Manuscript prepared for the
Second Annual Meeting of the
Gesellschaft für Informatik,
Karlsruhe, October 2-4, 1972

August 4, 1972

MOLB1.2.1. A Hierarchy of Control Structures

Eberhard Wegner
Informatik-Forschungsgruppe Programmiersprachen und Compiler II
Technische Universität Berlin, Berlin 10, Ernst-Reuter-Platz 8

Summary: Four classes of control structures for sequential programs are shown to offer increasing expressive power. They allow control to be specified by
(1) four well-chosen structures with one entry and one exit each,
(2) the one-entry loop with escapes,
(3) the distributor-entered loop with escapes, and
(4) the unrestricted jump.
It is shown that a given program scheme can be described by the one-entry loop with escapes if and only if all control paths from the start to a cycle enter that cycle at the same node (irrespective of the number of nodes where the cycle is re-entered).

1. Motivation

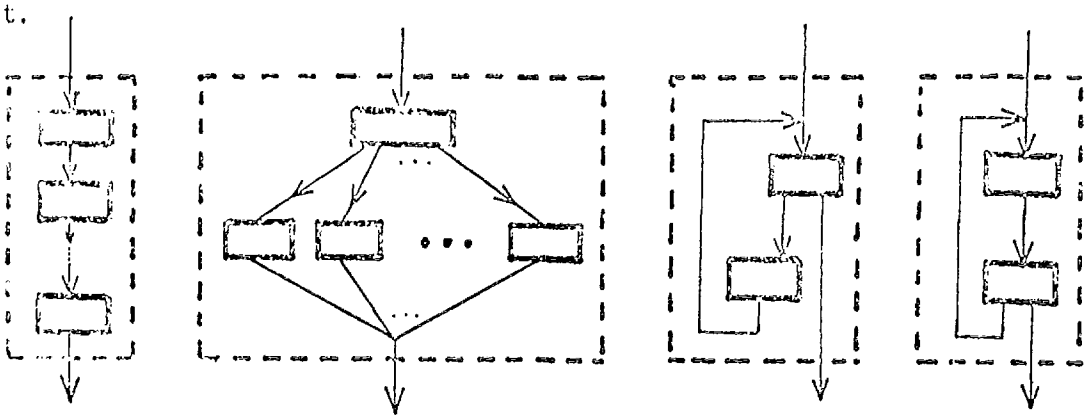
Frequently occurring control structures are represented in programming languages by special constructs in order to make programs easy to understand, to prove correct, to modify, to debug, and to optimize globally during compilation. These structures are used for the "deductive" approach to program construction by stepwise refinement (Dijkstra 1970, Wirth 1971). Programs which were originally designed in terms of general flowcharts can be "inductively" structured to fit such language constructs (Ashcroft and Manna 1971) either in order to conform to a programming language not providing the jump or simply to gain the advantages stated above.

2. Programs controlled by the unrestricted jump

Before describing special control structures, let us remember the most general form of control for sequential programs. Let a "program scheme" be a (non-empty and finite) directed graph containing a distinguished node (the "start") from which paths lead to all other nodes. From any node to any other node there leads at most one edge, and there is no reflecting edge. A "program" is a program scheme in which each node is labelled by some executable "action" which has a single entry and any number of exits. Each exit must identify an edge leaving the node. Let us depict each node by a rectangle (labelled or not) and each edge by an arrow connecting two rectangles. Edges that enter the same node are drawn to flow together before entering the node in order to indicate that there is only one entry to its action. The start is marked by an entering edge which leaves no node. A figure representing a part of a program scheme will contain edges entering or leaving that part.

3. The four structures with one entry and one exit each

Dijkstra (1970) and Wirth (1971) use four structures for decomposing actions into subactions: the concatenation, the selection (including two special cases), the pre-checked loop, and the post-checked loop. Each subaction is either a test which cannot be further decomposed or a piece of program with any number of internal paths but only one exit.



Ashcroft and Manna (1971) give an algorithm to express any flowchart program by some of these control structures (where control variables and branches must be added if and only if the flowchart contains two or more tests and at least one loop that are not contained in inner blocks). These structures meet the objectives in section 1 rather well, but occasionally the programmer is burdened with joining several control paths into the single exit and separating them immediately afterwards. Knuth and Floyd (1971) show that two familiar programs, a table search (a loop with two exits) and a backtracking program (two crossing loops), are more naturally controlled by jumps than expressed in terms of these structures by the addition of some actions.

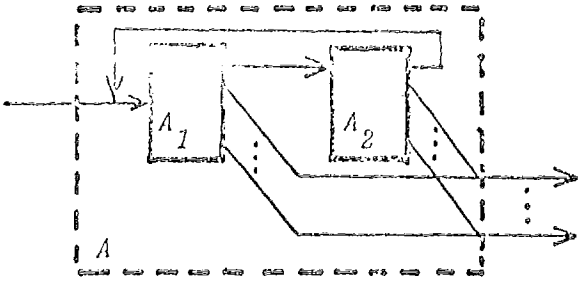
4. Programs controlled by the one-entry loop with escapes

4.1 The jump to a labelled *end* is frequently used in ALGOL-like languages. Several languages provide special instructions to leave a control construct:

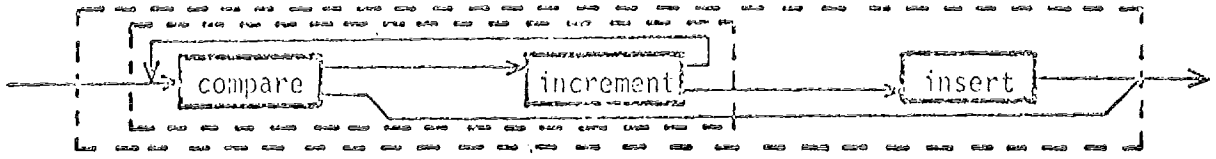
-- ALGOL 68 with its completer (Wijngaarden, Mailloux, Peck, and Koster 1969),
-- BCPL with the break command (Richards 1969), and
-- BLISS with the escape expressions (Wulf, Russell, and Habermann 1971).

Wulf (1971) claims that with the escape "the desirable properties of goto-less graphs are retained", and Clint and Hoare (1972) state that "programs which confine their jumping to genuine returns and exits do not present any great difficulty in the proof of their correctness". Since such an exit jump may terminate the execution of any enclosing control construct, we will not adopt the condition that each control structure has a single exit.

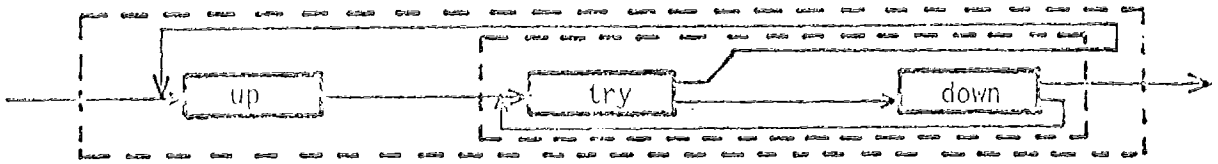
4.2 Definitely the most primitive decomposition of an action A is the decomposition into two subactions A_1 and A_2. The node labelled by A is replaced with two nodes labelled by A_1 and A_2. The definitions in section 2 imply the following rules for the "binary decomposition": The single entry to the action A will transform into the entry to one of the subactions, say A_1. Consequently, all edges entering node A will enter A_1, and if node A is the start, then A_1 will be the start. A "forward" edge from A_1 to A_2 must be introduced so that each node can still be reached from the start. A "backward" edge from A_2 to A_1 may be added. Each edge leaving A must be replaced by an edge leaving A_1 or by an edge leaving A_2 or by both edges. Given an arbitrary number of exits from A, the following figure shows the resulting "one-entry loop with escapes" including all possible control paths:



4.3 Conversely, in a given program scheme or program two nodes forming a one-entry loop with escapes may be found and combined into a single node. This "binary composition" is illustrated here for the two programs discussed by Knuth and Floyd:



and



4.4 Theorem: A given program scheme can be reduced to a single node by repeated binary composition if and only if all paths from the start to a cycle enter that cycle at the same node.

Corollary: A given program in linear representation (in which each transfer of control is understood as a jump) can be reduced to a single action by repeated binary composition if there is no forward jump to a point between a backward jump and its target. (The inverse assertion does not hold since there may be e.g. a jump out of a loop and a forward jump back into it.)

4.5 As a preparation for the proof of the theorem, we give an algorithm to check a program scheme for the condition stated and to mark each edge either as forward or as backward. (The edges marked as forward will impose a lattice structure on the set of nodes.) We will use a notation based on ALGOL 68.

```
proc can be reduced = (programscheme p) bool:
begin
   proc mark leaving edges = (node node):
   begin mark node as active;
      for each edge leaving node do
      if the node entered by edge is marked active
      then mark edge as backward;
         if edge is marked forward then goto false fi
      else mark edge as forward;
         if edge is marked backward then goto false fi;
         mark leaving edges (the node entered by edge)
      fi;
      mark node as not active
   end;
   mark all edges as not forward and not backward;
   mark all nodes as not active;
   mark leaving edges (start of p); true exit
   false: false
end
```

4.6 The "only if" assertion of the theorem is obvious. As a proof for the "if" assertion we give an algorithm to accept a program scheme which has been successfully marked by the procedure *can be reduced* in 4.5 and to deliver a "composition tree", that is an ordered tree in which the terminal nodes are the nodes of the given program scheme and in which the terminal nodes of each subtree can be reduced to a single node by repetitive binary composition.

proc dependent tree = (node node) tree: co assume a global *proc composition = (tree t1, t2) tree: c* the ordered tree having *t1* and *t2* as its direct subtrees *c;* assume a global program scheme *p* which has been successfully marked; accept as parameter a node *node* of *p* and deliver a subtree of a decomposition tree of *p* which contains as its terminal nodes exactly those nodes *n* of *p* for which each path from the start to *n* contains *node;* if called with the start of *p* as parameter, this procedure will deliver a composition tree of *p co*

```
begin node n; tree t:= node;
   while there is a node n such that each forward edge entering n leaves a terminal
   node of t
   do t:= composition (t, if n is entered by a backward edge then dependent tree (n)
   else n fi); t
end
```

4.7 A possible representation of the one-entry loop with escapes is (in semi-precise explanation, related to the definition of ALGOL 68) *rep range r; s_1; s_2 per* where *r* is a "range identifier" defined by the "range declaration" *range r*, the pair *rep* ...

per encloses the two direct subactions s_1 and s_2 to be executed *rep*eatedly, and each s_i may contain conditional escapes of the form *if b then v leave r fi*; here *b* is a boolean expression and *v* yields the value to be delivered by the action if its execution is completed by the conditional escape. (Note how this escape can replace the jumps and the completer in the procedure *can be reduced* above.) For convenience we introduce some contractions:

-- *if true then v leave r fi* may be replaced by *v leave r*.

-- *skip leave* may be replaced by *leave*.

-- *leave r*, where *r* identifies the directly containing range, may be replaced by *leave* if simultaneously the declaration of *r* is removed.

The following contraction allows to represent a one-entry loop with any number of constituents and escapes:

-- *rep rep ...; leave per; ... per* may be replaced by *rep ...; ... per*.

In order to obtain representations of the four one-exit structures, we give some more syntactic sugar:

-- *rep .... leave per* may be replaced by *begin .... end*.

-- *rep if not* b_1 *then* v_1 *leave fi; s; if not* b_2 *then* v_2 *leave fi per* may be replaced by v_1 *while* b_1 *rep s per* v_2 *while* b_2.

-- *skip while true* may be omitted.

-- *begin range r; begin if b then leave fi;* s_2 *leave r end;* s_1 *end* may be replaced by *if b then* s_1 *else* s_2 *fi*.

-- *else skip fi* may be replaced by *fi*.

-- *if i=1 then* s_1 *else s fi* may be replaced by *case i in* s_1 *out s esac*.

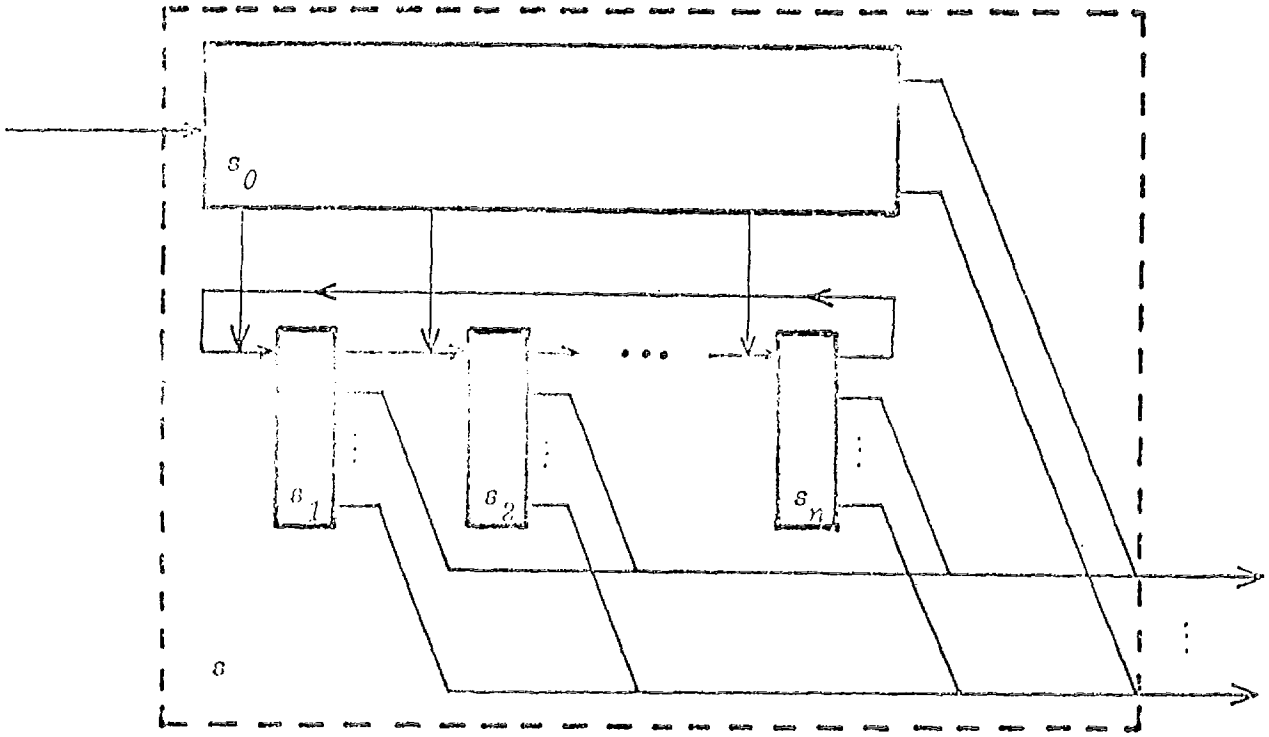
-- *if i=n then* s_n *else case i in* s_1, ..., s_{n-1} *out s esac fi* for $n \geq 2$ may be replaced by *case i in* s_1, ..., s_n *out s esac*.

4.8 The escape from any enclosing control construct is provided in BLISS. The BCPL break command is restricted to escape from the smallest enclosing loop. The ALGOL 68 completer causes completion of the elaboration of the directly surrounding serial-clause but not of a repetitive statement; it can be generalized to complete the elaboration of any containing serial-clause (Wegner 1972). The BLISS escape must and the BCPL break cannot deliver a value, while the ALGOL 68 completer may.

4.9 As Wulf (1971) reports, the escape allows to program conveniently without the jump. The contractions in 4.7 show that the one-entry loop with escapes is at least as powerful as the collection of one-exit structures. Indeed, it is even more powerful since it can describe the two examples of Knuth and Floyd as well as the program that Ashcroft and Manna constructed to demonstrate the need for additional control variables. However, it is less powerful than the unrestricted jump, since the following program cannot be reduced to a single node by binary composition:

5. The distributor-entered loop with escapes

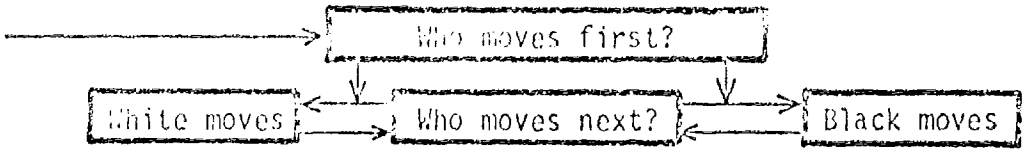
5.1 The fact that the simple game in 4.9 cannot be described by the one-entry loop with escapes gives rise to an afterthought. Here is a more powerful and complicated control structure, a loop preceded by a distributor to its constituents:



5.2 By a slight extension to the argument in 4.5 and 4.6, a given program scheme can be reduced to a single node by repeatedly combining nodes into such a "distributor-entered loop with escapes" if and only if all paths from the start to a cycle enter that cycle by edges that leave the same node.

5.3 The cases where the distributor s_0 delivers an integer i or a boolean b lend themselves to the representations *at* i *enterloop* s_1, ..., s_n *out* s_{n+1} *endloop* (where the *out* s_{n+1} is easily built around the basic construct) and *on* b *enterloop* s_1 *otherwise* s_2 *endloop*. This latter construct effectively allows to transfer control between the *then* and the *else* clauses of an *if* construct. The general construct might read *performing* s_0 *enterloop* s_1, ..., s_n *endloop* where s_0 may contain *enter* statements identifying a loop constituent, and each s_i for $i = 0, 1, \ldots, n$ may contain escapes.

5.4 The one-entry loop with escapes is a special case of this structure; let s_0 be *enter 1*. Here is a program which cannot be expressed even by the distributor-entered loop with escapes:



References

E. Ashcroft and Z. Manna, The translation of "go to" programs to "while" programs,
IFIP Congress, Ljubljana, preprint TA-2, 147-152, August 1971.

M. Clint and C.A.R. Hoare, Program proving: jumps and functions,
Acta informatica 1,214-224, 1972.

E.W. Dijkstra, Notes on structured programming,
Report 70-Wsk-03, Technical University Eindhoven, April 1970.

D.E. Knuth and R.W. Floyd, Notes on avoiding "go to" statements,
Information processing letters 1, 23-31, February 1971.

M. Richards, BCPL: A tool for compiler writing and systems programming,
AFIPS SJCC 34, 557-566, 1969.

E. Wegner, A generalized completer for ALGOL 68,
ALGOL Bulletin, to be published.

A.van Wijngaarden, B.J. Mailloux, J.E.L. Peck, and C.H.A. Koster,
Report on the algorithmic language ALGOL 68,
Numerische Mathematik 14, 79-218, 1969.

N. Wirth, Program development by stepwise refinement,
Comm. ACM 14, 221-227, April 1971.

W.A. Wulf, Programming without the goto,
IFIP Congress, Ljubljana, preprint TA-3, 84-88, August 1971.

W.A Wulf, D.B. Russell, and A.N. Habermann, BLISS: A language for systems programming,
Comm. ACM 14, 780-790, December 1971.

Editors note:

This paper is to be included in the Springer lecture notes on operations research and mathematical systems. It is printed here so that the MOLB readership need not suffer through the lead-time of conventional publication.