

**UC Irvine**  
**ICS Technical Reports**

**Title**

Irreducible Flowcharts

**Permalink**

<https://escholarship.org/uc/item/3zr5k6vv>

**Author**

Climenson, W. Douglas

**Publication Date**

1973-04-01

Peer reviewed

Irreducible Flowcharts

W. Douglas Clinenson

TECHNICAL REPORT #35 - APRIL 1973

## ABSTRACT

A schema is defined for characterizing flowchart programs. This schema is used to construct a procedure for generating the set of flowcharts which contain  $n$  branching tests but which are irreducible; i. e., have no embedded flowcharts. The sets of such flowcharts for  $n \leq 4$  are enumerated here. Some examples from these sets are discussed. Suggestions are made concerning the utility of these flowcharts in the analysis of goto-less programming and in gaining insight into potential programming language constructs. A proof of the procedure for generating irreducible flowcharts is given.

## PREFACE

The investigation that led to this report was prompted by the notion that the concepts of block structured, goto-less programming could be used in devising an automatic flowchart drawing scheme superior to the familiar ones such as AUTOFLOW, a software product of Applied Data Research. In drawing flowcharts in the usual way on normal size pages, the goto spaghetti in a program must be cut in many places. Following a path through the program requires shuffling through several pages. The question arose: could the flowchart of an existing program with many goto's be made more understandable if it were drawn so that each page would have one input arrow and one output arrow? The first page would be an attempt to characterize the entire program. If all of the statements could not fit on this page a search would be made for an embedded flowchart; that is a block of statements which had one input and one output (but perhaps with internal jumps). The embedded flowchart would be extracted and placed on some other page and a page reference substituted for it on the first page. If the program still did not fit, another search for an embedded flowchart would be made and so on. The procedure would be recursive; the embedded flowchart would be examined to see if it could fit on one page, etc.

The obvious difficulty is that there might be no set of embedded flowcharts which, if extracted, would allow the remainder of the program to fit on one page. In such cases an analysis of the program's control structure would be made to find some way of translating a portion of this structure into an embedded flowchart.

Assuming this could be done, would the resulting structure be more understandable than a conventional flowchart drawing? One way of looking for an answer to this question is to examine the pathological flowcharts that might arise. The emphasis of the investigation shifted to enumerating these pathological flowcharts and analyzing their characteristics. This paper addresses only this aspect of the investigation. Although the notion of structured flowcharting of programs with goto's (or if you like, indented listing of such programs) is still being investigated, the results reported here suggest that it is questionable whether a structured flowchart would make a program with goto's more understandable.

"Wulf shows how certain pathological algorithms become worse without goto's but argues that most useful algorithms don't so suffer. (I believe his argument, but I would be happier about it if it weren't true that almost all possible programs fall in the pathological class)."

- M. D. McIlroy (10).

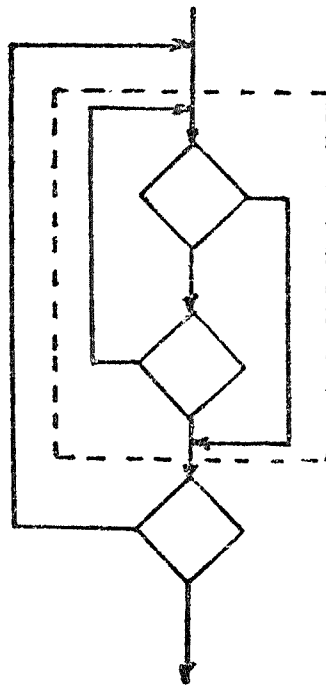
### Introduction

Since 1968, when Dijkstra (4) argued that the goto statement in programming languages is not only unnecessary but also harmful, there has been much discussion of the theory and practice of programming without the goto. It has been shown by Bohm and Jacopini (2) that the goto is not necessary; that some pair of language constructs, such as IFTHENELSE and DOWHILE, are adequate to express any desired flow of program control. The cited reasons for writing programs using only such higher level constructs are many: the programs are easier to read and understand; they fit well into a top-down approach to program design (11); they are easier to compile, optimize, and modify; they submit more readily to formal analysis such as proof of termination. Because some people have embraced these advantages to the point of advocating the elimination of the goto, the discussion of goto-less programming has taken on an adversary flavor. The session at the ACM Conference in Boston in August 1972 called "The GOTO Controversy" is the most recent example (6), (9), (14).

In such discussions, examples of programming problems are often cited to support or attack a given point of view. This paper

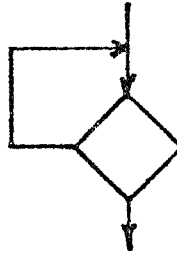
identifies and analyzes those flowchart programs which must be considered in any discussion of goto-less programming which centers on programming structure: the class of flowchart programs with  $n$  branching tests which have no flowchart programs embedded in them. (A flowchart program is a program whose paths for its control token can be represented by a connected directed graph with some input node with in-degree zero and some output node with out-degree zero, such that each node is on some path from input to output). These are the only flowchart programs we will consider. Flowcharts which have an embedded flowchart can be considered as two flowcharts: (a) the embedded flowchart, and (b) the flowchart with the embedded flowchart removed and replaced by a simple nonbranching edge.

For example, the flowchart:



(1)

has the flowchart enclosed by broken lines embedded in it, so we can treat it separately and reduce (1) to:



(2)

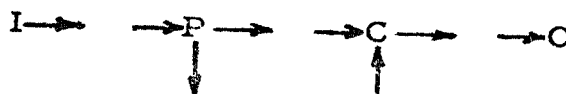
The set of flowcharts with  $n$  branching tests which have no embedded flowcharts -- hereafter called irreducible  $n$ -predicate flowcharts -- will be the subject of this paper. These are the pathological flowcharts in goto-less programming.

First a flowchart schema useful for discussion will be defined, then a complete (but redundant) procedure for generating the set of irreducible  $n$ -predicate flowchart will be described. The flowcharts generated by the procedure for  $n=1, 2, 3$  and  $4$  are given. For  $n=1$ , there are two irreducible flowcharts; for  $n=2$ , there is one; for  $n=3$ , there are eight, for  $n=4$ , there are 82. Several of these flowcharts are discussed. Finally some suggestions are made about the utility of further analysis of these flowcharts.

Flowchart schema.\*

A flowchart is considered to be a directed graph containing four types of nodes:

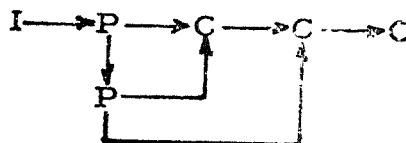
- I : input node, in-degree zero and out-degree one.
- O : output node, in-degree one and out-degree zero.
- P : branching test or predicate nodes, in-degree one and out-degree two.
- C : collector nodes, in-degree two and out-degree one.



In the examples, the horizontal branch out of a P node will be assumed if the predicate evaluates to true, and the vertical branch if false.

All program statements which change the state of the program are on the edges of the graph. In the examples, all statements on one edge are gathered in a block. It is assumed that a predicate can test any state variable and any state variable can be modified by any block of statements.

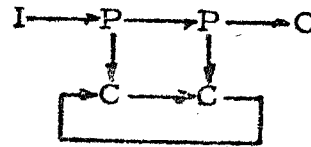
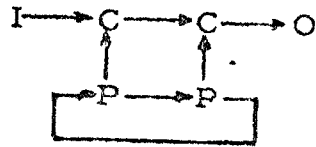
Each flowchart is connected and each C or P node in the flowchart is on some path between I and O. The following is a flowchart:



\*The schema and notation used here are nearly the same as that used by Kessler (7). Some aspects of this investigation are refinements and extensions of his comments.

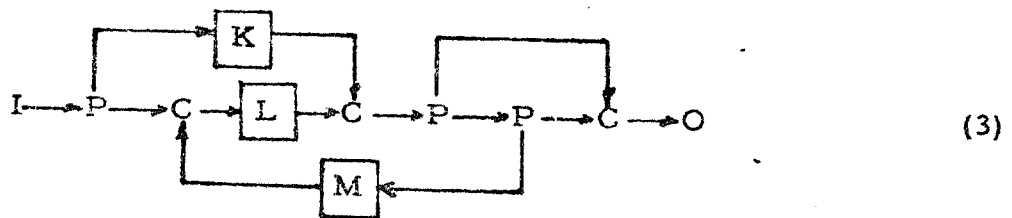


But these two directed graphs are not flowcharts:



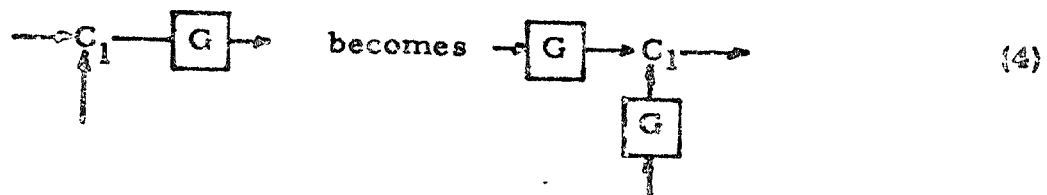
Every flowchart with  $n$  P nodes will have  $n$  C nodes and has  $(3n)+1$  edges.

A main path of a flowchart is the simple (nonlooping) path from I to O which contains the most P nodes. There could be more than one main path in a flowchart. In the examples, a main path will be drawn as a horizontal line. For example:

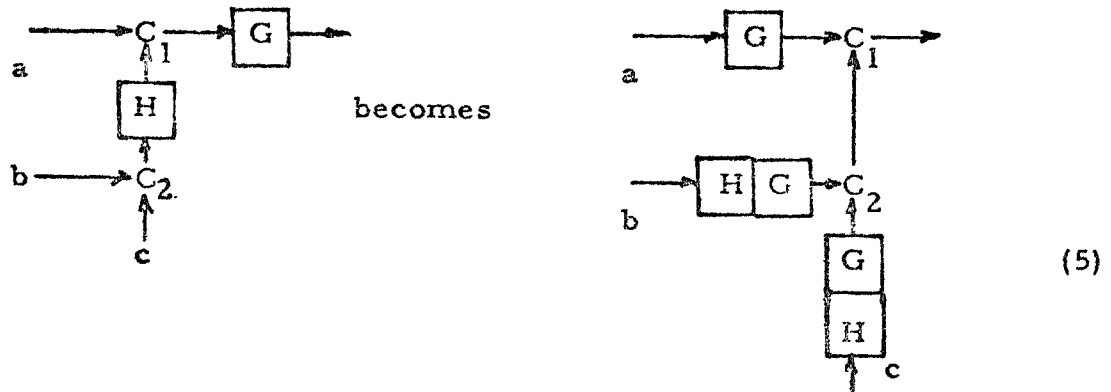


An irreducible flowchart is one with no embedded flowcharts. Two transformation rules are applied to a flowchart as part of the process of testing for irreducibility:

(a) The block (if any) on the outgoing edge of a C node is removed and replicated on each incoming edge of that C:

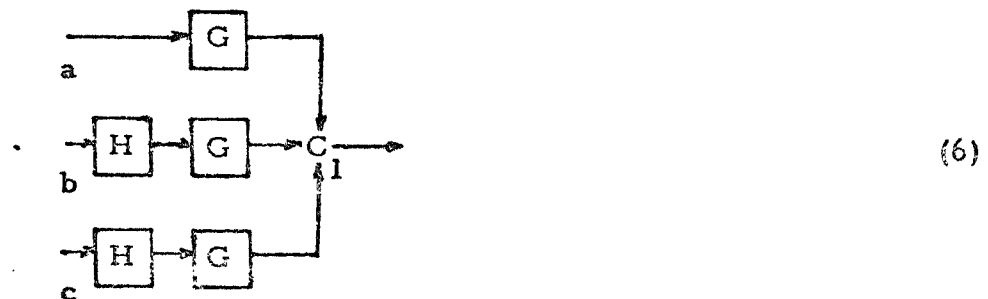


This is repeated until there are no blocks on any outgoing edges of C nodes. Thus if the flowchart has a tree of C nodes, all blocks in that tree are moved to the leaves of the tree:



and there are no blocks on any  $(C \rightarrow C)$  edges of the flowchart.

(b) Relabel the C nodes in any tree of C nodes to make them indistinct. This is equivalent to collapsing a tree of C nodes into a simple C node:

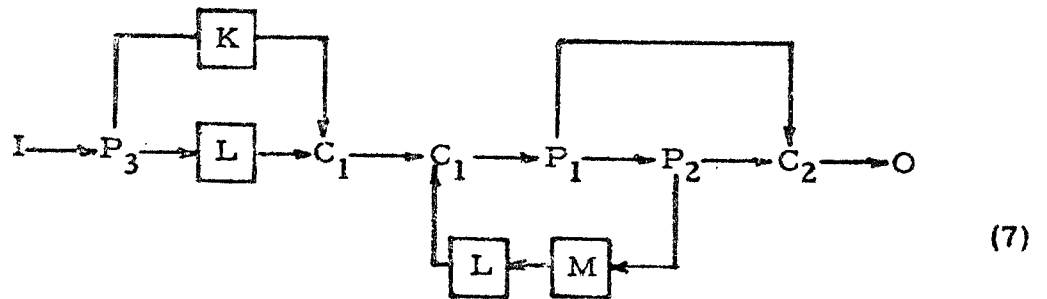


It should be clear that program equivalence is retained under these two transformations (called "node splitting" in (1)).

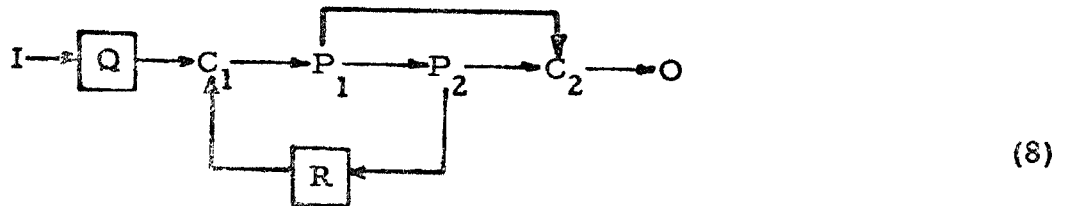
If a flowchart is reducible, that is if it contains an embedded flowchart, we can apply a third transformation rule (recursively) to make it irreducible:

(c) Replace each embedded flowchart by an edge with a block to represent it. Since embedded flowcharts could be nested, apply transformations (a) and (b) to the result and test for irreducibility. If reducible, apply transformation (c).

For example, if we apply transformations (a) and (b) to flowchart (3), we find it is reducible:



then by transformation (c) this becomes:

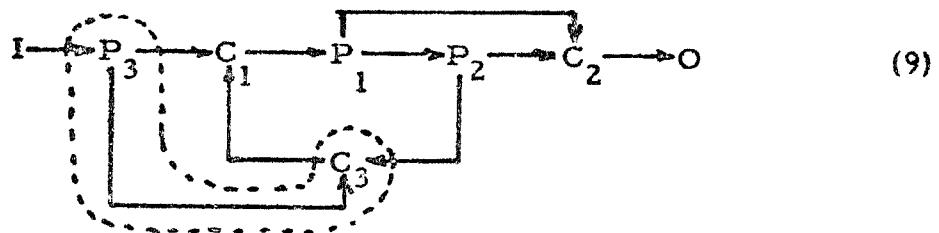


which is irreducible.

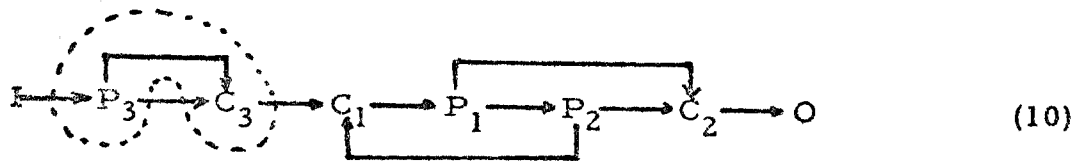
Procedure For Generating Irreducible n-Predicate Flowcharts.

We first describe a procedure for generating all n-predicate flowcharts and then show how this procedure can be used selectively to generate the set of irreducible n-predicate flowcharts.

The procedure uses a recurrence relation; that is, the set of n-predicate flowcharts is generated from the set of (n-1)-predicate flowcharts ( $n \geq 1$ ). For simplicity there are no blocks in the flowcharts. A flowchart is generated by inserting  $P_n$  and  $C_n$  nodes at cut-points  $cp_1$  and  $cp_2$  on the same or on different edges of a flowchart, and constructing an edge ( $P_n \rightarrow C_n$ ). For example, consider the 2-predicate flowchart in (8). Use the edges containing blocks Q and R as cut-points. We could then generate the 3-predicate flowchart (ignoring any blocks on the edges):



The inserted portion of the flowchart, enclosed by a dashed line, is called a PC segment. We could generate another 3-predicate flowchart from (8) by interchanging the positions of  $P_3$  and  $C_3$  (and reversing the direction of the edge). Another flowchart could be generated from (8) by inserting both  $P_3$  and  $C_3$  in the edge containing block Q:

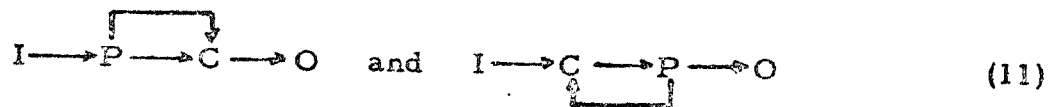


This is the same as the flowchart in (7), ignoring the blocks on the edges. Both (9) and (10) are reducible.

In general, since there are  $(3n)+1$  edges in an  $n$ -predicate flowchart, we can generate  $2(3n+1)$  flowcharts from it if we choose both cut-points to be on a given edge (all of these are reducible), and  $2 \cdot \binom{3n+1}{2}$  flowcharts if we put the two cut-points on different edges.

In Appendix B, it is shown in Lemma 1 that this procedure, when applied to the set of  $(n-1)$ -predicate flowcharts, generates at least one copy of all  $n$ -predicate flowcharts. The procedure is complete, but redundant -- obviously it generates more than one copy of some flowcharts.

We are interested only in the subset of  $n$ -predicate flowcharts which are irreducible. The above procedure could be used, along with procedures for applying transformations (a), (b), and (c), and procedures for finding isomorphic graphs. This would not be very practical. For  $n=1$ , there are two flowcharts:



for  $n=2$ , there are 16 flowcharts. These have been enumerated by

Kessler (7). For  $n=3$  .....? \*

The above procedure can be used to generate irreducible  $n$ -predicate flowcharts if we can show that they can all be generated from the subset of  $(n-1)$ -predicate flowcharts which are irreducible. This is the case, as proved in Appendix B. The proof uses the property that any irreducible  $n$ -predicate flowchart ( $n \geq 2$ ) can be generated by inserting two PC segments in some irreducible  $(n-2)$ -predicate flowchart (proved deductively as Lemma 2). This procedure, though complete, is also redundant, so additional procedures are needed to eliminate isomorphic flowcharts as well as those which are reducible.

A set of LISP functions were defined for these procedures. The irreducible 1-predicate flowcharts are the two shown in (11). Kessler (7), in enumerating all the 2-predicate flowcharts showed that only one of these is irreducible:



The LISP functions were applied to this flowchart to generate the set of irreducible 3-predicate flowcharts. There are eight of these;

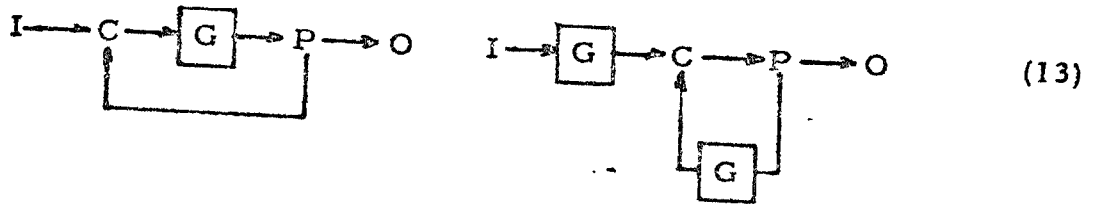
---

\*This interesting problem might be attacked by first enumerating the balanced regular (undirected) graphs with  $2n$  nodes of degree 3; that is find the number of undirected graphs with  $2n$  nodes, each of which has three other nodes connected to it. For each such graph, we could distribute  $n$  P's and  $n$  C's, cut one edge for inserting the I and O nodes, and assign directions to the edges (within the constraints of the in- and out-degree of P and C nodes).

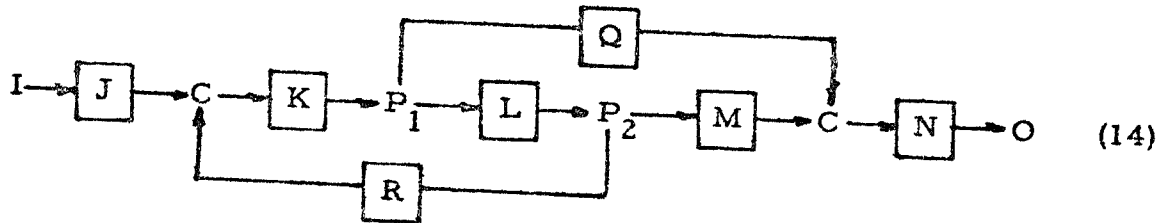
they are shown on page A-3. These eight were used in turn to generate the 82 irreducible 4-predicate flowcharts which are shown on pages A-5 to A-17, with a preceding index.

Examples of Irreducible n-Predicate Flowcharts

n=1. The two flowcharts in (11) are the familiar IFTHENELSE and DOWHILE constructs. Notice that transformation (a) permits us to transform DO G UNTIL P into G; DO G WHILE (NOT P):



n=2. The only irreducible 2-predicate flowchart is the following (drawn with a block on every edge):



This is the flowchart underlying most of the examples in the literature on goto-less programming. It is the construct used by Hopkins (6) in his example of the difficulties in goto-less programming. The specific example he uses is the problem of finding an instance of X in the vector A. If it is found, increment a value in a corresponding vector of counters and return the index of X; if an instance of X is not found, add X to A, extend the counter vector, and return the index



of X. In flowchart (14), blocks J and K would initialize and increment the index variable; P1 would test the index variable; P2 would test the current element in A to see if it was X. Blocks Q and M would modify the vector of counters appropriately, and N would assign the returned value. Without using goto's or multiple return statements, how can the search loop be exited in a way that retains information about which condition caused the exit? Kessler (7) also saw this difficulty and has suggested that because this kind of problem is so common, SEARCH might be useful as a language primitive. It would require four arguments: the vector, the value to be matched, and the blocks to be executed on success and on failure.

Flowchart (14) is also the underlying structure of the complete DOWHILE in, say, PL/I: DO i=j TO k WHILE P1; R; but only block R in the flowchart is accessible to the user. The blocks that one would like to make accessible are Q and M. This suggests the construct:

```
DO i=j TO k (THEN M) WHILE P1 (THEN Q); R;
```

which is somewhat cumbersome, but feasible.

The flowchart in (14), in fact, has been implemented as a primitive: it is embedded in the construct called treewalk in MADCAP 6 (12).

Flowchart (14) has also been used to illustrate methods for avoiding the goto or to translate a program with goto's into one without

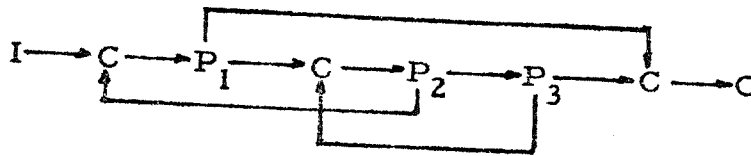
goto's. Knuth and Floyd (8) analyze this flowchart exhaustively. Ashcroft and Manna use it as an example to describe two algorithms for eliminating goto's. Their ALGORITHM 2 is illustrated in the following section on irreducible 3-predicate flowcharts.

$n=3$ . Two of the eight irreducible 3-predicate flowcharts will be used to show two methods which can be used to avoid goto statements. There are other methods, including Cooper's (3), which will not be discussed here.

The introduction of auxiliary variables is necessary to avoid goto statements in irreducible  $n$ -predicate flowcharts ( $n \geq 2$ ). This has been shown by Ashcroft and Manna (1). Methods for eliminating goto statements in such flowchart programs differ according to the way auxiliary variables are introduced and tested. Whether or not the method chosen results in a "natural" program depends on the semantics of the problem. The proponents of goto-less programming would argue that if the problem semantics are interpreted properly, the auxiliary variables are not really auxiliary variables at all -- they are the variables one would want to use if one designed the program from the top down. Problem semantics and good/bad programming are not the primary issues here; the examples are illustrative only.

We can use flowchart 3.5 from page A-3, Appendix A to illustrate some of the difficulties Hopkins (6) is concerned about if we choose to introduce auxiliary variables instead of permitting goto's.

Consider flowchart 3.5;



3.5

An example of this flowchart is the program for determining whether there is a row of X's in the array  $A(m, n)$ .  $P_1$  tests whether the last row of  $A$  has been tried.  $P_2$  tests whether the current element of  $A$  is  $X$ , and  $P_3$  tests for end of row. There are two loops in the program which must be handled by DOWHILE's, but each of these must have more than one exit. Following Hopkins' example, auxiliary variables  $SW1$  and  $SW2$  can be used to produce the goto-less PL/I program:


EXES:

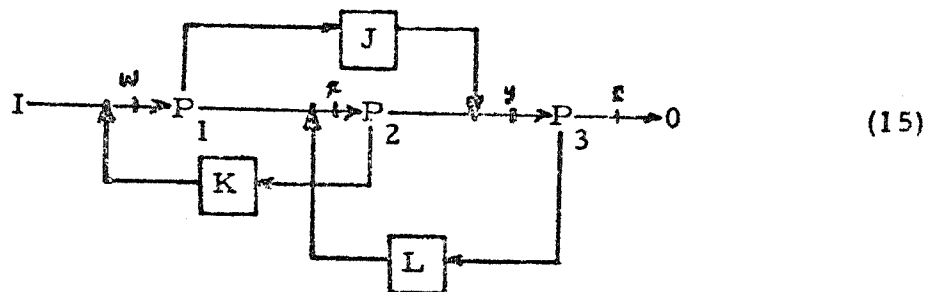
```

PROC (X);
  SW1=1;
  DO I=1 to M WHILE SW1;
    SW2=1;
    DO J=1 to N WHILE SW2=1;
      IF A(I, J) ≠ X THEN SW2=0;
    END;
    IF SW2=1 THEN SW1=0;
  END;
  IF SW1=0 THEN OUT=1;
  ELSE OUT=0;
  RETURN (OUT);
END EXES;

```

The LEAVE < proc > WITH < value > construct in BLISS (13), which is a disguised goto, could avoid the use of  $SW1$  and  $SW2$  in the above example. The problems of writing goto-less programs for many

of the irreducible n-predicate flowcharts go away if this construct is used. But consider flowchart 3.7 from Appendix A (with blocks on three edges and C nodes replaced by ):



LEAVE will not help in this case. Nor is it obvious how one would use auxiliary variables as we did in the above example to avoid goto's. Before going on, if the reader is not familiar with Ashcroft and Manna's method (1) he might try to write a goto-less program representing the above flowchart in ALGOL, PL/I, or pure LISP.

Of the eight irreducible 3-predicate flowcharts in Appendix A, 3.7 is the only one with three loops. In the example flowchart (15) these are:  $(P_1 P_2 K)$ ,  $(P_2 P_3 L)$ , and  $(P_1 J P_3 L P_2 K)$ . This flowchart will be used to illustrate Ashcroft and Manna's ALGORITHM 2: cut-set points are chosen so that each loop in the flowchart contains at least one cut-set point. One cut-set point is also placed on the output edge. The program is represented by a DOWHILE containing a CASE (or cascaded IFTHENELSE's) with a block for each path from one cut-set point to another. Auxiliary variables are used to determine which path is being followed, or more specifically, which cut-set point is being traversed. For example, if we use  $w$ ,  $x$ ,  $y$ , and  $z$  as the cut-set

points of flowchart (15), we could write in PL/I:

THREE\_LOOPS:

PROC;

W=1; X=0; Y=0; Z=1

DO WHILE Z;

IF W THEN DO; W=0;

IF P1 THEN X=1;

ELSE DO; J; Y=1; END;

END;

ELSE IF X THEN DO; X=0;

IF P2 THEN Y=1;

ELSE DO; K; W=1; END;

END;

ELSE IF Y THEN DO; Y=0;

IF P3 THEN Z=0;

ELSE DO; L; X=1; END;

END;

END THREE\_LOOPS;

The DOWHILE block is repeated until Z is set to 0 ( $P_3$  is true). Which of the three THEN DO; . . . . END; blocks is performed depends on which cut-set point was traversed on the last iteration. Each of these three blocks is the set of statements on the path from one cut-set point to the next. An auxiliary boolean variable representing a cut-set point is set to false when the control token departs the cut-set point and is set to true when the control token arrives at the cut-set point.\*

In the above example enough cut-set points were chosen so there are no repetitions of program blocks. Since Ashcroft and Manna's statement of ALGORITHM 2 requires only one cut-set point in each loop and one on the output edge, we should need cut-set points only at x and z.

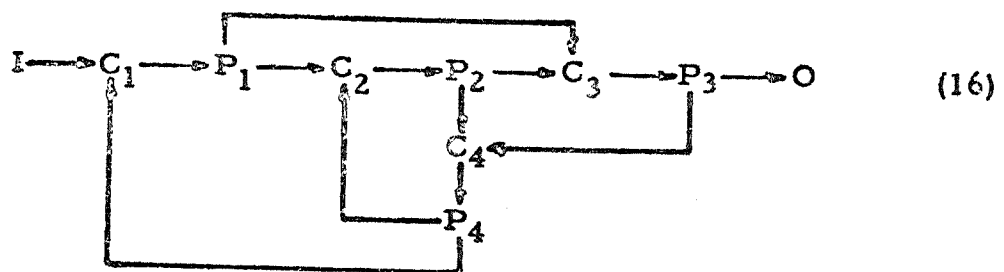
---

\* The auxiliary boolean variable Y is redundant since  $Y=1$  is equivalent to ( $W=0$  and  $X=0$ ), and  $Y=0$  is equivalent to ( $W=1$  or  $X=1$ ). Y is included here for clarity.

But  $w$  must be included because the control token must have one cut-set point to move from initially. (The statement of Ashcroft and Manna's ALGORITHM 2 should be modified to include this requirement.) If only cut-set points  $w$ ,  $x$ , and  $z$  are used, the resulting program would eliminate the IF Y THEN DO; .....END; block but this would require the substitution of an IF P3 THEN.....ELSE.....; statement for the two appearances of the  $Y=1$  assignment statement in the above PL/I program. Thus there is a tradeoff between (a) duplicate statements (or multiple procedure calls) in a program and (b) the number of auxiliary variables added to avoid the goto.

One final point about flowchart 3.7: it has a symmetry beyond that of the particular way it has been drawn: each of the three loops in 3.7 has as many entry points as it has exits (put another way, each loop has as many C nodes as it has P nodes).

$n=4$ . There are 82 irreducible 4-predicate flowcharts; these are drawn in Appendix A. The maximum number of loops in any of these flowcharts is five--and there are five such flowcharts: 4.22, 4.25, 4.51, 4.52 and 4.60. Only one of these, 4.25, has the symmetry property mentioned above: each of its five loops has as many entries as it has exits. Flowchart 4.25 is redrawn here:



The five loops are  $(P_2 C_4 P_4 C_2)$ ,  $(C_1 P_1 C_2 P_2 C_4 P_4)$ ,  
 $(C_2 P_2 C_3 P_3 C_4 P_4)$ ,  $(C_1 P_1 C_3 P_3 C_4 P_4)$ , and  
 $(C_1 P_1 C_2 P_2 C_3 P_3 C_4 P_4)$ .

Another interesting subset is the set of flowcharts that contain some loop that does not touch the main path. These are flowcharts 4.1 and 4.2. Each of them contains three loops. A cut-set point off of the main path breaks each of them.

### Utility of Results.

The enumeration of all irreducible flowcharts with up to four predicate nodes is a practical limit. This set should provide enough examples (or counterexamples) to fuel the goto-less programming "controversy" and to aid in the analysis of the general case.

For example, the suggestion to include a SEARCH primitive in a language to eliminate the 2-predicate flowchart problem naturally raises the question of why this would be more useful than primitives for some of the 3-predicate cases. Because of the number of irreducible 3-predicate flowcharts, someone looking for interesting counter examples might have a stronger point in using one of these-- they are all quite innocent-looking. On the other hand, one might be hard-pressed to find a problem which requires the control paths of flowcharts 3.7 or 4.25, for example.

The abstract flowchart schema used here should be investigated further; there might be more useful properties which could be extracted from the set of irreducible  $n$ -predicate flowcharts. The flowcharts enumerated here might provide some direction to the search for these properties. Note for example that none of these flowcharts contain trees of predicate nodes. Since looping is the main difficulty in goto-less programming, we should look for additional properties of loops in these flowcharts.



Perhaps there is something more subtle but just as simple as no goto's that we should seek in developing control structures for flowchart programs. If this is so, we must look for properties of the entire flowchart, not just individual loops and paths. These might be topological properties or transformation properties more powerful than node splitting.


A more efficient generative grammar should also be sought. For example, it can be shown that the procedure used here for generating irreducible  $n$ -predicate flowcharts avoids the production of any flowchart with 1-predicate flowcharts or  $(n-2)$ -predicate flowcharts embedded in it. Can the procedure be further refined to avoid generation of duplicate irreducible flowcharts? The relationship between Ashcroft and Manna's cut-set points for an irreducible  $(n-1)$ -predicate flowchart and the cut-points used here for inserting a PC segment should be investigated.

## REFERENCES

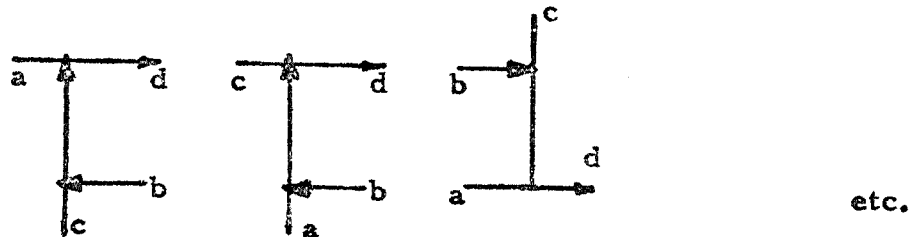
1. Ashcroft, Edward and Manna, Zohar. "The translation of 'goto' programs to 'while' programs". Proc. IFIP Congress 71, Ljubljana, Aug. 1971.
2. Bohm, Corrado and Jacopini, Giuseppe. "Flow diagrams, Turing machines and languages with only two formation rules". CACM 9 (May 1966).
3. Cooper, D. C. "Bohm and Jacopini's reduction of flow charts". Letter to the Editor, CACM 10 (Aug. 1967).
4. Dijkstra, E. W. "Go to statement considered harmful". Letter to the Editor, CACM 11 (March 1968).
5. Fisher, David A. "A Survey of Control Structure in Programming Languages", SIGPLAN Notices, 7, 11, (November 1972).
6. Hopkins, Martin, "A case for the goto", Proceedings ACM '72, Boston, August 1972.
7. Kessler, M. M. Internal IBM Federal Systems Division Report on structured programming, 18 December 1970.
8. Knuth, D. E. and Floyd, R. W. "Notes on avoiding 'goto' statements" Information Processing Letters 1, North-Holland, Amsterdam (1971), 23-31.
9. Leavenworth, B. M. "Programming With(out) the GOTO", Proceedings ACM '72, Boston, August 1972.
10. McIlroy, M. D. Review of Leavenworth, B. M. "Programming Without the GOTO". Review no. 24, 392, Computing Reviews, 14, 1 (January 1973).
11. Mills H. "Top down programming in large systems", Debugging Techniques in Large Systems (Ed. Rustin, Randall), Prentice-Hall, Englewood Cliffs, N. J. 1967.
12. Morris, James B. Jr. and Wells, Mark B. "The Specification of Program Flow in MADCAP 6", SIGPLAN Notices, 7, 11, (November, 1972).
13. Wulf, W. A. et al. "BLISS: A Language for systems Programming," CACM 11 (December 1971).
14. Wulf, W. A. "A case against the goto". Proceedings ACM '72, Boston, August 1972.

## Appendix A

### Irreducible 3- and 4-Predicate Flowcharts

Each flowchart below has the main path drawn as a horizontal line. The main path is the simple (nonlooping) path from input (I) to output (O) which traverses the most predicates (P's). \* Each flowchart is drawn so that there is no sequence of two or more adjacent collector nodes ( does not appear) on the main path. Where a flowchart has such a sequence of collector nodes, or more generally a tree of collector nodes, permutations of the edges of such trees are not distinguished; only one (arbitrary) form of these trees is shown.

For example



are isomorphic in this flowchart schema and only one form is shown.

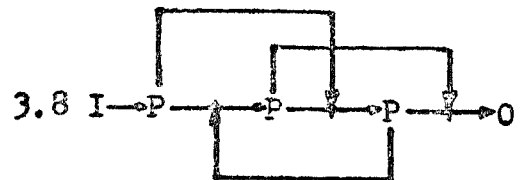
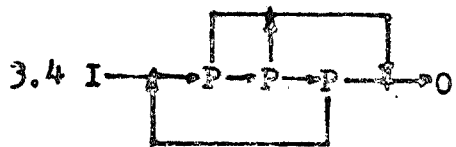
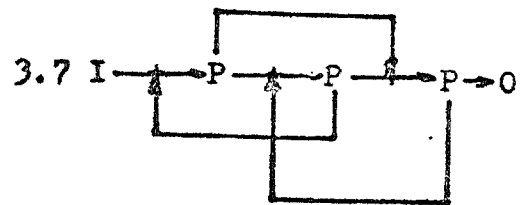
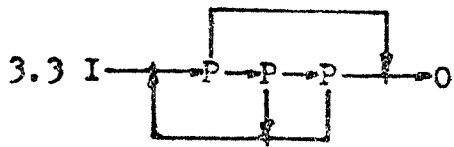
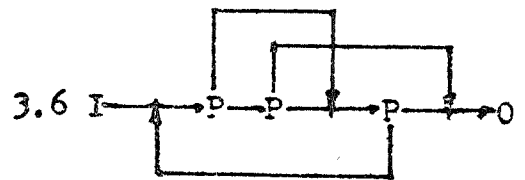
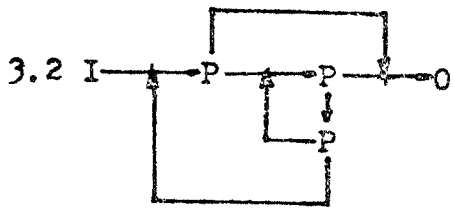
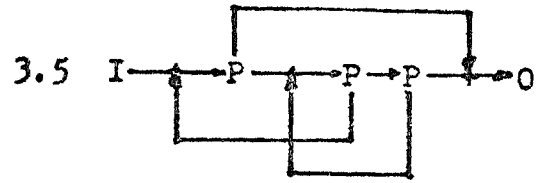
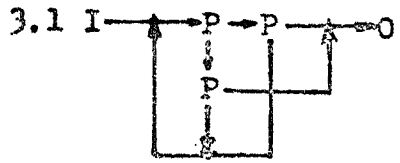
All predicate nodes not on the main path are drawn below it. Wherever possible, within the constraints of the above conventions, edges not on the main path are arranged so that back-pointing (looping) edges are below the main path, and forward-pointing ("skip" or "escape") edges are above the main path.

\* In a few cases the main path is not unique. An example is 4.66, which could have been drawn as:



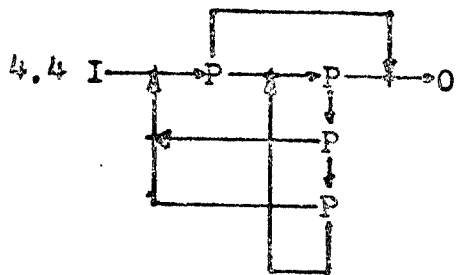
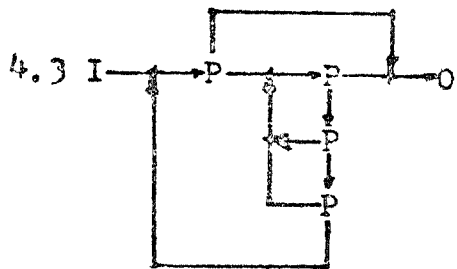
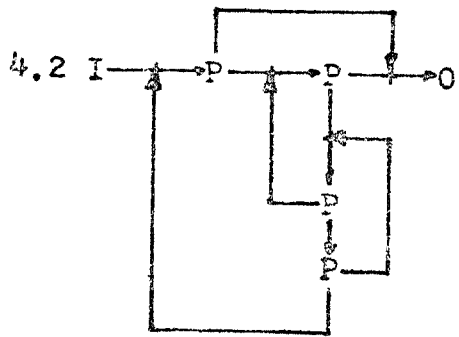
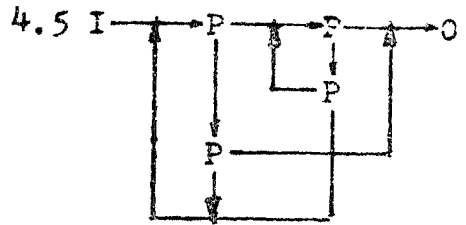
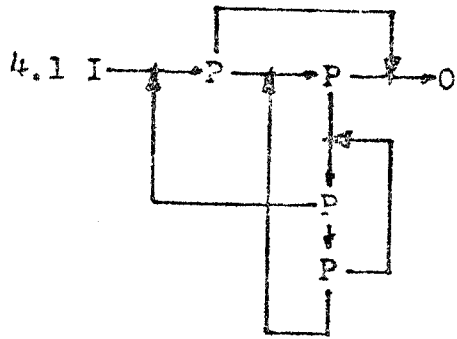
The eight irreducible 3-predicate flowcharts are shown on the next page. This is followed by an index to the irreducible 4-predicate flowcharts. The major ordering of this set of flowcharts is by number of predicate nodes on the main path (2, 3, or 4). Minor ordering is by the position of intervening collector nodes on the main path. The subset of flowcharts with the same sequence of nodes on the main path are shown in arbitrary order on one page.

### Nonreducible 3-Predicate Flowcharts

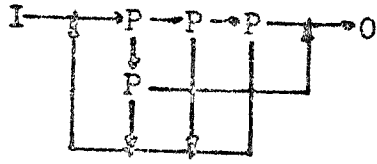


Index to the Set  
of 4-Predicate Flowcharts

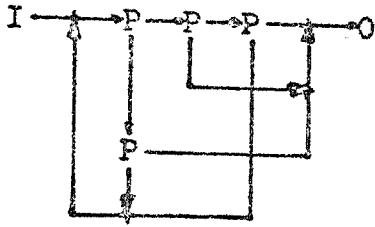
Main Path (excluding I and O nodes and collector nodes adjacent to I or O nodes)	Page
P —  — P	A-5
P — P — P	A-6
P —  — P — P	A-7
P — P —  — P	A-8
P —  — P —  — P	A-9
P — P — P — P	A-10
P —  — P — P — P	A-11
P — P —  — P — P	A-12
P — P — P —  — P	A-13
P —  — P —  — P — P	A-14
P —  — P — P —  — P	A-15
P — P —  — P —  — P	A-16
P —  — P —  — P —  — P	A-17



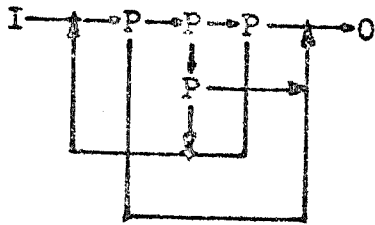
4.6



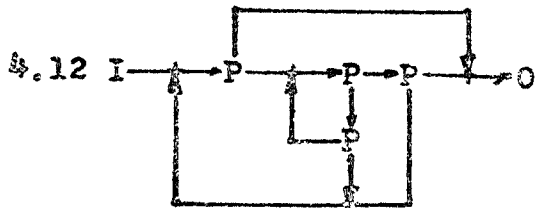
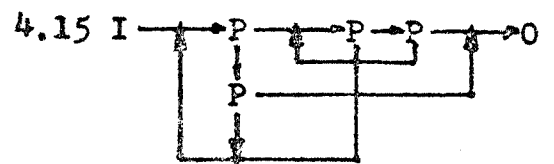
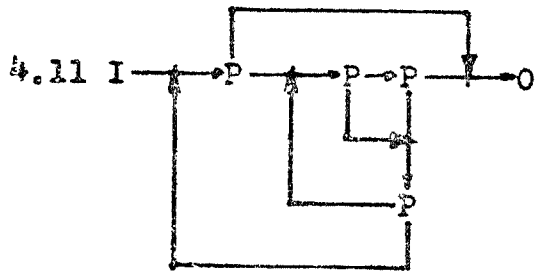
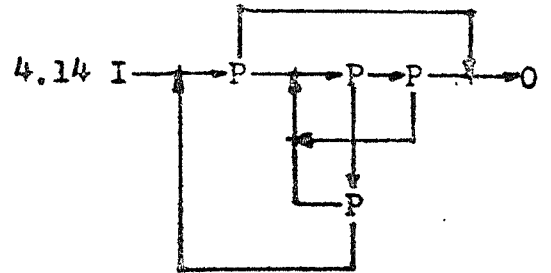
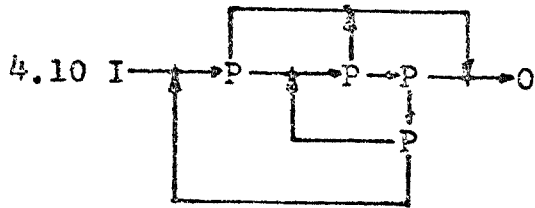
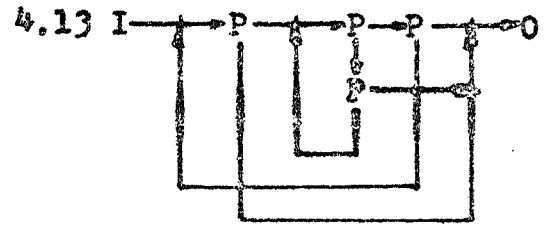
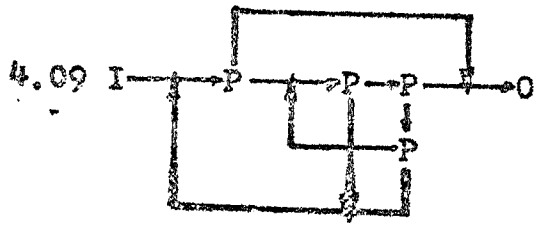
4.7



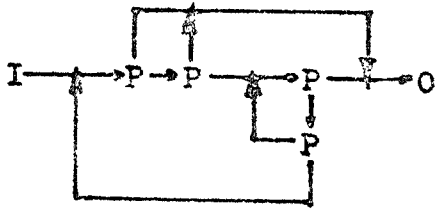
4.8



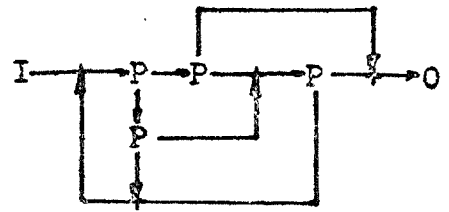




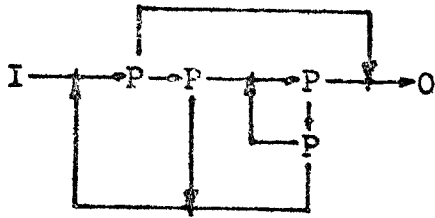
4.16



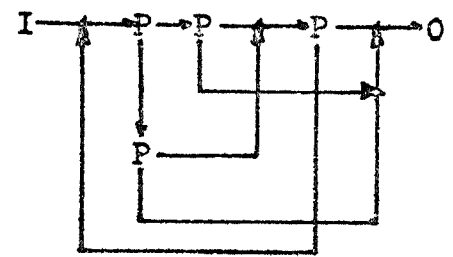
4.20



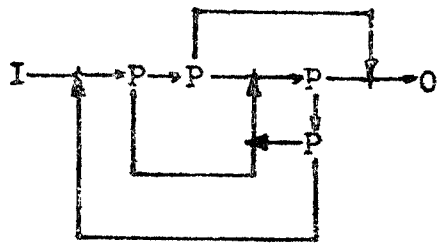
4.17



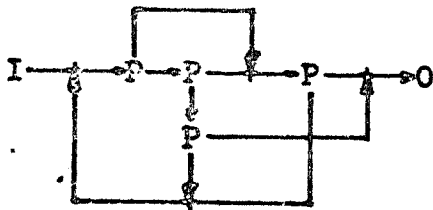
4.21

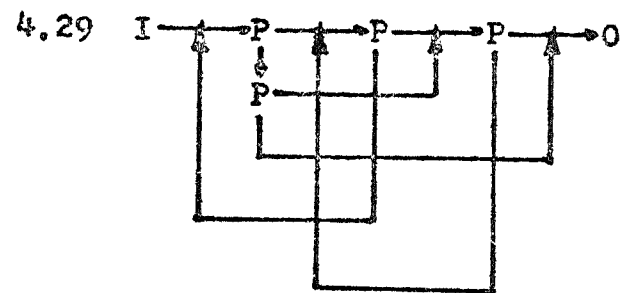
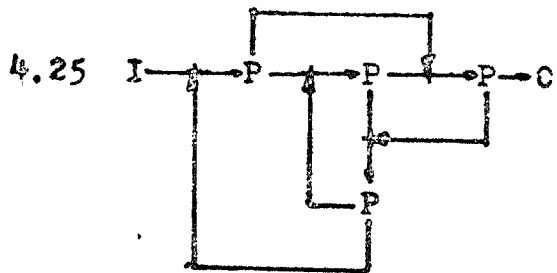
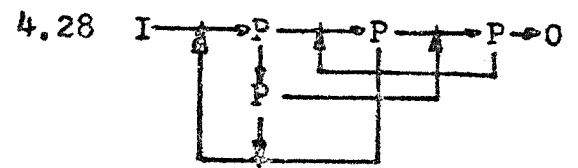
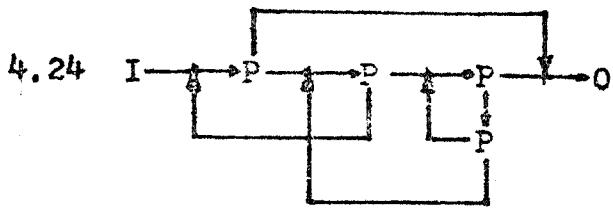
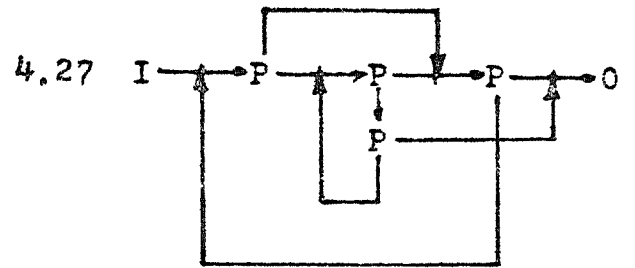
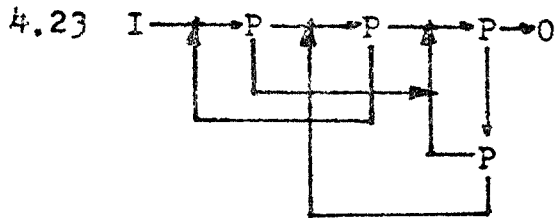
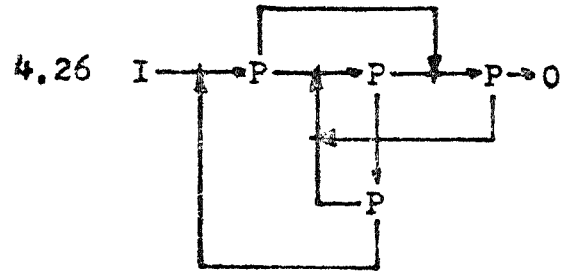
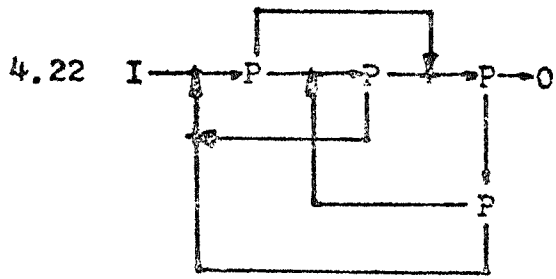


4.18

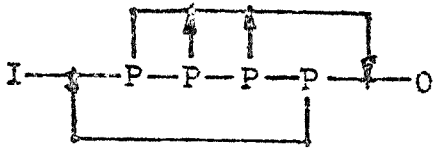


4.19

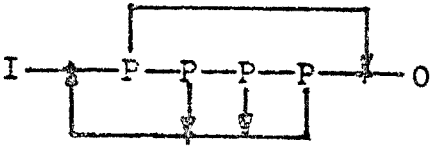




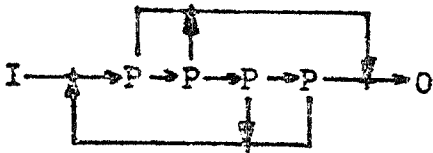
4.30



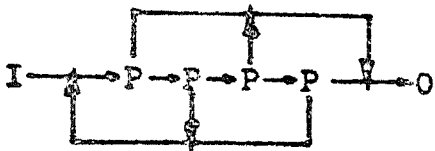
4.31

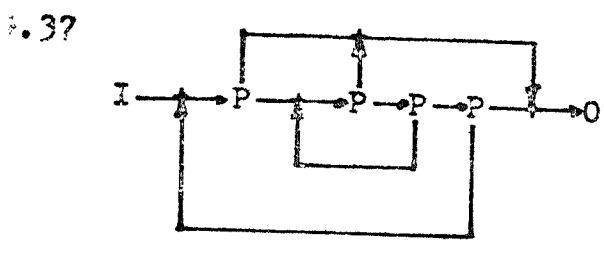
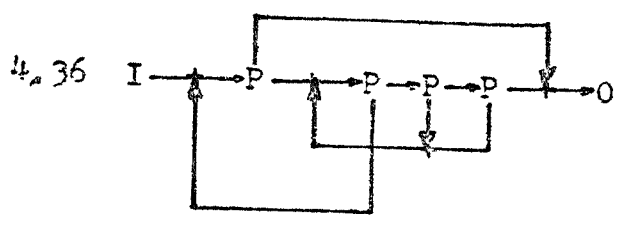
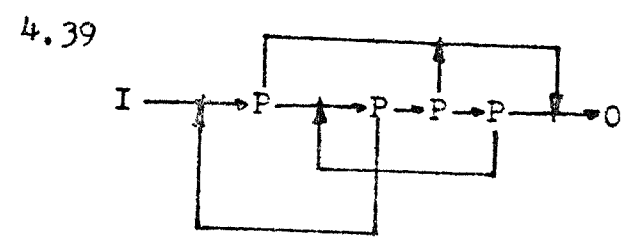
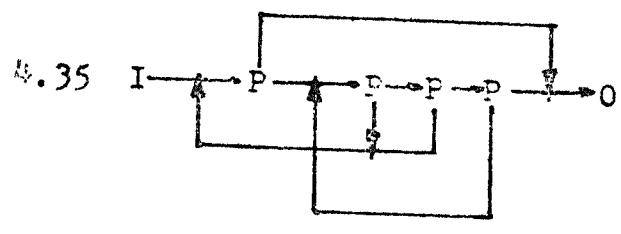
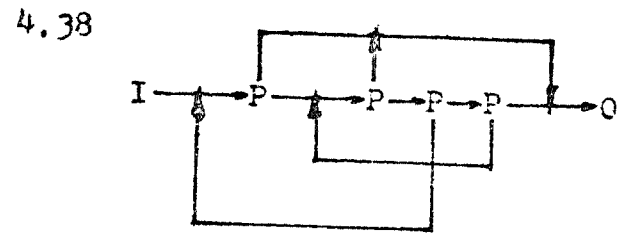
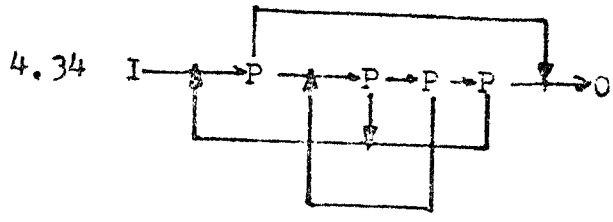


4.32

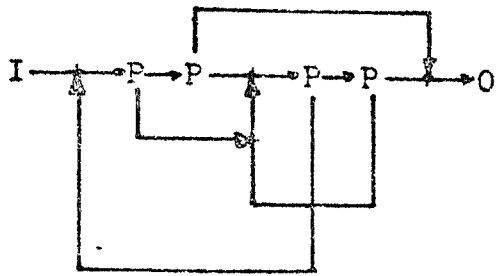


4.33

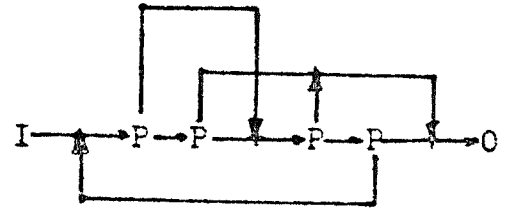




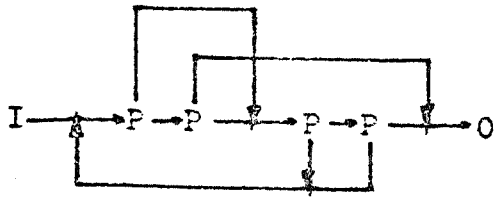
4.40



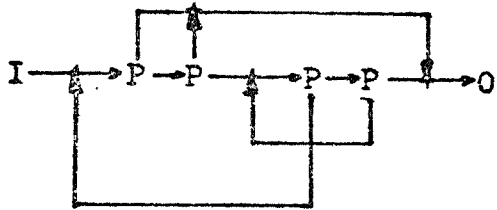
4.44



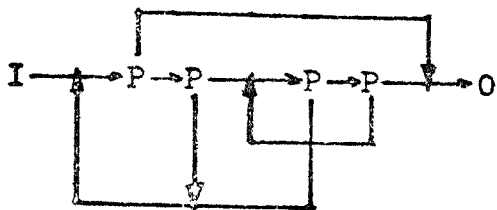
4.41

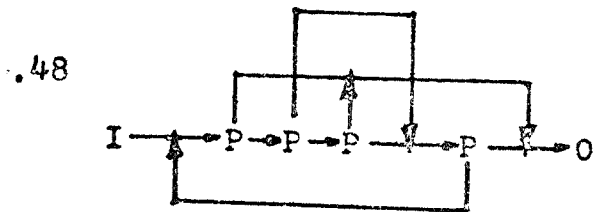
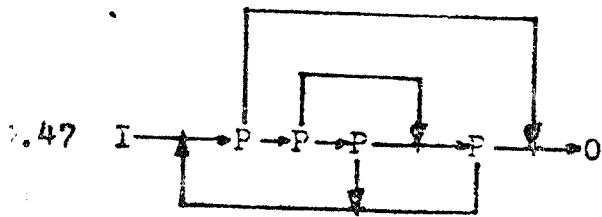
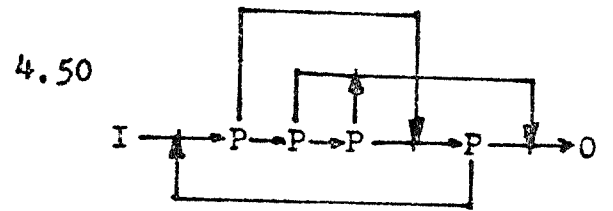
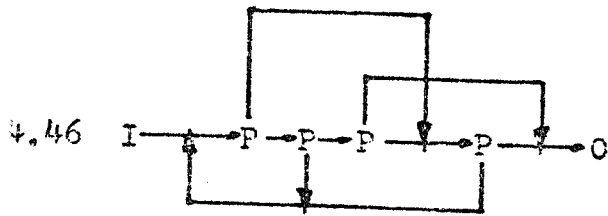
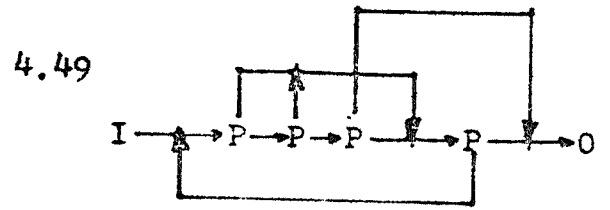
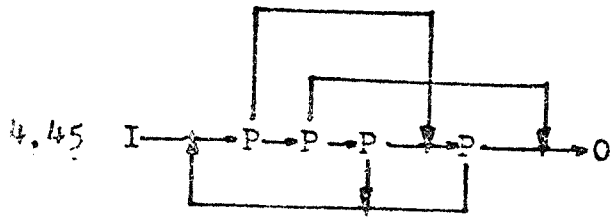


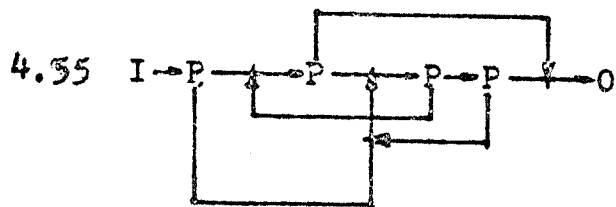
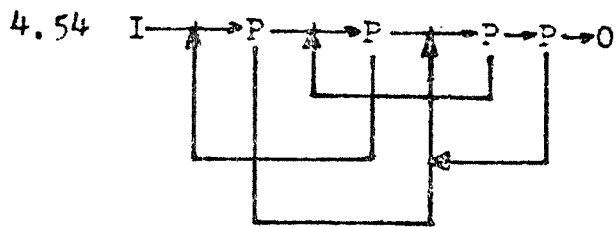
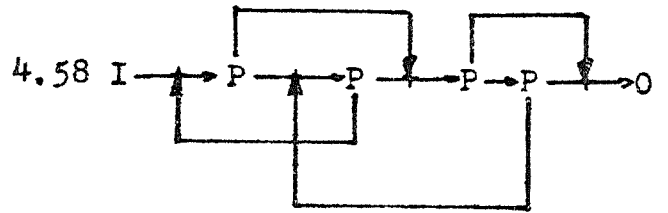
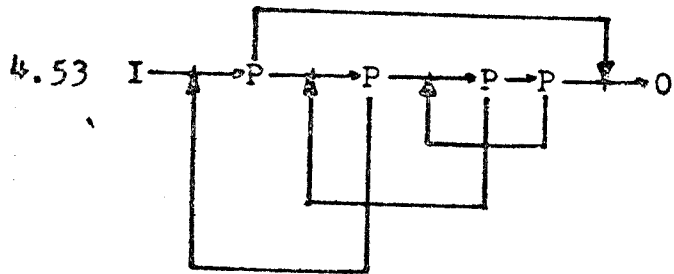
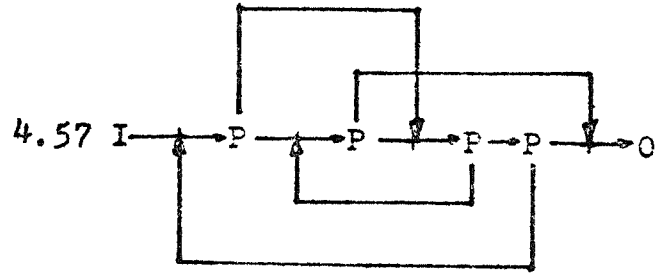
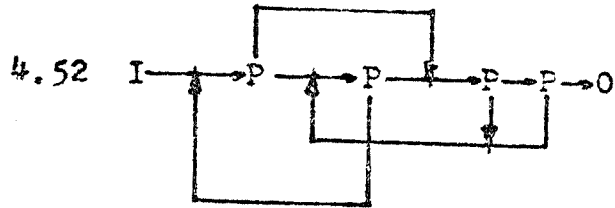
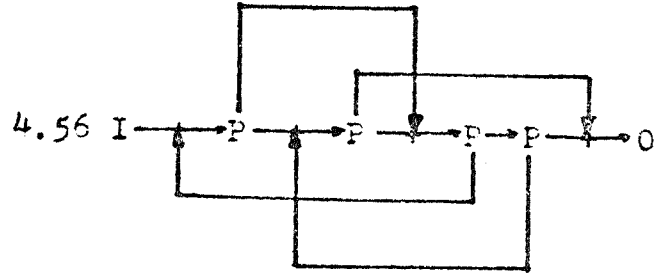
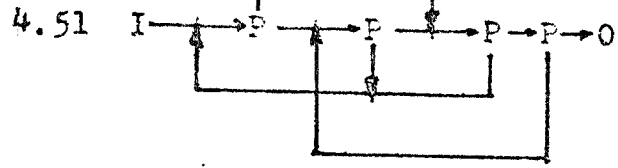
4.42



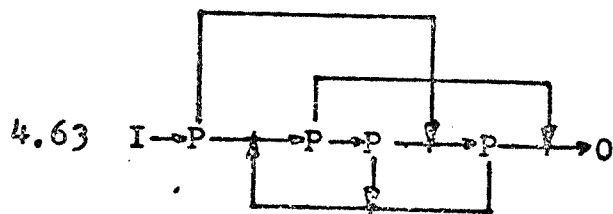
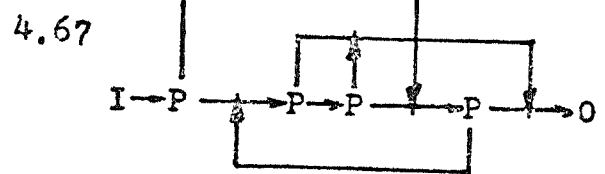
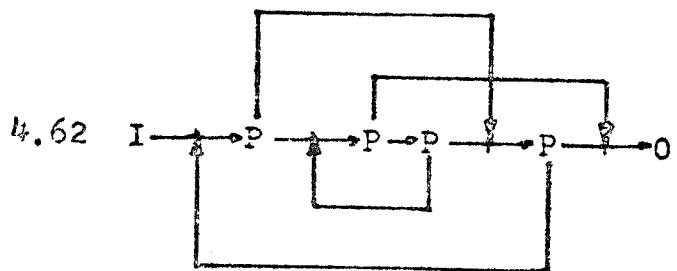
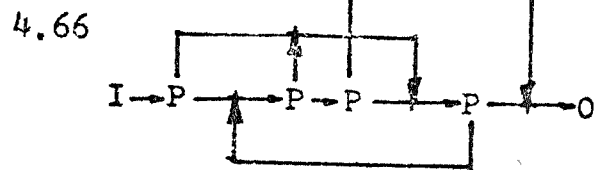
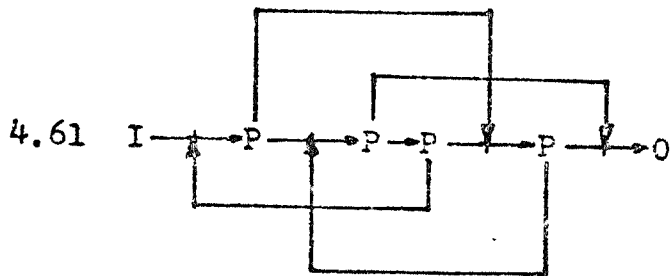
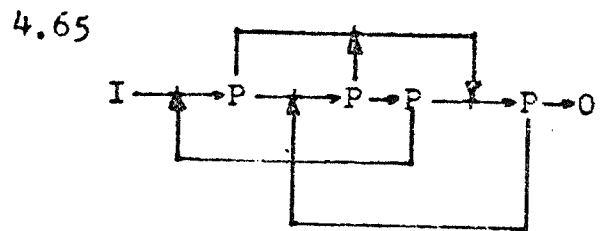
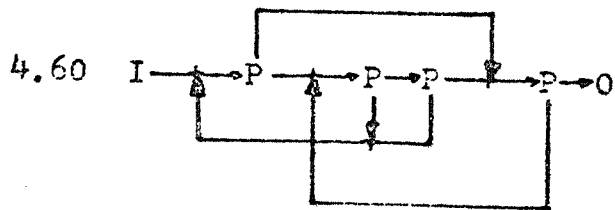
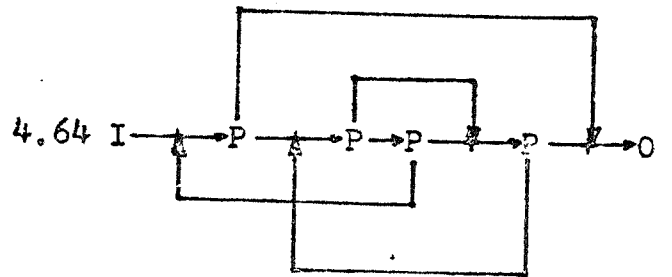
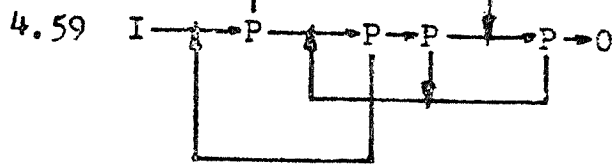
4.43

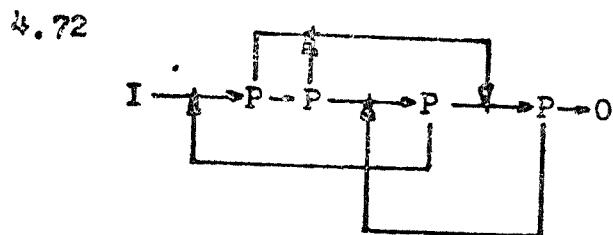
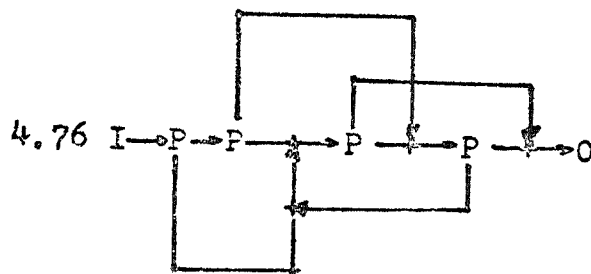
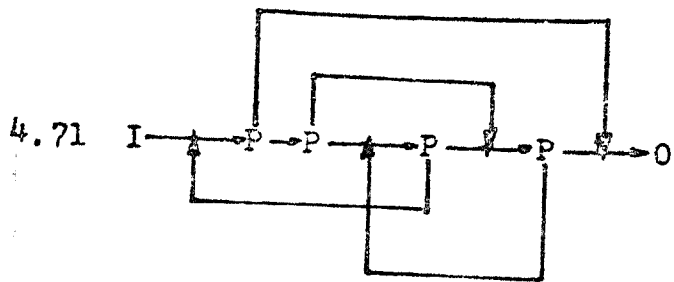
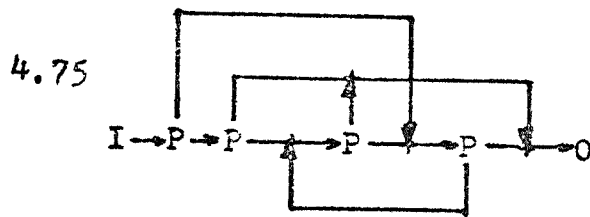
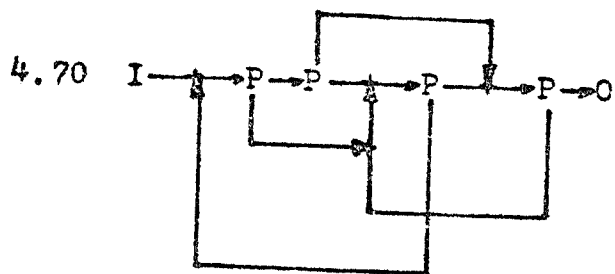
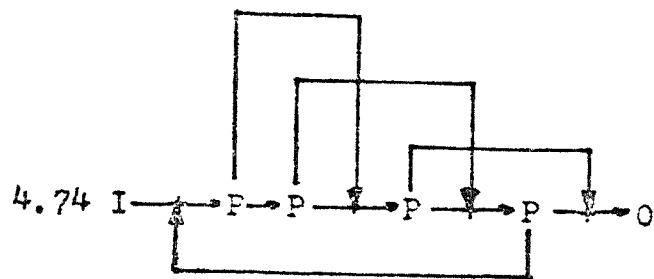
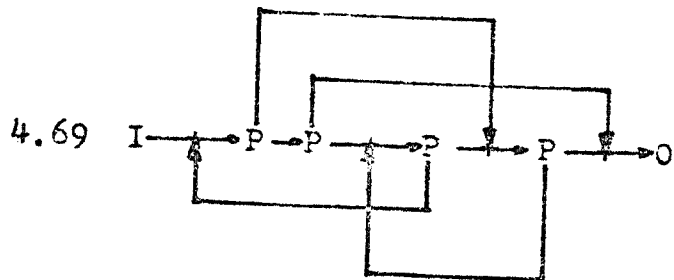
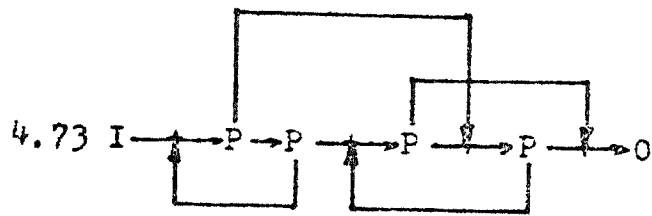
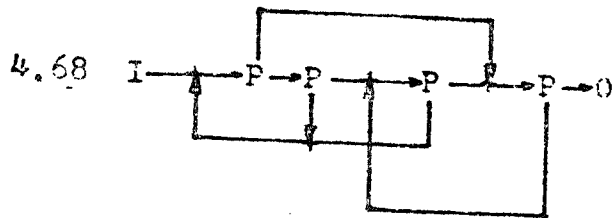


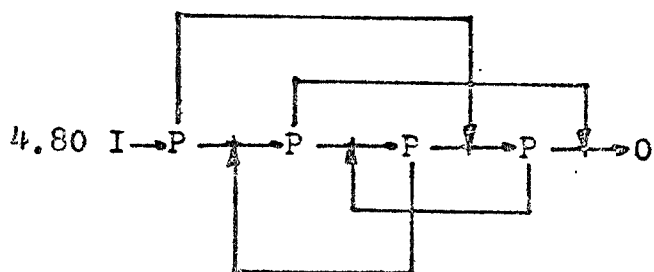
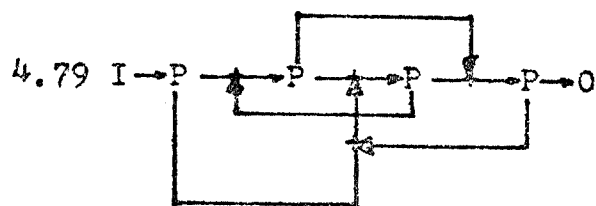
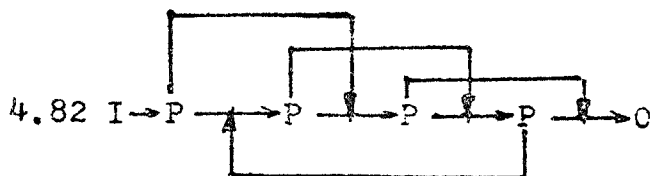
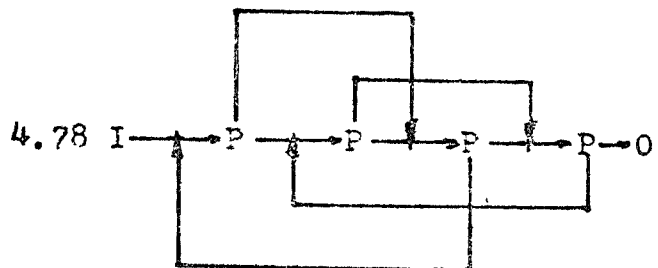
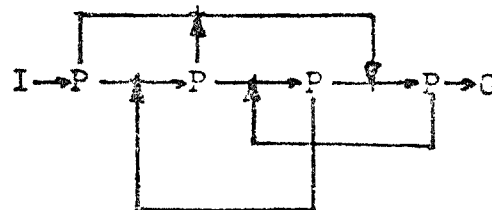
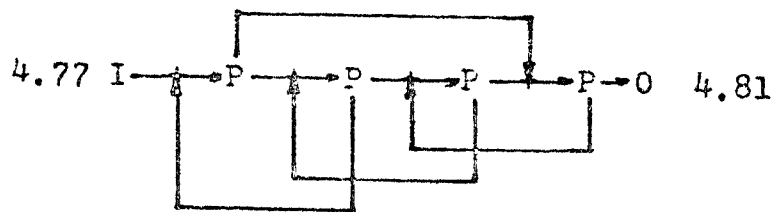










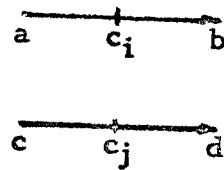


## Appendix B

### Description and Proof of the Procedure for Generating All Irreducible $n$ -Predicate Flowcharts

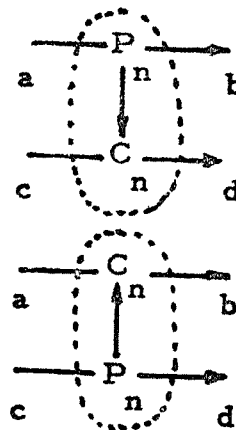
The procedure for generating all irreducible  $n$ -predicate flowcharts is as follows:

- a. For a given  $(n-1)$ -predicate flowchart, identify all ordered pairs of cut-points  $(c_i, c_j)$  on the flowchart. The pair of cut-points can be on the same edge of the flowchart or on different edges.
- b. For each pair of cut-points generate two  $n$ -predicate flowcharts from the  $(n-1)$ -predicate flowchart by cutting each edge of the pair:



and inserting a flowchart segment:\*

or:



\* The inserted flowchart segment, within the dashed lines, is called a PC segment hereafter.

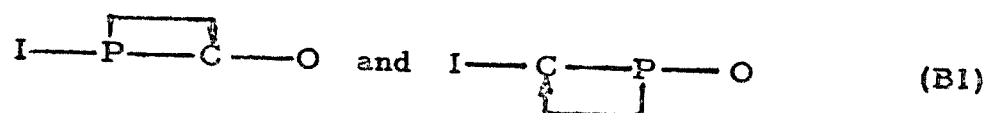
- c. Collapse all trees of collector nodes with  $\lambda$  leaves in the flowcharts to a single collector node of in-degree  $\lambda$  and out-degree one.
- d. Perform steps a. -d. on all irreducible  $(n-1)$ -predicate flowcharts.
- e. Eliminate from the set of generated flowcharts any copies of flowcharts which are isomorphic. Isomorphic here means: if under some relabelling of nodes, the list of edges of two flowcharts are the same, then the flowcharts are isomorphic.
- f. Eliminate any flowchart which has a flowchart embedded in it.

In the proof of this procedure, steps e. and f. will be ignored.

Our main concern here is whether the set of  $n$ -predicate flowcharts generated in steps a. -d. contains at least one copy of all the irreducible  $n$ -predicate flowcharts. We will first prove the lemma:

Lemma 1: For  $n \geq 1$ , at least one copy of all possible  $n$ -predicate flowcharts can be generated by steps a. -c. of the above procedure when applied to the set of all  $(n-1)$ -predicate flowcharts.

Proof: by induction on  $n$ . For  $n=1$ , the set of  $(n-1)$ -predicate is the single edge:  $(I-O)$ . Step b. then produces:

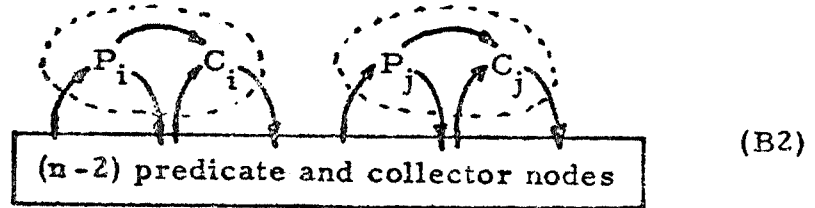


Clearly these are the only 1-predicate flowcharts.

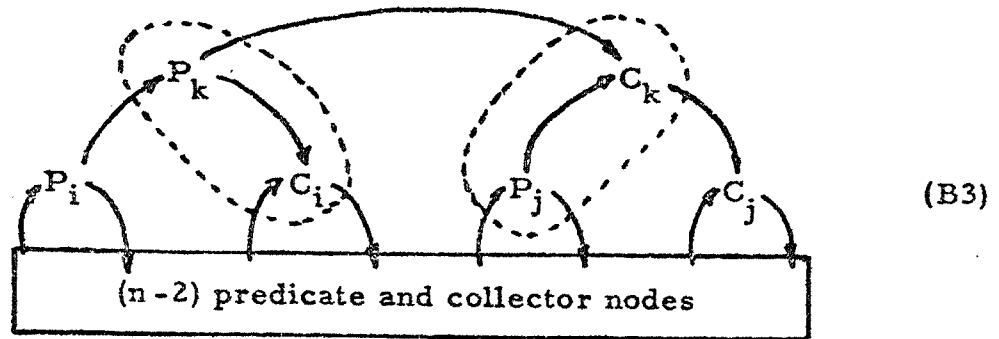
Inductive Step: suppose there is some  $(n+1)$ -predicate flowchart that cannot be generated from the set of  $n$ -predicate flowcharts, that it must be generated by inserting PC segments in some  $(n-i)$ -predicate flowchart,  $1 \leq i \leq (n-1)$ . This implies that the sequence of adding PC segments is significant. Thus a necessary and sufficient condition for the contradiction is that there is some  $(n+1)$ -predicate flowchart which can only be generated by inserting  $PC_1$  and  $PC_2$  in that order in some  $(n-1)$ -predicate flowchart. For such an ordering to be significant,  $PC_2$  must be inserted so that it "breaks"  $PC_1$  in the  $(n+1)$ -predicate flowchart. "Break" means that when  $PC_2$  is inserted, one of the cut-points chosen in step b, must be on the new  $(P \rightarrow C)$  edge added when  $PC_1$  was inserted in the  $(n-1)$ -predicate flowchart. Therefore when  $PC_2$  is added,  $PC_1$  no longer exists. (If neither of the edges chosen satisfies this condition, then  $PC_1$  and  $PC_2$  could be added to the  $(n-1)$ -predicate flowchart in either sequence). It follows that the contradiction implies that there is only one PC segment in such an  $(n+1)$ -predicate flowchart.

When  $PC_1$  is added to an  $(n-1)$ -predicate flowchart, an  $n$ -predicate flowchart ( $X_n$ ) is generated. By the inductive assumption and the above argument,  $X_n$  must have at least two PC segments, including  $PC_1$ . When  $PC_2$  is inserted in  $X_n$ , two cases arise:

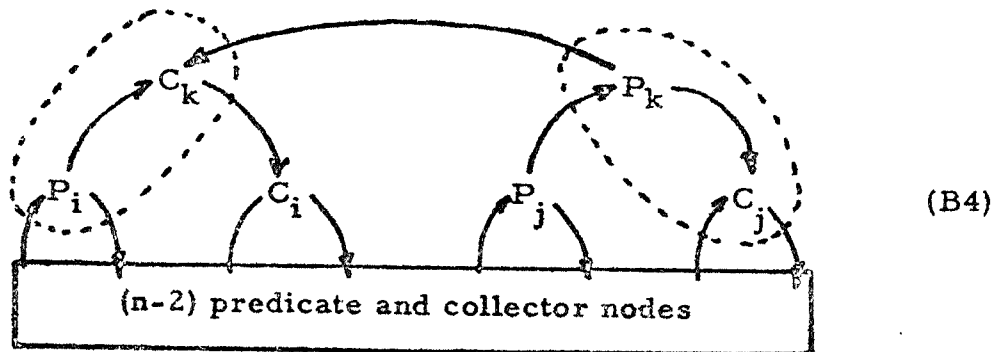
(a)  $PC_2$  breaks only one of the PC segments in  $X_n$ . Then the  $(n+1)$ -predicate flowchart so generated will have at least two PC segments -- the one left unbroken and  $PC_2$ . (b)  $PC_2$  breaks both distinct PC segments in  $X_n$ . If we show two of the PC segments in  $X_n$  explicitly, it looks like:



The  $(n+1)$ -predicate flowchart looks like:



or like:

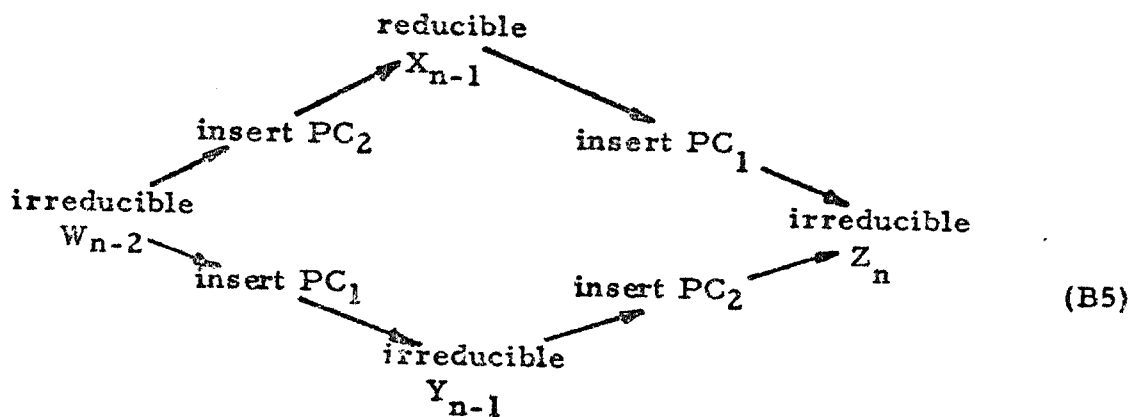


There are still two PC segments in (B3) and two PC segments in (B4): those enclosed by dashed lines. Thus in breaking two PC segments,  $PC_2$  creates two more. Therefore if any  $n$ -predicate flowchart has at least two PC segments, any  $(n+1)$ -predicate flowchart generated by steps a. and b. must also have at least two PC segments, so the order in which

these two PC segments are inserted is not significant. The contradiction is false, and Lemma 1 is proved. Also from observation of (B2), (B3), and (B4), the following corollary is true: for  $n \geq 3$ , every  $n$ -predicate flowchart has at least two PC segments.

A second lemma will now be proved: Lemma 2: For  $n \geq 2$ , an irreducible  $n$ -predicate flowchart  $Z_n$  can be generated by successive insertions of distinct  $PC_1$  and  $PC_2$  segments\* in an irreducible  $(n-2)$ -predicate flowchart  $W_{n-2}$ ; an intermediate  $n$ -predicate flowchart  $X_{n-1}$  is reducible, and reversing the order of insertion will generate an intermediate irreducible  $(n-1)$ -predicate flowchart  $Y_{n-1}$ .

Schematically:

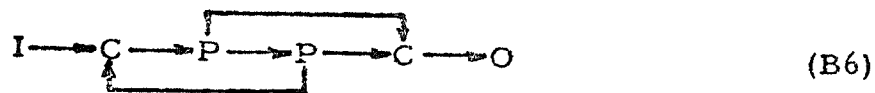


\* Distinct means that  $PC_1$  does not break  $PC_2$  or vice versa.

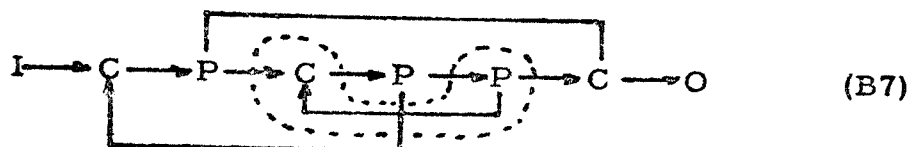


For example, an irreducible 4-predicate flowchart can be generated from the only irreducible 2-predicate flowchart as follows.

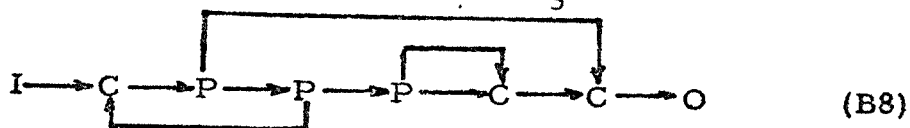
$W_2$  is:



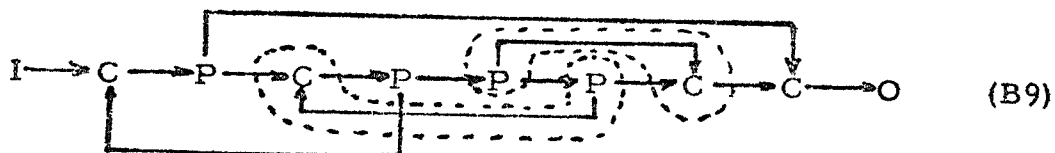
A  $Y_3$  can be generated from (B6) by inserting a  $PC_1$  (enclosed in dashed lines):



This is the irreducible flowchart 3.5 in Appendix A. A  $PC_2$  (in dashed lines) can be inserted in (B6) to generate an  $X_3$ :



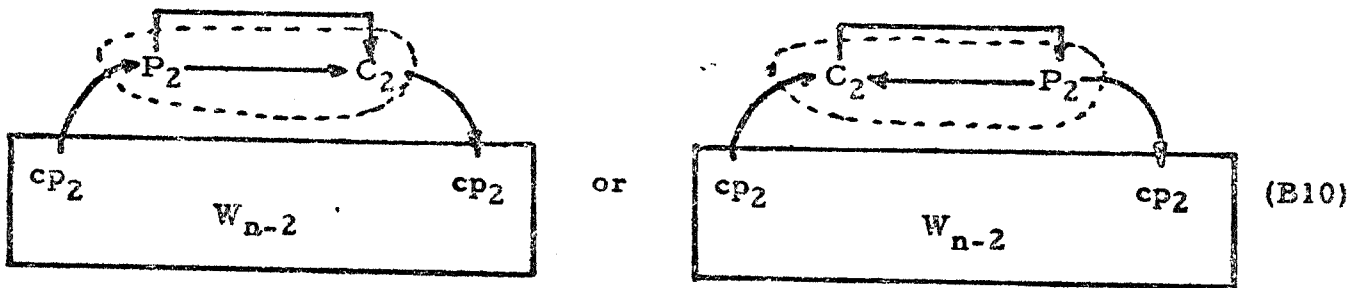
This flowchart is reducible (it has a 1-predicate flowchart embedded in it). Inserting both  $PC_1$  and  $PC_2$  in (B6) yields an irreducible flowchart  $Z_4$ , which is equivalent to flowchart 4.39 in Appendix A:



Proof: First we note that if an irreducible  $(n-1)$ -predicate flowchart  $Y_{n-1}$  is to remain irreducible after the insertion of a  $PC_2$  segment, the cut-points selected must be on different edges. This is also a sufficient condition for retaining an irreducible flowchart if,

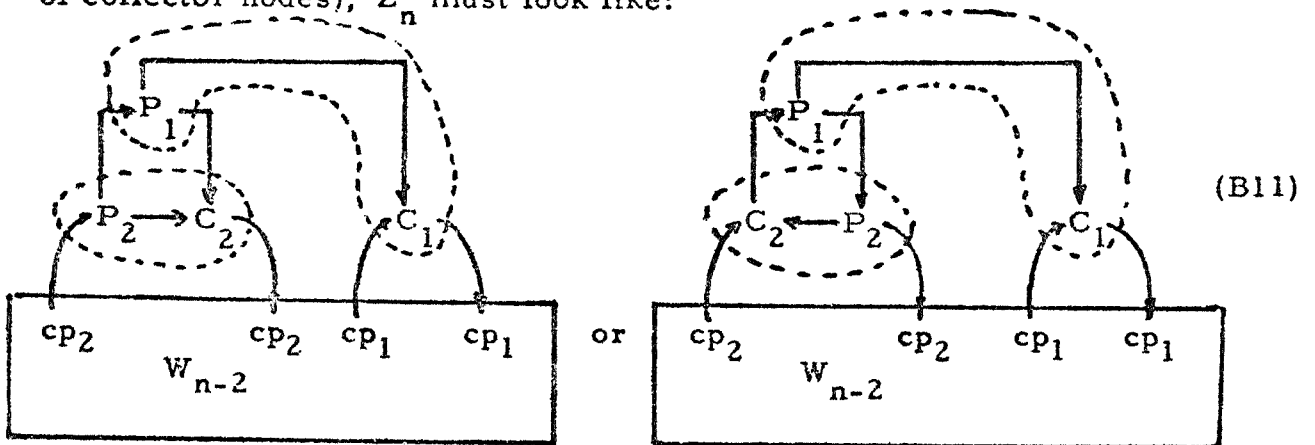
as assumed here, step C, of the procedure (collapsing trees of collector nodes) is applied to  $Y_{n-1}$  before  $PC_2$  is inserted. Second, if an  $(n-1)$ -predicate flowchart has an embedded flowchart, then in order to generate an irreducible flowchart from it, only one of the cut-points used to insert a PC segment can be inside the embedded flowchart: if both cut-points were in the embedded flowchart, it would remain an embedded flowchart.

Now consider the properties of  $X_{n-1}$ ,  $Y_{n-1}$ , and the two inserted PC segments required to generate  $Z_n$  from  $W_{n-2}$ . We can generate a (reducible)  $X_{n-1}$  from an (irreducible)  $W_{n-2}$ , if the cut-points ( $cp_2$ ) for inserting  $PC_2$  are on the same edge. The only other way to generate  $X_{n-1}$  would be to put one cut-point on the input edge and other on the output edge of  $W_{n-2}$ , but this would violate a condition for  $PC_1$  (see below). So  $X_{n-1}$  will have a 1-predicate flowchart embedded in it:

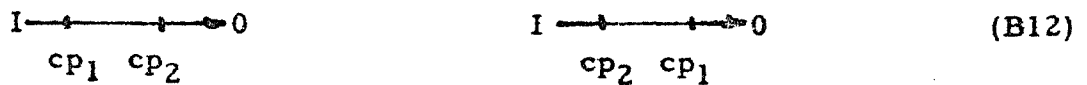


If  $Z_n$  is irreducible, this embedded flowchart must be broken by  $PC_1$  but it may not break the  $PC_2$  segment and vice versa. Finally, if we

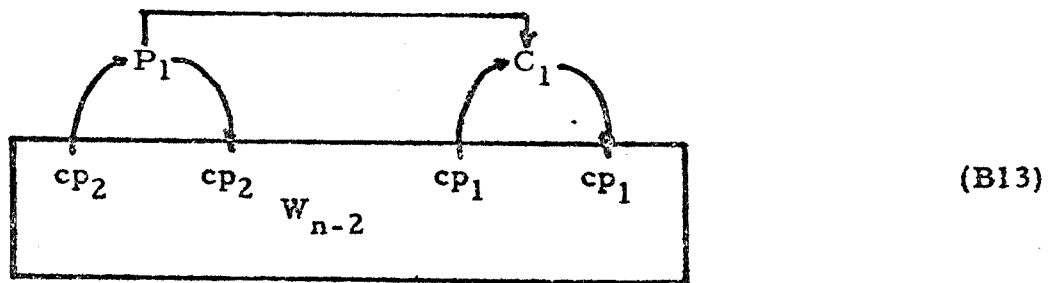
take into account step C. of the generating procedure (collapsing trees of collector nodes),  $Z_n$  must look like:



where cut-points  $cp_1$  and  $cp_2$  are on different edges. This construction holds for  $n=2$ : the one irreducible 2-predicate flowchart will be generated if the cut-points on the single edge ( $I \rightarrow 0$ ) flowchart are respectively:



Returning to the flowcharts of (B11), since  $PC_1$  and  $PC_2$  are distinct,  $Y_{n-1}$  can be generated from  $W_{n-2}$  by adding  $PC_1$  at the same cut-points.



$Y_{n-1}$  must be irreducible because  $CP_1$  and  $CP_2$  are on different edges of (irreducible)  $W_{n-2}$ . Thus the hypothesized construction is possible and when it is applied, the lemma holds. The lemma is proved. Also since

$PC_1$  and  $PC_2$  are distinct, the process of generating  $Z_n$  is reversible, so we have the corollary: for  $n \geq 2$  we can obtain an irreducible  $(n-2)$ -predicate flowchart by removing two distinct PC segments from some irreducible  $n$ -predicate flowchart; one of the two possible intermediate  $(n-1)$ -predicate flowcharts is reducible and the other is irreducible.

Proof of the Procedure for generating the set of irreducible  $n$ -predicate flowcharts: as noted previously, the main concern is whether steps a. -c. of the procedure when applied to the subset of  $(n-1)$ -predicate flowcharts which are irreducible  $\{f'_{n-1}\}$  is sufficient to generate the set of irreducible  $n$ -predicate flowcharts  $\{f'_n\}$ ,  $n \geq 1$ .

Proof: by induction of  $n$ . for  $n=1$ , the procedure generates the flowcharts in (B1), the set of 1-predicate flowcharts, which by definition are irreducible. Inductive step: suppose there is some irreducible  $(n+1)$ -predicate flowchart  $Z_{n+1}$  which cannot be generated from  $\{f'_n\}$ . Lemma 1 states that  $Z_{n+1}$  can be generated from some  $n$ -predicate flowchart, and for the contradiction to hold, such a flowchart,  $X_n$ , must be reducible. By Lemma 2, (a)  $Z_{n+1}$  can be generated by the insertion of two distinct PC segments in some irreducible  $(n-1)$ -predicate flowchart and (b) one of the two intermediate  $n$ -predicate flowcharts,  $Y_n$ , is irreducible. By the inductive assumption  $Y_n$  is in  $\{f'_n\}$ , so the contradiction is false:  $\{f'_{n-1}\}$  is sufficient to generate  $\{f'_n\}$ .