_n \\
& \text{end}
\end{align*}
\]
Figure (3) Definition of $\Omega_k$ - Structures
where the $s_i$ are other $\text{RE}_n$-structures (see Figure 4). On execution, the commands within a $\text{repeat-end}$ block are to be repeated indefinitely until an $\text{exit}$ command is encountered. The execution of an $\text{exit}(i)$ command causes termination of $i$ enclosing $\text{repeat-end}$ loops. In the case where there are fewer than $i$ enclosing loops, all enclosing loops are terminated. $\text{RE}_n$-structures are similar to $\text{D}$-structures, except that loops may be exited at arbitrary points within the loop. $\text{RE}_n$-structures have been developed from the control structures used in BLISS [W4].

An $\text{REC}_n$-structure (see Figure 5) is similar to an $\text{RE}_n$-structure, with the inclusion of additional commands of the form $\text{cycle}(i)$. The execution of a $\text{cycle}(i)$ command is similar to an $\text{exit}(i)$ command, except that the $i$-th enclosing $\text{repeat-end}$ loop is re-executed.

A $\text{DRE}_n$-structure is defined as a $\text{RE}_n$-structure with the possible inclusion of $\text{while-do}$ structures. The execution of an $\text{exit}(i)$ command causes termination of the $i$-th enclosing $\text{repeat-end}$ loop, ignoring any enclosing $\text{while-do}$ loops.

A $\text{DREC}_n$-structure is defined as a $\text{DRE}_n$-structure with the addition of $\text{cycle}(i)$ command.

$\text{RE}_n$, $\text{REC}_n$, $\text{DRE}_n$, and their variants conform to conventional programs for which transfers of control are restricted to the ends or beginning of enclosing control loops.

(e) $\text{P}_n$-Structures and $\text{L}$-Structures

Finally, a $\text{P}_n$-structure is defined as any well-formed structure such that all 1-in, 1-out sub-structures have at most $n$ predicates. An $\text{L}$-structure is defined as any well-formed structure, i.e., any structure with no restrictions on the number or configuration of predicates, actions, and transfers of control. An $\text{L}$-structure corresponds to a program with free use of labels and $\text{goto}$ statements.

The above control structures embrace most of the explicit control structures found in conventional languages. It is important to note that these control structures do not take into account various "scope" rules often associated with these control structures. In PASCAL for example, the value of a $\text{for}$-loop control variable upon exit is undefined. Such issues arising from the value of internal variables when control is transferred into or out of a control loop are not treated in this paper.
Figure (4) Repeat-exit Structures

Figure (5) Repeat-exit-cycle Structures
III. THE NOTIONS OF REDUCIBILITY AND EQUIVALENCE

Numerous results on the relative power of control structures are presented in the next section. In an attempt to present these results in a rigorous but conceptually simple framework, we first define five sets of conversion rules for converting a control structure into another form, and then introduce the notions of "reducibility" and "equivalence" of classes of control structures under these conversion rules. These notions are motivated by Kosaraju [K4].

3.1 Conversion rules

In defining the various notions of conversion of a structure $S_1$ to a structure $S_2$, the following five properties are singled out:

(P1) For every input $S_2$ computes the same function as $S_1$. (i.e., $S_2$ performs the same computation as $S_1$.)

(P2) The primitive actions and predicates in $S_2$ are precisely those of $S_1$.

(P3) For every input, the sequence of primitive actions and predicates executed in $S_2$ is identical to that in $S_1$.

(P4) $S_2$ can be obtained from $S_1$ by "node splitting". (Basically, node splitting allows one to eliminate structures with multiple inputs by making multiple copies of the paths through the structures [see P1].)

(P5) Each occurrence of a primitive action or predicate in $S_1$ is used at most once in $S_2$. (i.e., multiple copies of predicates and actions in $S_1$ are not allowed.)

The conversion rules can now be easily stated as follows:

(a) **Very Strong Conversion.** A structure $S_1$ is said to be "very strongly" convertible to a structure $S_2$ iff properties (P1) through (P5) are satisfied.

"Very strong conversion" is indeed very strong. Basically, the only allowed rewriting rule in converting $S_1$ to $S_2$ is a reconfiguration of the
existing predicates and actions in \( S_1 \). From a programmer's viewpoint, there is one important consideration, namely that \( S_1 \) and \( S_2 \) differ only in notational convenience.

(b) **Node-Splitting Conversion**: A structure \( S_1 \) is said to be "node-splitting" convertible to \( S_2 \) iff properties (P1) through (P4) are satisfied.

Node-splitting conversion is still quite strong. Basically, \( S_2 \) must be derivable from \( S_1 \) by well-defined rewriting rules (node-splitting).

(c) **Strong Conversion**. A structure \( S_1 \) is said to be "strongly" convertible to a structure \( S_2 \) iff properties (P1) through (P3) are satisfied.

Strong conversion is clearly not as strong as node-splitting conversion for a restructuring of the primitive actions and predicates in \( S_1 \) is allowed in \( S_2 \). Nevertheless, the reduction is still strong in the sense that the computation sequences in \( S_1 \) and \( S_2 \) must still be identical.

(d) **Semantic Conversion**. A structure \( S_1 \) is said to be "semantically" convertible to \( S_2 \) iff properties (P1) and (P2) hold.

Semantic conversion implies a significantly less restrictive condition than the above notions of conversion, for the only restriction on the conversion of \( S_1 \) to \( S_2 \) is the prohibition of "new semantics", i.e., new actions, predicates, or variables.

(e) **Computational Conversion**. A structure \( S_1 \) is said to be "computationally" convertible to \( S_2 \) iff property (P1) is satisfied.

Computational conversion is indeed weak. For the reduction of \( S_1 \) to \( S_2 \) we only require that the two structures compute the same function. The introduction of new predicates, actions, or variables are all allowed.
3.2 The Notions of Reducibility and Equivalence

The issue of "relative power" of various classes of control structures can now be precisely stated. Given a set of conversion rules, a class of structures $C_1$ is said to be

(a) **reducible** to a class $C_2$ (notationally $C_1 \preceq C_2$) if every structure in $C_1$ can be converted to a structure in $C_2$, but not necessarily vice versa.

(b) **strictly reducible** to a class $C_2$ (notationally $C_1 < C_2$) if every structure in $C_1$ can be converted to a structure in $C_2$, but not vice versa.

(c) **equivalent** to a class $C_2$ (notationally $C_1 \equiv C_2$) if every structure in $C_1$ can be converted to a structure in $C_2$ and vice versa.

Given a set of conversion rules from $C_1$ to $C_2$, reduction intuitively implies that $C_1$ is "less powerful" than $C_2$, strict reduction implies that $C_1$ is "strictly less powerful" than $C_2", and equivalence implies that $C_1$ and $C_2$ are "equally powerful."

Given the notions of:

(a) very strong conversion
(b) node splitting conversion
(c) strong conversion
(d) semantic conversion
(e) computational conversion

we shall denote the "reduction", "strict reduction", or "equivalence" two classes of structures by

(a) $\preceq_{vs}$, $\prec_{vs}$, and $\equiv_{vs}$

(b) $\preceq_{ns}$, $\prec_{ns}$, and $\equiv_{ns}$

(c) $\preceq_s$, $\prec_s$, and $\equiv_s$

(d) $\preceq_{sem}$, $\prec_{sem}$, and $\equiv_{sem}$

(e) $\preceq_c$, $\prec_c$, and $\equiv_c$
IV RESULTS ON CONTROL STRUCTURES

This section reviews several major results on control structures and discusses their practical significance. These results fall into two main categories:

1. The classic result of Bohm and Jacopini [Bl] on the theoretical completeness of D-structures.

2. The results of Kosaraju [K4], which place all of the control structures given earlier into a hierarchy under semantic conversion.

4.1 The Bohm and Jacopini Result

The classic result of Bohm and Jacopini on the theoretical completeness of D-structures was perhaps the first major (albeit initially little recognized) result in structured programming. This result is well-described in a paper by Mills [M1].

Briefly stated, the Bohm and Jacopini paper [Bl] makes the following points:

(a) \( D \equiv L \), i.e., any L-structure (including those permitting arbitrary transfer of control) can be converted to a computationally equivalent D-structure.

(b) In the computational conversion of an L-structure to a D-structure, boolean control variables may be introduced, but the values may be stored in a stack and only the value of the top element in the stack need be known at any given point in the program.

The importance of result (a) was the establishment that the "goto" statement is, at least theoretically, not needed to perform computations, and that three simple but familiar control structures: sequential composition, if-then-else, and while-do, are in fact theoretically complete control structures.
1: if A[i] < x
    then if L[i] = 0
        then L[i] := m
        else begin L := L[i];
            goto 1
            end
    else if R[i] = 0
        then R[i] := m
        else begin i := R[i]
            goto 1
            end;

A[m] := x

Figure (6a) A control structure not reducible to a D-structure
without new variables or predicates

v := false;

while ¬v do
    if A[i] < x
        then if L[i] = 0
            then begin L[i] := m;
                v := true
                end
        else i := L[i];
    else if R[i] = 0
        then begin R[i] := m;
            v := true
            end
        else i := R[i];

A[m] := x

Figure (6b) Use of a new variable to reduce the control structure
of (6a) to a D-structure
An example of the conversion of an L-structure into a D-structure is given in Figure (6). This example comes from a tree searching and insertion program of Knuth [K2]. The index \( m \) denotes the array location into which a variable \( x \) is to be inserted. The arrays \( L \) and \( R \) denote the left and right branches of a tree organization that is superimposed on an array \( A \). Figure (6a) depicts a program that contains two transfers of control back to the structure entry point. Figure (6b) shows a computationally equivalent D-structure. The conversion employs an intermediate boolean variable whose value is checked at the end of a while-do structure. Two comments are in order here. First, the D-structure of Figure (6b) is not necessarily less understandable than the L-structure of Figure (6a). Second, as we shall discuss later, the L-structure of Figure (6a) can be nicely expressed without goto's using an REC1-control structure.

The question arises, under what conditions is a control structure convertible to a D-structure, i.e. without introducing new boolean variables or changing the particular semantics of a program. The answer [K4] lies in the detection of two loops with two or more distinct exits. In general, an L-structure is convertible to a D-structure under semantic conversion if and only if the structure does not contain a traceable loop with two distinct exits. If a structure contains only loops with one exit, the structure is convertible to a D-structure.

For example, consider the program schema of Figure (7), taken from a program in Gross and Brainerd [G1]. This is a typical structure that cannot be converted to a D-structure without new variables or actions. Here we have a loop consisting of the sequence \( a_3 p_1 p_2 a_4 p_3 a_5 \) with two exits, one through \( a_7 \) and one through \( a_8 \). Note that the branch to \( a_6 \) is not an exit from this loop since the flow of control must return to \( a_3 \). Similar arguments hold for the structure of Figure (6a).
Figure (7) A "typical" structure that is not convertible to a D-structure without new variables or new actions
4.2 Kosaraju's Hierarchy of Control Structures

There have been numerous attempts to discover the limitations of D-structures as well as to explore the expressive power of other control structures. Knuth and Floyd [K3] have shown that \( D \prec_{\text{vs}} L \) and given some intuitive ideas that \( D \prec L \). Bruno and Steiglitz [B2] have formally proved that \( D \prec_{\text{ns}} L \). Kosaraju [K4] and Peterson et al [P1] have proved that \( D \prec_{\text{sem}} L \). These results point to the fact that there are indeed fully labeled programs that cannot be converted to D-structure form without changing the length, execution time, or primitives of a given program. Peterson et al have also shown similar results for \( R E_n \) structures, i.e. \( R E_{m} \prec_{\text{ns}} L \), and \( R E_{m} \equiv_{\text{ns}} L \).

These results fail to answer one important question, namely how do the structures given earlier relate to each other. The results of Koraraju [K4] resolve this question.

The basic results of Koraraju are outlined in Figure (8). [ref [K4] and private communication]. An upwards solid line connecting one class of structures \( C_1 \) to another class \( C_2 \) means that \( C_1 \) is strictly reducible to \( C_2 \) under the notion of semantic conversion, i.e., \( C_1 \prec_{\text{sem}} C_2 \). An upwards dotted line means that \( C_1 \leq_{\text{sem}} C_2 \). The main results defined in Figure 9 are summarized as follows. Under the notion of semantic conversion,

1. for \( m \leq n \), there exist \( B J_m \)-structures which cannot be converted to \( B J_m \)-structures. In particular, without the introduction of new predicates or actions, \( B J_m \)-structures are "more powerful" than \( B J_1 \)-structures, which are identical to D-structures.
Figure (8) Kosaraju's Hierarchy of Control Structures under "Semantic" conversion.
(2) $RE_n$-structures are "more powerful" than BJ $n$-structures or D-structures.

(3) It is believed that $RE_n$-structures are equivalent to REC $n$-structures (as yet, this is an unproven conjecture). Somewhat surprisingly, it is believed the addition of the cycle(1) command does not add theoretical power to the repeat-exit control structure under semantic conversion.

(4) DRE $n$-structures are more powerful than RE $n$-structures. Again somewhat surprisingly, the addition of a while-do control structure does in fact add theoretical power to the RE $n$-control structure.

(5) Finally, if no a priori bound is placed on the index $n$, any fully labelled structure is semantically convertible to an RE $n$, REC $n$, or DRE $n$ structure. Other results not shown in Figure (8) are given in [K4].

As an example illustrating this hierarchy, consider the structure of Figure (9a), which is based on the control structure recently proposed by Zahn [Z1, K2]. This control structure represents a computation where a computation sequence is to be repeated until one of a number of "events" occurs. Upon realization of one of the events, the repeated loop is exited. Termination of the loop then invokes a specific computation determined by the event that has actually occurred. This control situation is a fairly natural one, and is quite close to a BJ $n$-structure.

Noting that $D <_{sem} BJ_n$, the conversion of this structure to a D-structure requires a new variable, as shown by the program in Figure (9b). On the other hand, noting that $BJ_n <_{sem} RE_1$, this structure, can be nicely converted to an RE $1$-structure, which is given in Figure (9c).

As another example of the utility of RE $1$-structure, consider the tree searching and insertion of the program of Figure 6. This program can be readily converted to an RE $1$-structure, as shown in Figure (10).

From a programmer's viewpoint, the results given above suggest that there is some question over the practical utility of programming with only D or D'-structures. Aside from questions of efficiency, the examples also suggest that the use of stronger control structures like RE-structures and their variants may obviate the need for goto's. In the next section I present a key example that, in fact, presents evidence counter to these suggestions.
Figure (9a) The Control Structure proposed by Zahn

The diagram shows a control structure with decision points and event assignments. The structure is as follows:

- $a_1$ → $p_1$ → $b_1$ → $\text{event} := i_1$
- $a_2$ → $p_2$ → $b_2$ → $\text{event} := i_2$
- $\ldots$
- $a_m$ → $p_m$ → $b_m$ → $\text{event} := i_m$

In the right part of the figure, there is a case statement:

```
case (event) of
  c_1
  c_2
  \ldots
  c_n
end
```

The conditions $i_1, i_2, \ldots, i_m$ are involved in the decision points $p_1, p_2, \ldots, p_m$.

The diagram represents a hierarchical control structure where decisions are made based on events, leading to different paths through the process.
v := false
while ¬ v do
    begin
        v := true;
        a₁ if p₁
            then begin b₁; event := i₁
                  end
        else begin
            a₂ if p₂
                then begin b₂; event := i₂
                      end
            else
                begin
                aₙ; if pₙ
                    then begin bₙ; event := iₙ
                          end
                    else begin
                        aₙ₊₁; v := false
                    end
                end
            end;

        case (event) of
            1: c₁;
            2: c₂;
            n: cₙ
        end

Figure (9b) Zahn’s Control Structure expressed as a D-structure (under computational conversion)
repeat $a_1$;
  if $p_1$ then begin $b_1$;
    event := $i_1$;
    exit (1)
  end;

$a_2$;
  if $p_2$ then begin $b_2$;
    event := $i_2$;
    exit (1)
  end;

  ...

$a_m$;
  if $p_m$ then begin $b_m$;
    event := $i_m$;
    exit (1)
  end;

$a_{m+1}$
end

case (event) of
  1: $c_1$;
  2: $c_2$;
  ...
  $n$: $c_n$
end

Figure (9 c) Zahn's Control Structure expressed as an $RE_1$ -structure
repeat
    if A[i] < x
        then if L[i] = 0
            then begin L[i] := m;
                exit (1)
            end
        else i := L[i]
    else if R[i] = 0
        then begin R[i] := m;
            exit (1)
        end
    else i := R[i]
end;
A[m] := x

Figure (10) Tree-search program of Figure (6) as an RE_1-Structure
V. AN EXAMPLE

When all is said and done, the practicing programmer is primarily interested in solving problems using some given set of control structures. Theorems and results on the conversion of one program or flowchart into another form may be of some interest, but certainly not the basic issue. This section presents an example directed at resolving this important issue. In particular, are there problems for which D or D'-structures do not provide as clear a solution as REC_n-structures. More precisely, are there problems for which there is a solution S2 ∈ REC_2 and for any reasonable solution S1 ∈ D, S1 ⊂ S2 and S2 is significantly clear than S1?

I must admit, the problem presented here was originally proposed with the hope of supporting a positive answer to the above question. The problem appeared to have the right set of ingredients, i.e. the need for cycling back to a loop entry from within a loop and the need for an escape exit nested within multiple loops.

This problem is called the "qualified name" problem. Basically, the problem is to write a program segment to set the value of a variable LEGAL_NAME to true or false according to whether a given PL/I qualified name is a legal or illegal reference. In PL/I, one can declare "structures" with nested components, e.g.

```
DECLARE 1 A,
    2 B,
    3 'C CHAR(5),
    3 D FIXED;
DECLARE 1 X,
    2 B,
    3 C FLOAT,
    3 E FLOAT;
```

A "reference" to a structure is considered legal if and only if the reference refers to one and only one declared structure component. Using the above declarations, A, A.B, A.B.C, and B.E are legal references, whereas B and B.C are illegal.

To solve this problem, a number of primitives are assumed:

(a) A linked list of entries call QUALIFIED_NAME, which represents the information about a qualified name.

(b) A function BASE_ENTRY, which when applied to a qualified name yields the base entry in the list. e.g. the base entry in the qualified name A.B.C is the entry for C.
(c) A linked list of entries called SYMBOL_TABLE, which contains entries for each identifier declared in a program.

(d) A function NEXT, which when applied to a null symbol table entry gives the first entry, and which when applied to a non-null symbol table entry gives the next entry in the symbol table (assuming some pre-determined order).

(e) A function FATHER, which when applied to a qualified name entry or symbol table entry, yields the next higher-order entry in the corresponding qualified name or symbol table, or the null entry if there is no father entry. For example, in the linked list for A.B.C, the father of the entry for C is the entry for B, and the father of the entry for A is the null entry.

(f) A function NAME, which when applied to a qualified name entry or a symbol table entry yields the identifier for that entry.

A solution to this problem using REC₂-structures is given in Figure (11a). Here, the PASCAL case statement is extended to allow for multiple case conditions. This solution is quite clear and makes liberal use of cycles and exits. The conversion of this solution to a D or D'-structure under restricted conditions (e.g., any of the properties P2 through P5) is highly tedious exercise, resulting in a much longer and less efficient solution. Nevertheless, a new (computationally equivalent) solution using j-structures can be devised, as given in Figure (21b). This solution compares quite favorably with the solution using REC₂-structures.

It is important to comment here that, in addition to the formal results presented earlier, that there have been numerous other papers [F1,L1,L1,W4,Z1] suggesting the limitations of D or D'-structures. As far as I can perceive, most of these papers only compare the conversion of abstracted program schemas or flowcharts into D or D'-structure form. Not once have I seen a problem that really shows the limitations on clarity with D or D'-structures. The classic case of the abnormal exit from some deeply nested procedure just does not hold weight, for the notion of
passing control to a procedure that does not return control to the calling pro-
gram segment is counter to the very notion of 1-in, 1-out control structures.

I have long supported the view that D and D'structures are not sufficient
for the practicing programmer. Recently I have tried to support this opinion with
example problems far too numerous to mention here. Frankly, I have not found such
an example problem.
QN_ENTRY := BASE_ENTRY(QUALIFIED_NAME);
BASE_ID := NAME(QN_ENTRY);
ST_ENTRY := NULL;
DIRECT_HIT := false
NUM_PARTIAL_HITS := 0;

repeat
ST_ENTRY := NEXT(ST_ENTRY);

case (ST_ENTRY=NULL, NAME(ST_ENTRY)=BASE_ID) of
  (T,F): exit (1)
  (F,F): cycle (1)
  (T,T): {cannot occur}
  (F,T): begin
    LOCAL_QN_ENTRY := FATHER(QN_ENTRY);
    LOCAL_ST_ENTRY := FATHER(ST_ENTRY);
    SKIP := false;
    repeat
      case (LOCAL_QN_ENTRY=NULL, LOCAL_ST_ENTRY=NULL) of
        (T,T): if SKIP then NUM_PARTIAL_HITS := NUM_PARTIAL_HITS + 1 else begin DIRECT_HIT := true; exit(2) end
        (T,F): NUM_PARTIAL_HITS := NUM_PARTIAL_HITS + 1
        (F,T): {no operation}
        (F,F): begin
          if NAME (LOCAL_QN_ENTRY)=NAME(LOCAL_ST_ENTRY) then LOCAL_QN_ENTRY := FATHER(LOCAL_QN_ENTRY)
          else SKIP := true;
          LOCAL_ST_ENTRY := FATHER(LOCAL_ST_ENTRY);
          cycle(1)
        end {case};
      cycle(2)
    end {repeat}
  end {case}
end {case}
end;

if DIRECT_HIT v (NUM_PARTIAL_HITS=1) then LEGAL_NAME := true
else LEGAL_NAME := false

Figure (11a) A Solution to the Qualified Name Problem as an REC2-Structure
QN ENTRY := BASE_ENTRY(QUALIFIED_NAME);
BASE_ID := NAME(QN_ENTRY);
ST_ENTRY := NEXT(NULL);
DIRECT_HIT := false;
NUM_PARTIAL_HITS := 0;

while (ST_ENTRY ≠ NULL) ∧ (DIRECT_HIT) do
begin
    if NAME (ST_ENTRY) = BASE_ID
        then begin
            LOCAL_QN_ENTRY := FATHER (QN_ENTRY);
            LOCAL_ST_ENTRY := FATHER (ST_ENTRY);
            SKIP := false;

            while (LOCAL_QN_ENTRY ≠ NULL) ∧ (LOCAL_ST_ENTRY ≠ NULL) do
                begin
                    if NAME (LOCAL_QN_ENTRY) := NAME (LOCAL_ST_ENTRY)
                        then LOCAL_QN_ENTRY := FATHER (LOCAL_QN_ENTRY)
                        else SKIP := true;

                    LOCAL_ST_ENTRY := FATHER (LOCAL_ST_ENTRY);
                end;

            case (LOCAL_QN_ENTRY = NULL, LOCAL_ST_ENTRY = NULL) of
                (T,T): if SKIP
                    then NUM_PARTIAL_HITS := NUM_PARTIAL_HITS + 1
                    else DIRECT_HIT := true
                (T,F): NUM_PARTIAL_HITS := NUM_PARTIAL_HITS + 1
                (F,T): {no operation}
                (F,F): {cannot occur}
            end {case}
        end {begin};

    ST_ENTRY := NEXT (ST_ENTRY);
end;

if DIRECT_HIT ∧ (NUM_PARTIAL_HITS =1)
    then LEGAL_NAME := true
    else LEGAL_NAME := false

Figure (11b) A Solution to the Qualified Name Problem as a D'-Structure
VI. CONCLUSIONS

There are three basic conclusions of this paper.

(1) From a programmer's viewpoint, results relating to the conversion of one program form to another form under restricted conversion rules are mainly of theoretical interest only.

(2) The utility of the goto, as well as other higher (non D or D') control structures, is seriously questioned.

(3) The utility of D and D'-structures is supported.

My first conclusion may be difficult to accept, for there have been numerous formal results (presented here and elsewhere) on the limitations of control structures under various notions of conversion. It is tempting to conclude from these results that the practicing programmer would be unduly limited with the control structures that did not hold up well under conversion. As mentioned earlier, the practicing programmer is hardly ever concerned with converting programs from one form into another. My contention is that formal results on conversion provide little real support for the practical use of any particular control structure.

My second conclusion agrees with the views of Mills [M1] and others. I have found no evidence for retaining the goto statement. The recent work of Knuth [K2] surveys many opinions on the use of various control structures, including the goto. However, I strongly believe the arguments that he advances in favor of the goto, clarity and efficiency, are not supported.

The argument from clarity is exemplified by "Sometimes it is necessary to exit from several levels ... and the most graceful way to do this is a direct approach via the goto or its equivalent." [K2, p. 18] Knuth discusses eight example problems and points out the virtues of several solutions that use the goto. In my opinion, not one of these solutions is clearer than the solutions without the goto statements. Consider, for example, the programs in Figures (6a) and (6b) which were derived from the "tree searching" examples
of Knuth. The solution using the goto statement (6a) is not obviously clearer than the D-structure one in (6b). Furthermore, changing the name of the boolean variable "v" to a more descriptive one, e.g. "empty space found", makes the debate almost vacuous. Clarity is, of course, a highly subjective quality, but I believe that a thoughtful reading of these examples will support my contention.

The argument from efficiency, that the goto is less time consuming than alternative control structures, is frequently made. Knuth, for example, says "Sooner or later people are going to find that their beautifully structured programs are running at only half speed...[K2, p. 3] He does present several example programs where a solution with goto statements is indeed more efficient than solutions with alternative control structures (though a factor of two is never obtained). Nevertheless, it is my basic contention that all such example programs would be just as efficient if processed by a good optimizing compiler. Certainly, no optimizing compiler can be expected to perform "macro-efficient" optimizations like the conversion of a linear search into a binary one. On the other hand, redundant tests and repeated actions are typical of the "micro-efficient" conditions that can be eliminated by good optimizing compilers, rare though they may be. This latter type of optimization should not be the responsibility of the typical programmer, who should be primarily interested in developing clear, macro-efficient programs.

Similarly, the same clarity and efficiency arguments do not support to any great degree the multiple-exit control structures, like that proposed by Zahn [Z1, K2]. Furthermore, these structures do not appear to provide a more "natural" way of thinking about the problem.
My third conclusion relates to the utility of $D'$-structures over $D$-structures. Readers may have observed the use of case, if-then, and repeat-until structures (all of which are $D'$-structures) in the solution to the qualified name problem. From the results presented earlier, the only real differences between $D$ and $D'$-structures is notational convenience. For example, the use of case structures can often prevent the need for multiple nested if-then-else structures, and the use of repeat-until structures can often prevent the use of somewhat artificial while-do structures. Since $D'$-structures preserve the important 1-in, 1-out property of $D$-structures, the notational convenience provided by $D'$-structures is strongly recommended.

In parting, I must admit that any recommendation for a good set of control structures is indeed subjective. However, I must conclude from this examination that considerable new and definitive evidence is needed before we suggest that $D$ or $D'$ control structures, with all their clarity and simplicity, are not sufficient for the practicing programmer.

Acknowledgements

I am very grateful to Rao Kosaraju, whose mastery of the theoretical results in structured programming is impeccable, and to Michael Marcotty, who contributed many new insights to this paper and who suggested the PL/I qualified name problem as a challenge to $D'$-structures.
REFERENCES

A1. E. Ashcroft and Z. Manna
   The Translation of 'GOTO' Programs to 'WHILE' Programs

B1. C. Bohm and G. Jacopini
   "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules"

B2. J. Bruno and K. Steiglitz
   "The Expression of Algorithms by Charts"

D1. E.W. Dijkstra
   "Notes on Structured Programming"

F1. D. Friedman and S. Shapiro
   "A Case for the While - Until"

G1. J.L. Gross and W.S. Brainerd
   Fundamental Programming Concepts

H1. P. Henderson and R. Snowdown
   "An Experiment in Structured Programming"
   BIT 12, 38-53, 1972

H2. C.A.R. Hoare and N. Wirth
   An Axiomatic Definition of the Programming Language PASCAL
   Proc. Symposium on Theoretical Programming, Novosibirsk, August 1972

K1. B.W. Kernighan and P.J. Plauger
   The Elements of Programming Style

K2. D.E. Knuth
   "Structured Programming with GOTO Statements"

   Notes on Avoiding GO TO Statements

K4. R. Kosaraju
   "Analysis of Structured Programs"
L1. B.M. Leavenworth
"Programming with(out) the GOTO"

L2. H.F. Ledgard
Programming Proverbs

M1. H.D. Mills
Mathematical Foundations for Structured Programming

P1. W.W. Peterson, T. Kasami, and N. Tokura
"On the Capabilities of While, Repeat, and Exit Statements"

P2. Proceedings of an ACM Conference on Proving Assertions about Programs

W1. G.M. Weinberg
The Psychology of Computer Programming

W2. N. Wirth
The Programming Language PASCAL (revised report)
Eidgenössische Technische Hochschule, Zurich, November, 1972.

W3. N. Wirth
"Program Development by Stepwise Refinement"

W4. W.A. Wulf, D.B. Ruscitti, and A.N. Habermann
"BLISS: A Language for Systems Programming"

W5. W.A. Wulf and M. Shaw
"Global Variable Considered Harmful"
SIGPLAN Notices, Volume 8, Number 2, February 1973.

Z1. C.T. Zahn
"A Control Statement for Natural Top-down Structured Programming"