

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-138
COMPUTER SCIENCE DEPARTMENT
REPORT NO. STAN-CS-71-188

THE TRANSLATION OF 'GO TO' PROGRAMS
TO 'WHILE' PROGRAMS

BY

EDWARD ASHCROFT

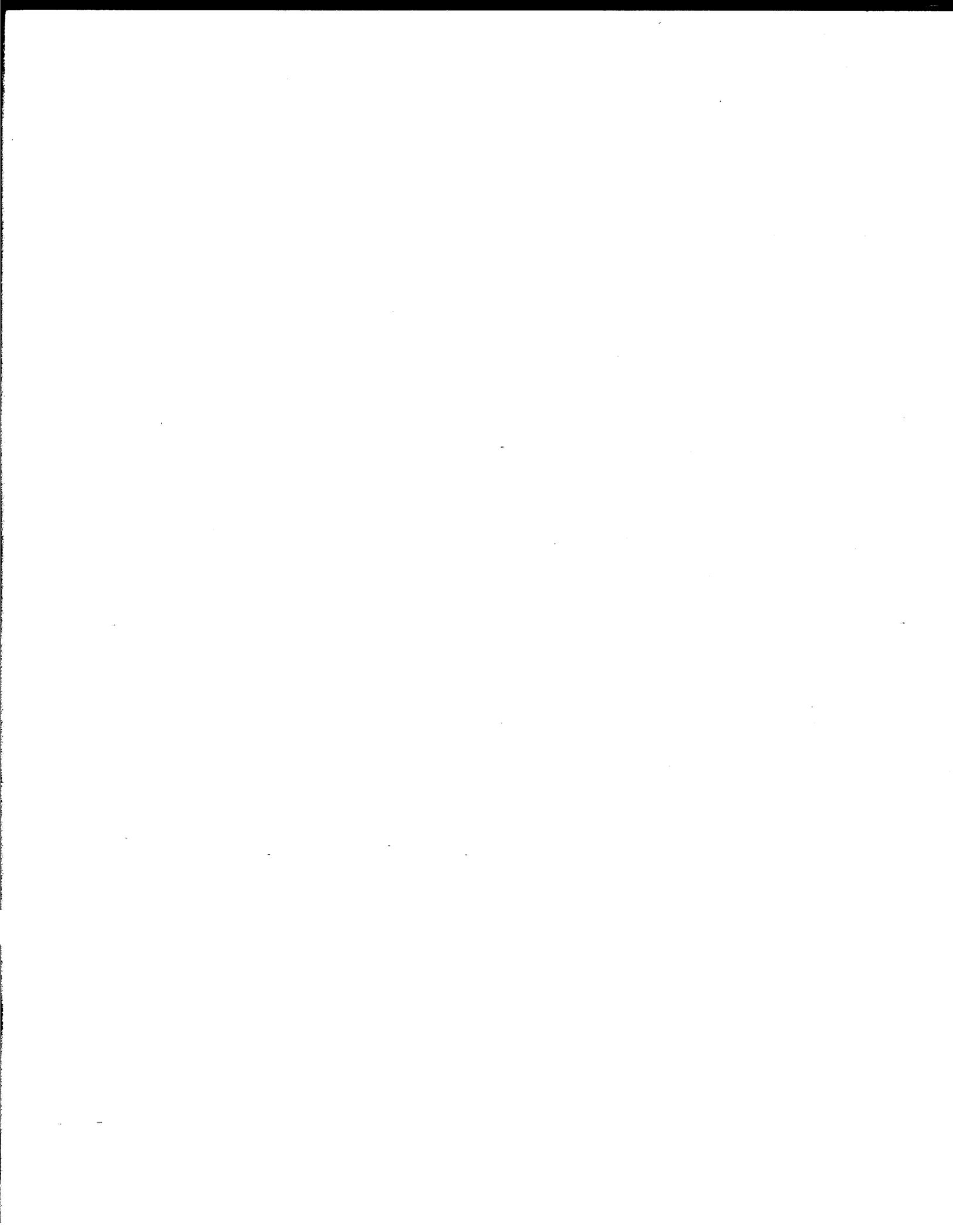
AND

ZOHAR MANNA

JANUARY 1971

COMPUTER SCIENCE DEPARTMENT
STANFORD UNIVERSITY





THE TRANSLATION OF 'GO TO' PROGRAMS
TO 'WHILE' PROGRAMS

by

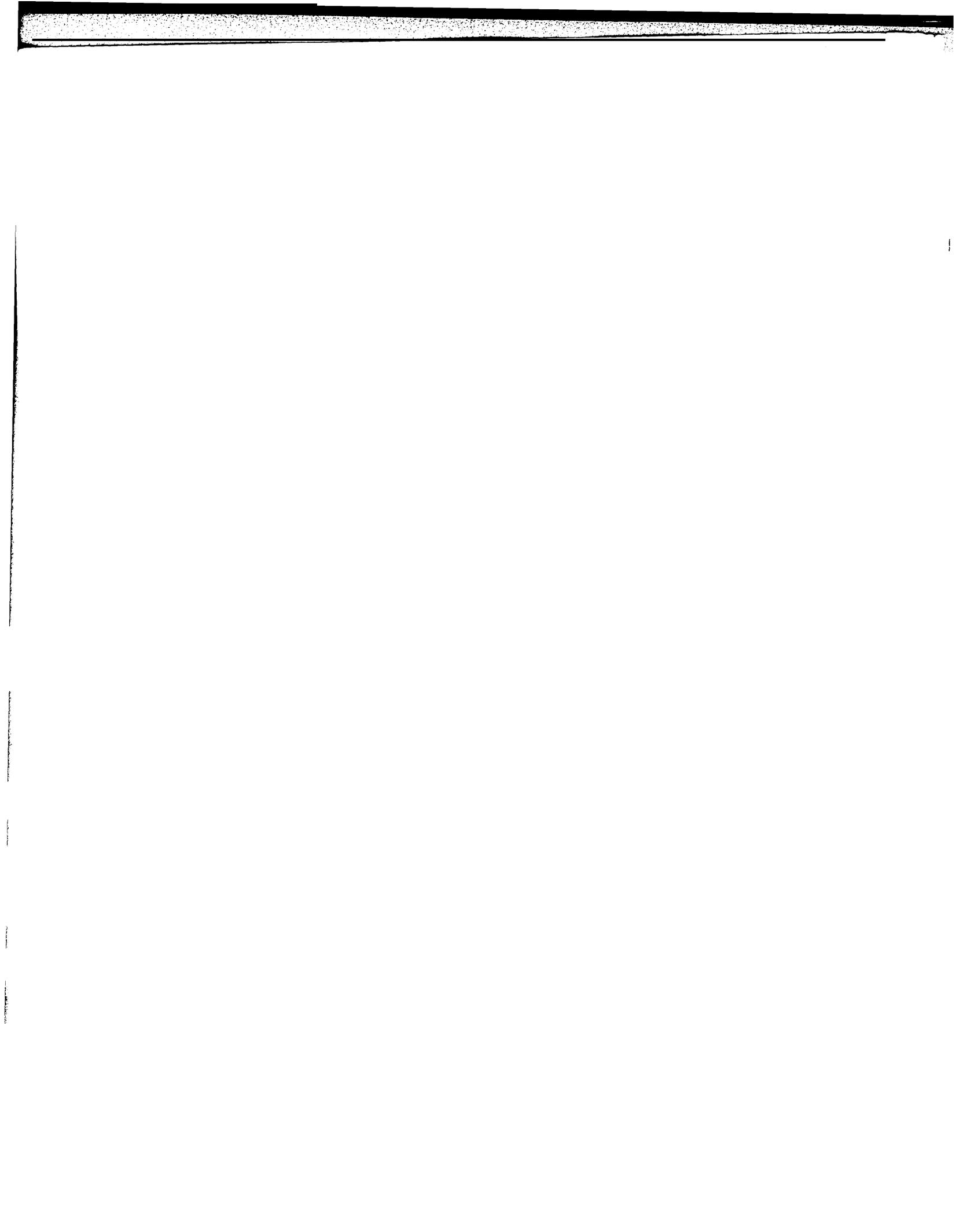
Edward Ashcroft

Zohar Manna

Abstract: In this paper we show that every flowchart program can be written without go to statements by using while statements. The main idea is to introduce new variables to preserve the values of certain variables at particular points in the program; or alternatively, to introduce special boolean variables to keep information about the course of the computation.

The 'while' programs produced yield the same final results as the original flowchart program but need not perform computations in exactly the **same** way. However, the new programs do preserve the 'topology' of the original flowchart program, and are of the **same** order of efficiency.

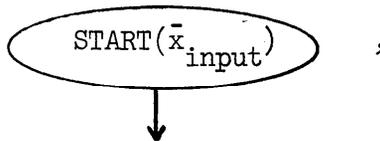
We also show that this cannot be done in general without adding variables.



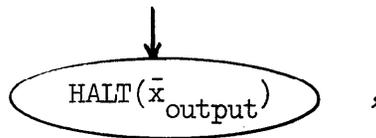
GENERAL DISCUSSION

1. Introduction

The first class of programs we consider are simple flowchart programs constructed from assignment statements (i.e., assigning terms to variables) and test statements (i.e., testing quantifier-free formulas) operating on a 'state vector' \bar{x} . The flowchart program begins with a unique start statement of the form



where \bar{x}_{input} is a subvector of \bar{x} , indicating the variables that have to be given values at the beginning of the computation. It ends with a unique halt statement of the form



where \bar{x}_{output} is a subvector of \bar{x} , indicating the variables whose values will be the desired result of the computation.

We make no assumptions about the domain of individuals, or about the operations and predicates used in the statements. Thus our flowchart programs are really flowchart schemas (see, for example, Luckham, Park and Paterson [1970]) and all the results can be stated in terms of such schemas.

Let P_1 be any flowchart program of the form shown in Figure 1. Note that, for example, the statement $\bar{x} \leftarrow e(\bar{x})$ stands for any sequence of assignment statements whose net effect is the replacement of vector \bar{x}

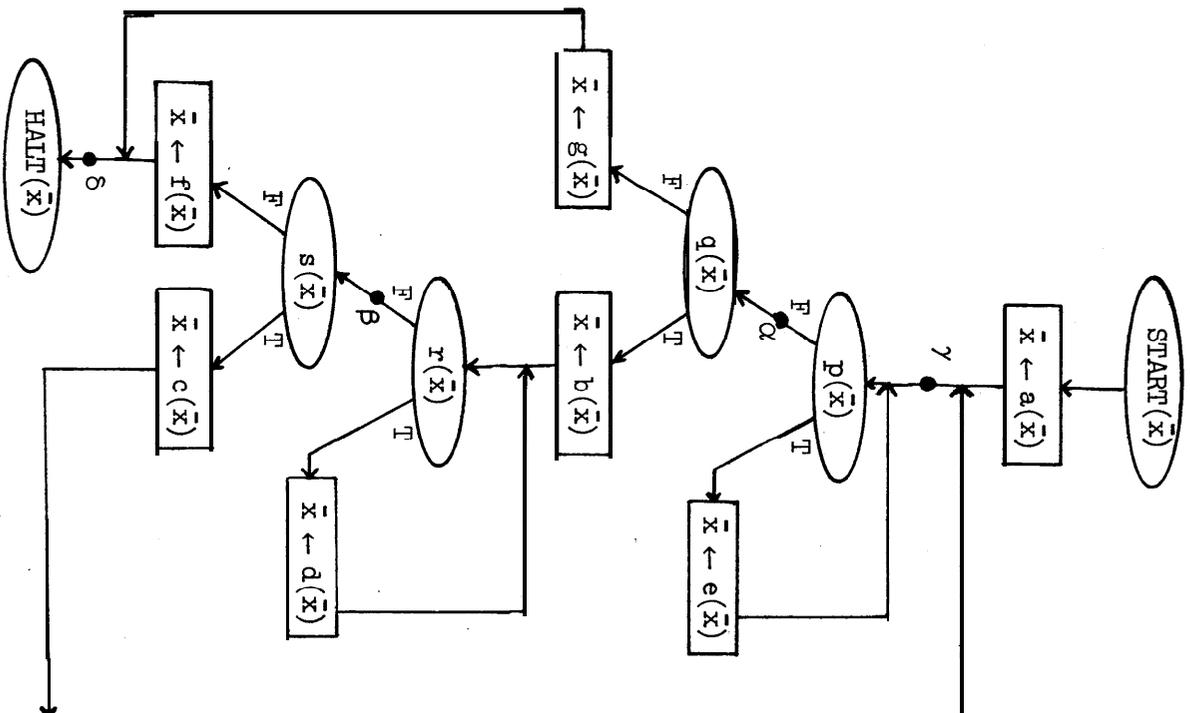


Figure 1. The Flowchart Program P_1 .

by a new vector $e(\bar{x})$. Similarly, the test $p(\bar{x})$, for example, stands for any quantifier-free formula with variables from \bar{x} . The flowchart program P_1 will be used as an example throughout the paper.

Flowchart programs are usually easy to understand, but if the program is to be written in a conventional programming language, goto statements are required. There has recently been much discussion (see, for example, Dijkstra [1968]) about whether the use of goto statements makes programs difficult to understand, and whether the use of while or for statements is preferable. It is clearly relevant to this discussion to consider whether the abolition of goto statements is really possible.

Therefore the second class of programs we consider are while programs, i.e., Algol-like programs consisting only of while statements of the form while (quantifier-free formula) do (statement), in addition to conditional, assignment and block^{*} statements. As before, each program starts with a unique start statement, $START(\bar{x}_{input})$, and ends with a unique halt statement, $HALT(\bar{x}_{output})$.

Since both classes of programs use the same kind of start and halt statements, we can define the equivalence of two programs independently of the classes to which they belong. Two programs (with the same length of input subvectors \bar{x}_{input} and the same length of output subvectors \bar{x}_{output}) are said to be equivalent if for each assignment of values to \bar{x}_{input} either both programs do not terminate or both terminate with the same values in \bar{x}_{output} .

^{*}/ A block statement is any sequence of statements enclosed by square brackets.

2. Translation to while programs by adding variables

(a) Extending the state vector \bar{x} .

We first show that by allowing extra variables which keep crucial past values of some of the variables in \bar{x} , one can effectively translate every flowchart program into an equivalent while program (ALGORITHM I). The importance of this result is that the original 'topology' of the program is preserved, and the new program is of the same order of efficiency as the original program. However, we shall not enter into any discussion as to whether the new program is superior to the original one or not.

This result, considered in terms of schemas, can be contrasted with those of Paterson and Hewitt [1970] (see also Strong [1970]). They showed that although it is not possible to translate all recursive schemas into flowchart schemas, it is possible to do this for 'linear' recursive schemas, by adding extra variables. However, as they point out, the flowchart schemas produced are much less efficient than the original recursive schemas.

As an example, ALGORITHM I will give the following while program which is equivalent to the flowchart program P_1 (Figure 1):

```

START(:);
 $\bar{x} \leftarrow a(\bar{x});$ 
[ while  $p(\bar{x})$  do  $\bar{x} \leftarrow e(\bar{x});$ 
   $\bar{y} \leftarrow \bar{x};$ 
  if  $q(\bar{x})$  then [ $\bar{x} \leftarrow b(\bar{x});$  while  $r(\bar{x})$  do  $\bar{x} \leftarrow d(\bar{x});$ ];
while  $q(\bar{y}) \wedge s(\bar{x})$  do
  [ $\bar{x} \leftarrow c(\bar{x});$ 
  [ while  $p(\bar{x})$  do  $\bar{x} \leftarrow e(\bar{x});$ 
   $\bar{y} \leftarrow \bar{x};$ 
  [ if  $q(\bar{x})$  then [ $\bar{x} \leftarrow b(\bar{x});$  while  $r(\bar{x})$  do  $\bar{x} \leftarrow d(\bar{x});$ ]];
if  $q(\bar{y})$  then  $\bar{x} \leftarrow f(\bar{x})$  else  $\bar{x} \leftarrow g(\bar{x});$ 
HALT( $\bar{x}$ ).

```

If the test $q(\bar{x})$ uses only a subvector of \bar{x} , then the algorithm will indicate that the vector of extra variables \bar{y} need only be of the same length as this subvector.

Note that on each cycle of the main while statement, the state vector \bar{x} is at point β , while \bar{y} holds the preceding values of \bar{x} at point α .

Note also that the two subprograms enclosed in broken lines are identical. This is typical of the programs produced by the algorithm. One might use this fact to make the programs more readable by using 'subroutines' for the repeated subprograms.

(b) Adding boolean variables.

Inspection of the above example will suggest that we do not need to introduce a whole vector \bar{y} , but rather a single boolean variable t which is assigned the value $q(\bar{x})$, as illustrated below. This while program, which is still equivalent to the program P_1 , will in practice be more efficient than the preceding while program, since t requires only one memory bit whereas \bar{y} may be a very large vector.

```

START(%);
 $\bar{x} \leftarrow a(X)$ ;
while  $p(\bar{x})$  do  $\bar{x} \leftarrow e(\bar{x})$ ;
 $t \leftarrow q(\bar{x})$ ;
if  $t$  then [ $\bar{x} \leftarrow b(\bar{x})$ ; while  $r(\bar{x})$  do  $\bar{x} \leftarrow d(\bar{x})$ ];
while  $t \wedge s(\bar{x})$  do
    [ $\bar{x} \leftarrow c(\bar{x})$ ;--
    while  $p(\bar{x})$  then  $\bar{x} \leftarrow e(\bar{x})$ ;
     $t \leftarrow q(\bar{x})$ ;
    if  $t$  then [ $\bar{x} \leftarrow b(\bar{x})$ ; while  $r(\bar{x})$  do  $\bar{x} \leftarrow d(\bar{x})$ ]];
if  $t$  then  $\bar{x} \leftarrow f(\bar{x})$  else  $\bar{x} \leftarrow g(g)$ ;
HALT( $\bar{x}$ ).

```

The translation of flowchart programs into while programs by the addition of boolean variables is not a new idea. Böhm and Jacopini [1966], Cooper [1967] and Bruno and Steiglitz [1970] have shown that every flowchart program can be effectively translated into an equivalent while program (with one while statement) by introducing new boolean variables into the program, new predicates to test these variables, together with assignments to set them true or false. The boolean variables essentially simulate a program counter, and the while program simply interprets the original program. On each repetition of the while statement, the next operation of the original program is performed, and the 'program counter' is updated. As noted by Cooper and Bruno and Steiglitz themselves, this transformation is undesirable since it changes the 'topology' of the program, giving a program that is less easy to understand. For example, if a while program is written as a flowchart program and then transformed back to an equivalent while program by their method, the resulting while program will not resemble the original.

We give an algorithm (ALGORITHM II) for transforming flowchart programs to equivalent while programs by adding extra boolean variables, which is an improvement on the above methods. It preserves the 'topology' of the original program and in particular it does not alter while-like structure that may already exist in the original program.

For the flowchart program P_1 , for example, ALGORITHM II will produce the following while program.

```

START( $\bar{x}$ );
 $\bar{x} \leftarrow a(\bar{x})$ ;
t true;
while t do
    [while p( $\bar{x}$ ) do  $\bar{x} \leftarrow e(\bar{x})$ ;
    if q( $\bar{x}$ ) then [ $\bar{x} \leftarrow b(\bar{x})$ ;
        while r( $\bar{x}$ ) do  $\bar{x} \leftarrow d(\bar{x})$ ;
        if s( $\bar{x}$ ) then  $\bar{x} \leftarrow c(\bar{x})$ 
            else [ $\bar{x} \leftarrow f(\bar{x})$ ; t false]]
    else [ $\bar{x} \leftarrow g(\bar{x})$ ; t false]];
HALT( $\bar{x}$ ).

```

Note that each repetition of the main while statement starts from point y and proceeds either back to y or to δ . In the latter case, t is made false and we subsequently exit from the while statement.

3. Translation to while programs without adding variables

It is natural at this point to consider whether every flowchart program can be translated into an equivalent while program without adding extra variables (i.e., using only the original state vector \bar{x}). We show that this cannot be done in general, and in fact there is a flowchart program of the form of Figure 1 which is an appropriate counter-example.

A similar negative result has been demonstrated by Knuth and Floyd [1970] and Bruno and Steiglitz [1970]. However, the notion of equivalence considered by those authors is more restrictive in that it requires equivalence of computation sequences (i.e., the sequence of assignment and test statements in order of execution) and not just the equivalence

of final results of computation as we do. Thus, since our notion of equivalence is weaker, our negative result is stronger.

Our counter-example is a program of the form of Figure 1 in which:

r identical to q , s identical to p , b and e identical to g , and c and d identical to f .

There is also another similar counter-example in which:

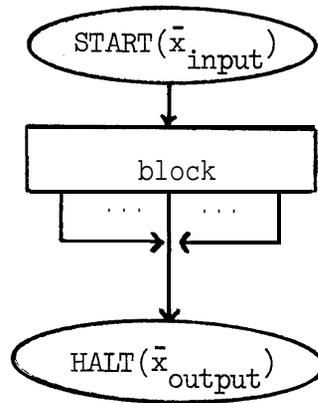
r identical to q , s identical to p , d and e identical to g , and b and c identical to f .

The fact that these restricted forms are counter-examples is especially interesting since we have found while programs, with no extra variables, which are equivalent (in our sense) to most of the programs of the form of Figure 1. In particular, we can do this for any flowchart program of the form of Figure 1 with only two distinct tests and two distinct operations in which

c is identical to e ,
or b is identical to d ,
or f is identical to g .

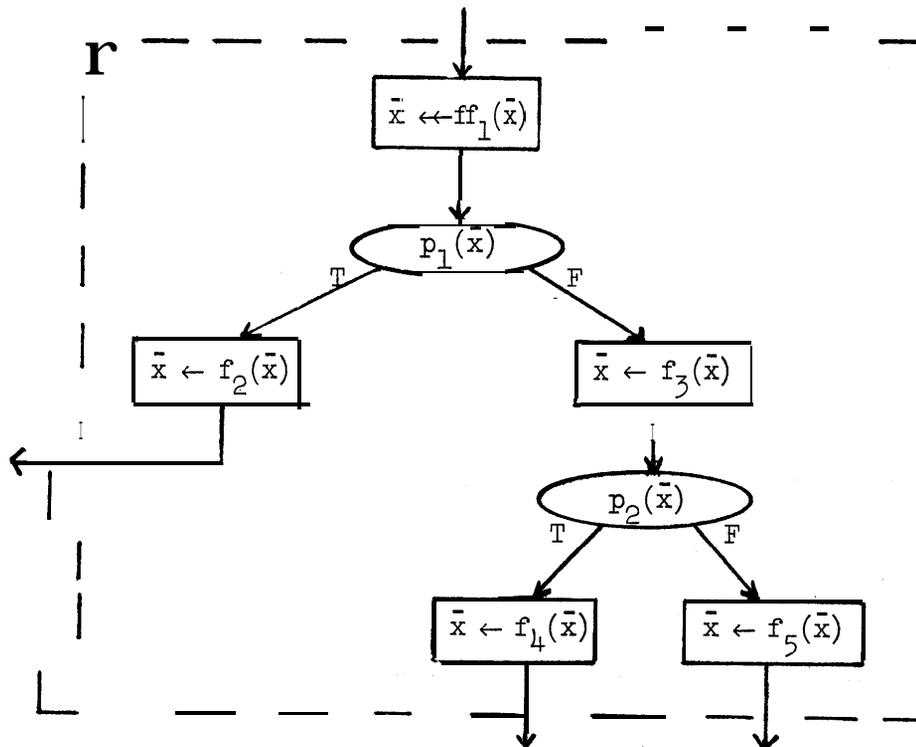
ALGORITHM I: TRANSLATION BY EXTENDING THE STATE VECTOR \bar{x}

Our algorithm depends on the fact that every flowchart program can be put effectively into a normal form (see Cooper [1970] and Engeler [1970]). A flowchart program is in normal form if it is of the form

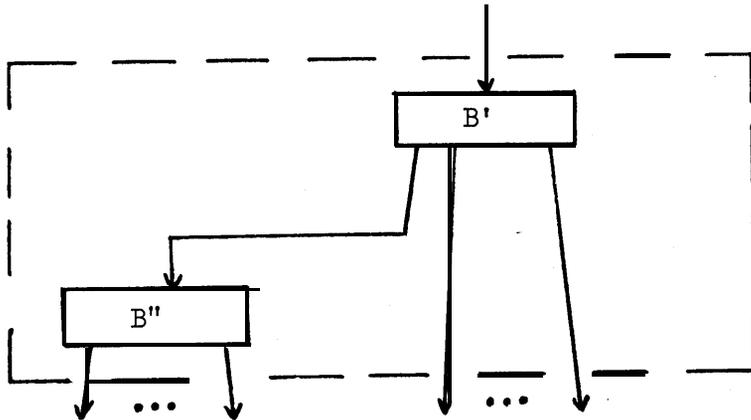
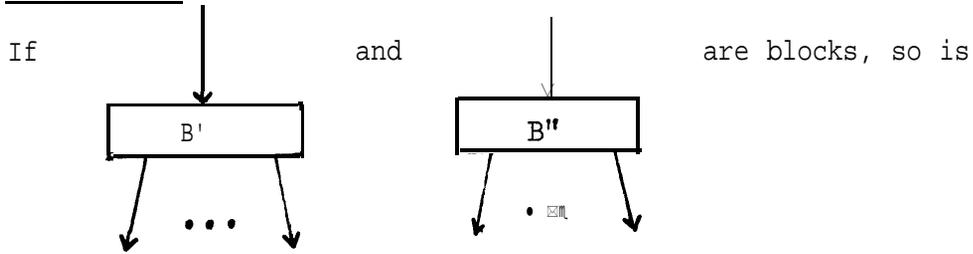


where a block is defined recursively as follows:

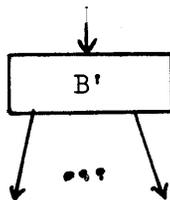
1. A basic block is any tree-like, loop-free, one-entrance piece of flowchart program (without start and halt statements). For example,

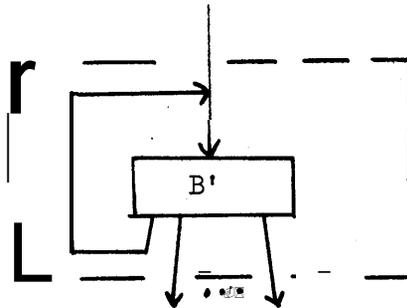


2. Composition



3. Looping

If  is a block, so is



We shall consider only flowchart programs in normal form. By induction on the structure of the blocks we show how to associate with each block $B(\bar{x})$ (with state vector \bar{x}) a piece of while program $\alpha_B(\bar{x}, \bar{y})$,^{*} and with the i -th exit of the block a pair $\langle \varphi_i(\bar{x}, \bar{y}), \tau_i(\bar{x}) \rangle$, where $\varphi_i(\bar{x}, \bar{y})$ is the 'exit-condition' and $\tau_i(\bar{x})$ is the 'exit-term',

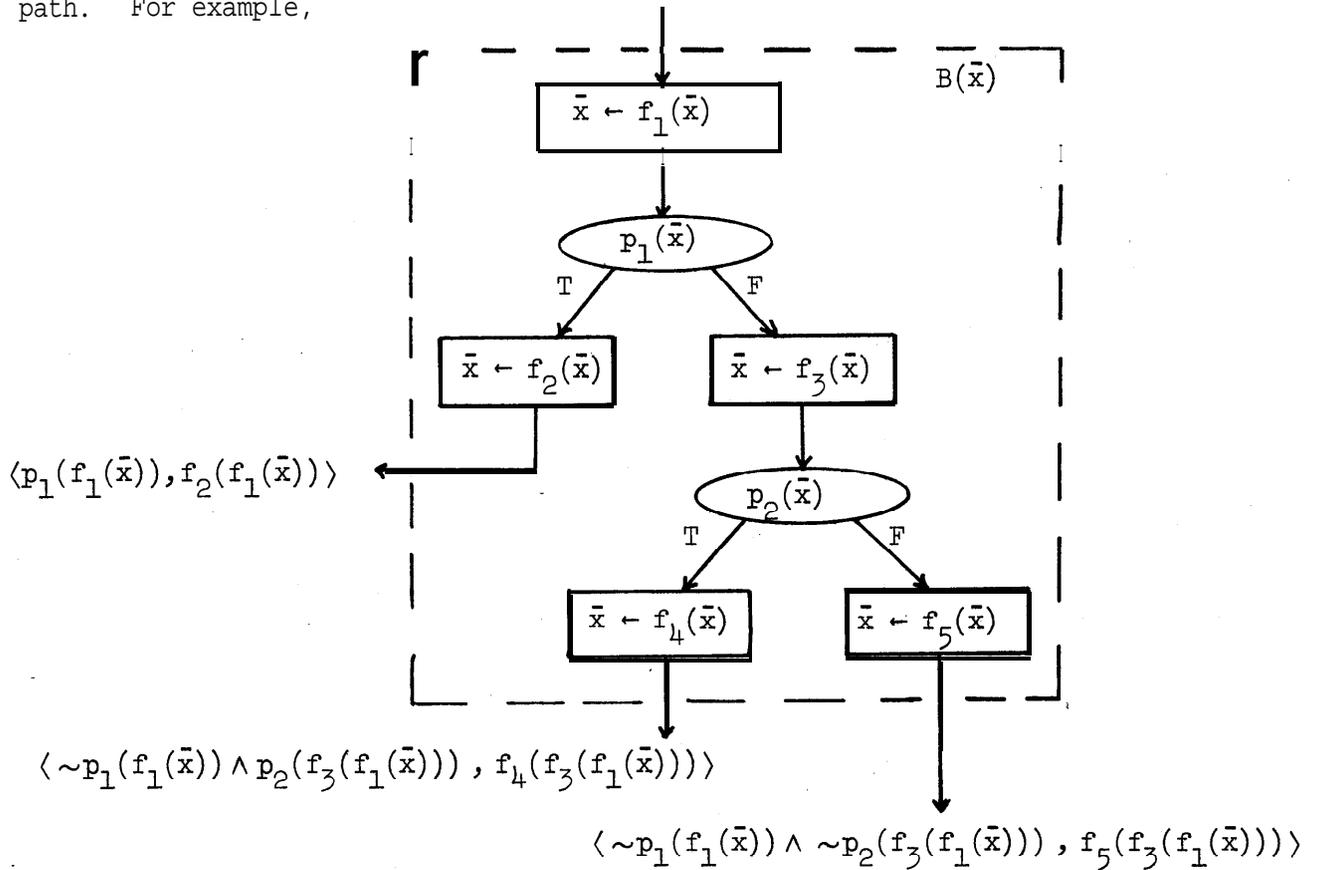
^{*} \bar{y} is a (possibly empty) vector of additional variables introduced by the translation.

such that $B(\bar{x})$ comes out of the i -th exit with $\bar{\xi}$ if and only if $\alpha_B(\bar{x}, \bar{y})$ terminates with some $\bar{\xi}'$ s.t. $\varphi_i(\bar{\xi}', \bar{y}) = T$ and $\bar{\xi} = \tau_i(\bar{\xi}')$. Each φ_i is a quantifier free formula constructed from the tests and operations in the flowchart. The φ_i 's for a given block are complete and mutually exclusive.

In each of the three cases we have to consider, the above relationship between α_B and $\langle \varphi_i, \tau_i \rangle$ is preserved.

1. $B(\bar{x})$ is a basic block (i.e., tree-like, loop-free, one-entrance flowchart)

In this case $\alpha_B(\bar{x})$ is always null (the empty program), $\varphi_i(\bar{x})$ is the condition that control will take exit i for input \bar{x} , and $\tau_i(\bar{x})$ is the result of performing the assignment statements on the corresponding path. For example,

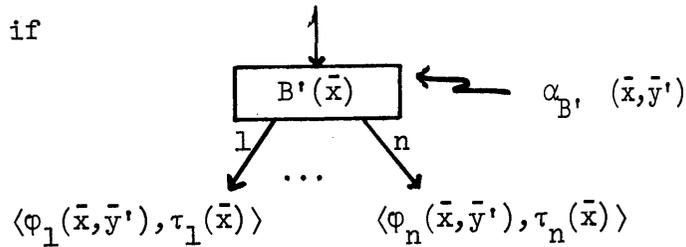


where $\alpha_B(\bar{x})$ is null.

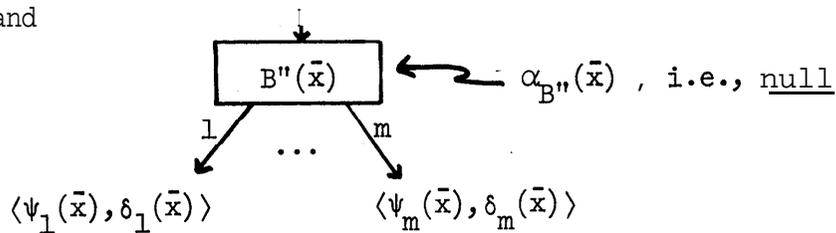
2. $B(\bar{x})$ is constructed from $B'(\bar{x})$ and $B''(\bar{x})$ by composition

We consider two cases:

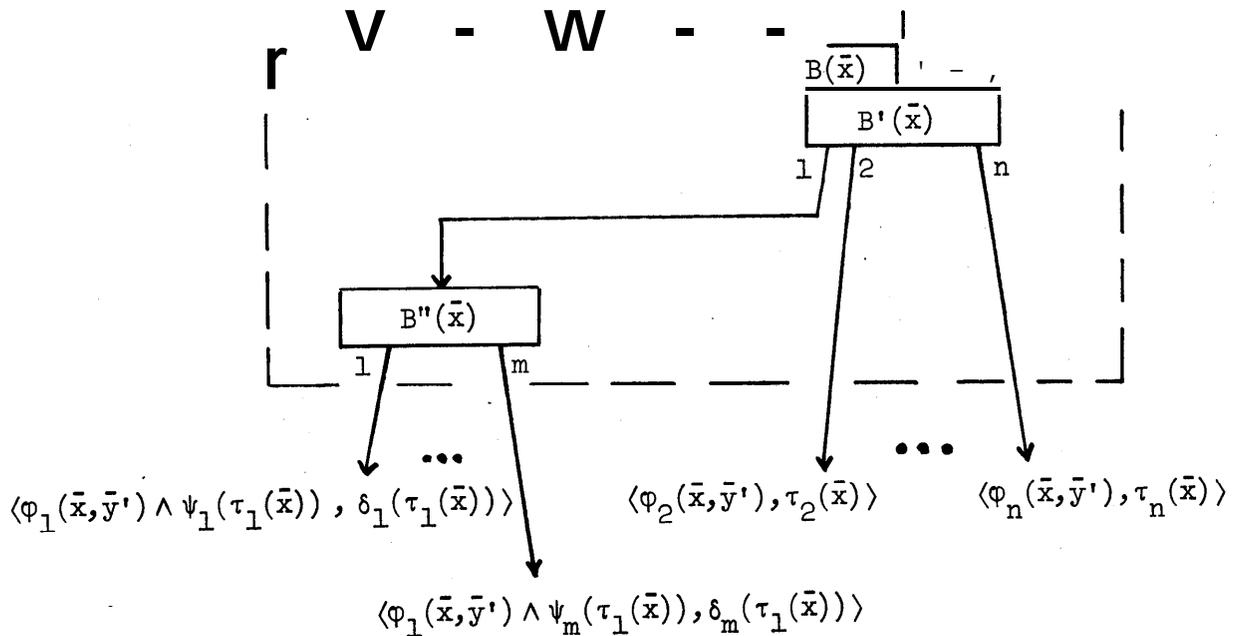
(a) B'' is a basic block.



and



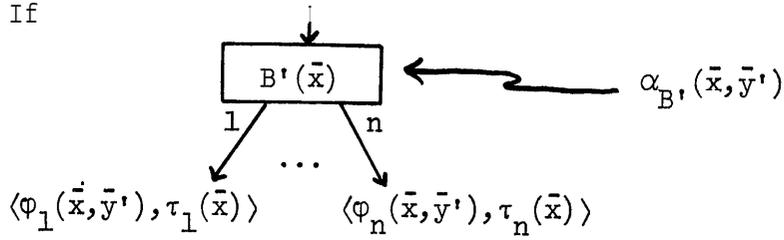
then



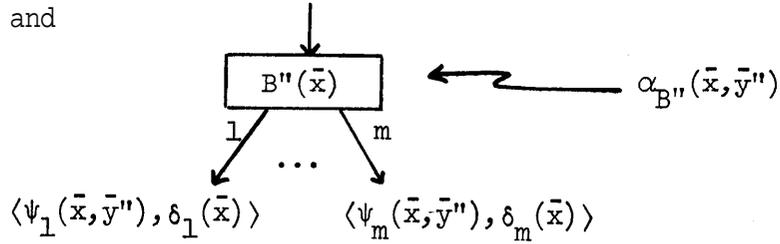
where $\alpha_B(\bar{x}, \bar{y}')$ is $\alpha_{B'}(\bar{x}, \bar{y}')$.

(b) B'' is a non-basic block.

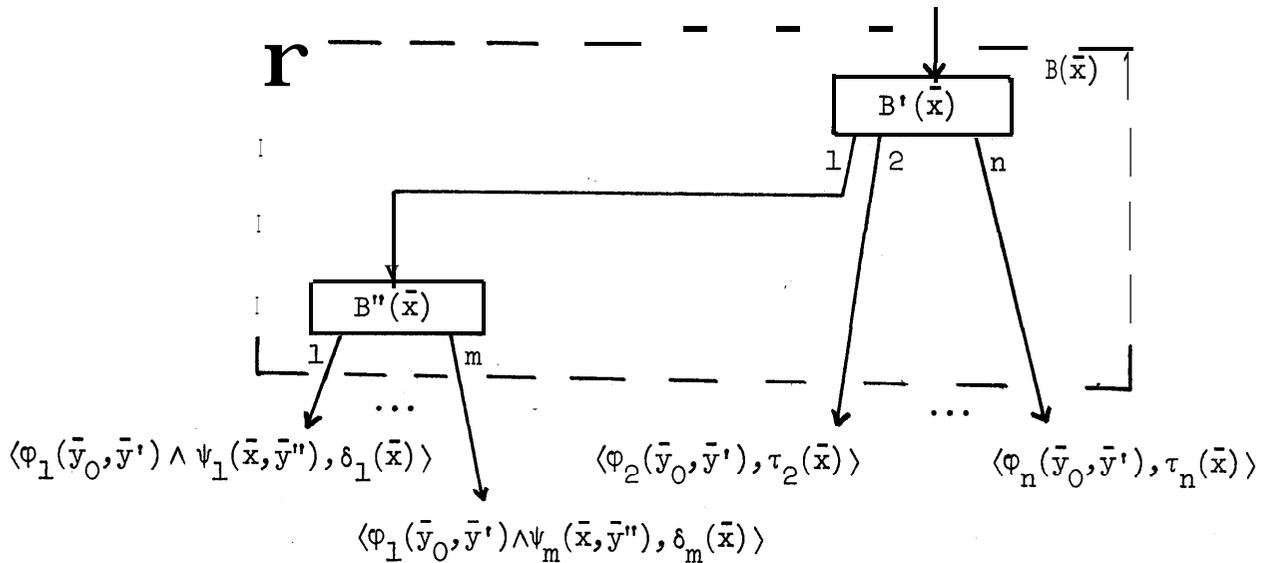
If



and



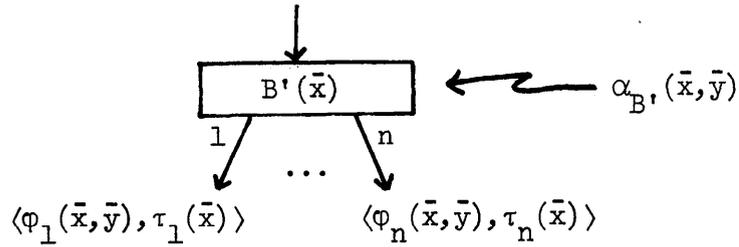
then let \bar{x}_0 be the subvector of \bar{x} which is used in exit conditions $\varphi_1, \dots, \varphi_n$. Let \bar{y}_0 be a new vector of the same length as \bar{x}_0 . Then $\bar{y} = \{\bar{y}', \bar{y}'', \bar{y}_0\}$ and



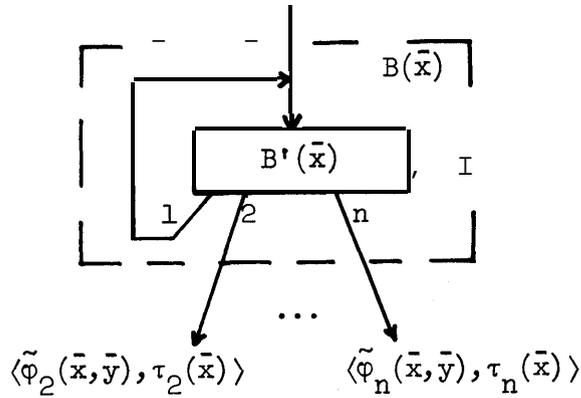
where $\alpha_B(\bar{x}, \bar{y})$ is $\begin{cases} \alpha(\bar{x}, \bar{y}'); \\ \bar{y}_0 \leftarrow \bar{x}_0; \\ \text{if } \varphi_1(\bar{x}, \bar{y}') \text{ then } [\bar{x} \leftarrow \tau_1(\bar{x}); \alpha_{B''}(\bar{x}, \bar{y}'')] \end{cases}$

3. $B(\bar{x})$ is constructed from $B'(\bar{x})$ by looping.

If



then



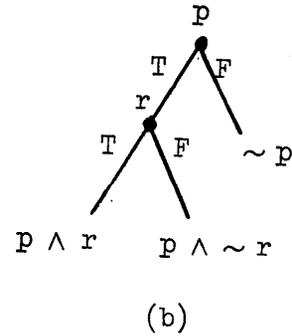
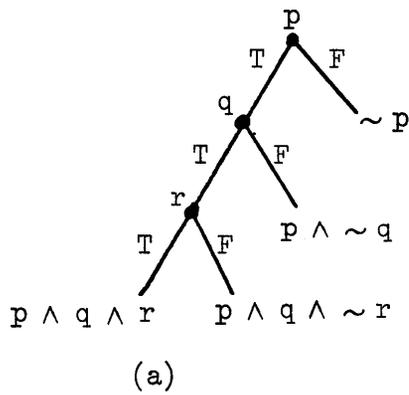
where $\alpha_B(\bar{x}, \bar{y})$ is $\alpha_{B', (\bar{x}, \bar{y})}$;

while $\varphi_1(\bar{x}, \bar{y})$ do $[\bar{x} \leftarrow \tau_1(\bar{x}); \alpha_{B', (\bar{x}, \bar{y})}]$

and $\tilde{\varphi}_2, \dots, \tilde{\varphi}_n$ are complete and mutually exclusive and

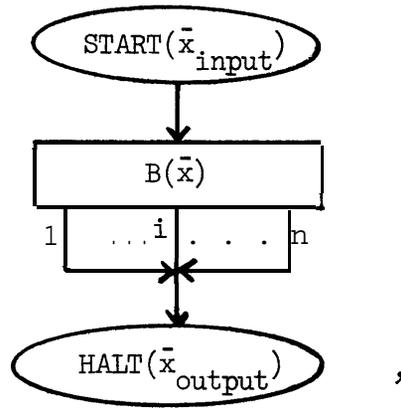
$\varphi_j(x, y) \supset \tilde{\varphi}_j(x, y)$, $2 \leq j \leq n$.

Comment: To find $\{\tilde{\varphi}_j\}$ note that the algorithm ensures that each φ_j is a conjunction of literals (i.e., atomic formulas and negations of atomic formulas), and therefore we can represent $\{\varphi_j\}$ by a binary tree; e.g. $\{p \wedge q \wedge r, p \wedge q \wedge \sim r, p \wedge \sim q, \sim p\}$ is represented by tree (a).



If we remove the node in the tree leading directly to the terminal node representing ϕ_1 , the new tree represents the desired conditions $\{\tilde{\phi}_{j3}\}$. For example, if we remove $p \wedge \sim q$ from the above set of conditions, we get the new tree (b) which represents the new set of exit conditions $\{p \wedge q, p \wedge \sim r, \sim p\}$.

Conclusion: This covers all cases of blocks we need to consider. To find the while program equivalent to a given flowchart program (in normal form)



we find $\alpha_B(\bar{x}, \bar{y})$ and $\{(\phi_i(\bar{x}, \bar{y}), \tau_i(\bar{x}))\}$. The desired while program is then

```

START( $\bar{x}_{input}$ )
 $\alpha_B(\bar{x}, \bar{y})$ ;
if  $\phi_1(\bar{x}, \bar{y})$  then  $\bar{x} \leftarrow \tau_1(\bar{x})$ 
    else if  $\phi_2(\bar{x}, \bar{y})$  then  $\bar{x} \leftarrow \tau_2(\bar{x})$ 
        else . . .  $\bar{x} \leftarrow \tau_n(\bar{x})$ ;
HALT( $\bar{x}_{output}$ ) .

```

Example: Let us consider **again** the flowchart Program P_1 (Figure 1). It is already in normal form, and the blocks are indicated in Figure 2. The exit conditions and exit terms for the exits of all blocks are also indicated. The corresponding α 's are given below:

α_{B_1} is nu(\bar{x}).

α_{B_2} is whi(\bar{x}) p(\bar{x}) do $\bar{x} \leftarrow e(\bar{x})$.

$\alpha_{B_3}(\bar{x})$ is null.

α_{B_4} is whi(\bar{x}) r(\bar{x}) do $\bar{x} \leftarrow d(\bar{x})$

$\alpha_{B_5}(\bar{x}, \bar{y})$ is $\alpha_{B_2}(\bar{x}); \bar{y} \leftarrow \bar{x}_0; \underline{\text{if}} \ q(\bar{x}) \ \underline{\text{then}} \ [\bar{x} \leftarrow b(\bar{x}); \alpha_{B_4}(\bar{x})]$

Note that \bar{x}_0 is the subvector of \bar{x} occurring in the exit conditions of B_2 , i.e., in $q(\bar{x})$.

$\alpha_{B_6}(\bar{x}, \bar{y})$ is. $\alpha_{B_5}(\bar{x}, \bar{y}); \underline{\text{while}} \ q(\bar{y}) \wedge s(\bar{x}) \ \underline{\text{do}} \ [\bar{x} \leftarrow c(\bar{x}); \alpha_{B_5}(\bar{x}, \bar{y})]$

Thus the original flowchart program is equivalent to the following while program;

```

START(%);
 $\bar{x} \leftarrow a(\bar{x});$ 
while  $p(\bar{x})$  do  $\bar{x} \leftarrow e(\bar{x});$ 
 $y \leftarrow x_0;$ 
if  $q(\bar{x})$  then [ $\bar{x} \leftarrow b(\bar{x});$  while  $r(\bar{x})$  do  $\bar{x} \leftarrow d(\bar{x})$ ];
while  $q(\bar{y}) \wedge s(\bar{x})$  do
    [ $\bar{x} \leftarrow c(\bar{x});$ 
    while  $p(\bar{x})$  do  $\bar{x} \leftarrow e(\bar{x});$ 
     $y \leftarrow x_0;$ 
    if  $q(\bar{x})$  then [ $\bar{x} \leftarrow b(\bar{x});$  while  $r(\bar{x})$  do  $\bar{x} \leftarrow d(\bar{x})$ ]];
if  $q(\bar{y})$  then  $\bar{x} \leftarrow f(\bar{x})$  else  $\bar{x} \leftarrow g(\bar{x});$ 
HALT( $\bar{x}$ ).

```

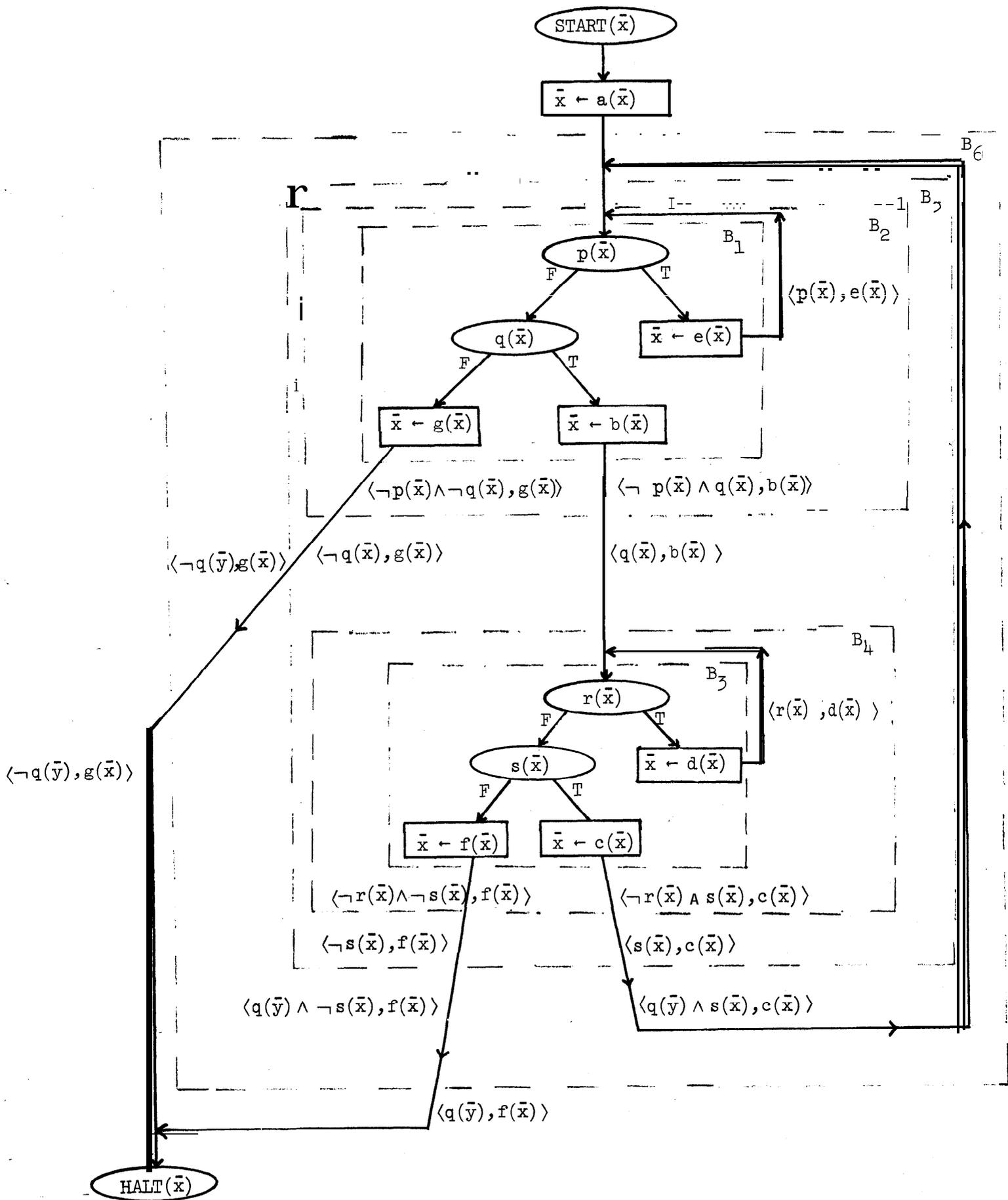
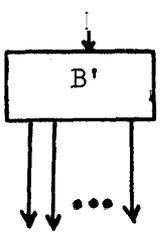
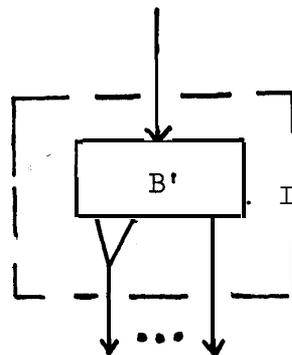


Figure 2. The flowchart program P_1 (for ALGORITHM I).

Comment: In general the transformation of a program to normal form results in **exponential growth** in the size of the program. This can be reduced if we allow the following extra case in the definition of blocks.

4. Merging (optional)

If  is a block, so is



The algorithm can be easily modified to cover this case, but since it would complicate our notation, we will not discuss it here.

ALGORITHM II: TRANSLATION BY ADDING BOOLEAN VARIABLES

The second algorithm, ALGORITHM II, translates flowchart programs to equivalent while programs by adding boolean variables. It makes use of the fact that every flowchart program (without the start and halt statements) can be decomposed into blocks where a block is any piece of flowchart program with only one exit (but possibly many entrances).^{*/} This is obvious since in particular the whole body of the given flowchart program can be considered as such a block. The aim, whenever possible, is to get blocks containing at most one top-level test statement (i.e., test statement not contained in inner blocks) since such blocks can be represented as a piece of while program without adding boolean variables. In particular, if a while program is expressed as a flowchart program, this latter program can always be decomposed into such simple blocks, and the algorithm will give us back the original while program.

For any given flowchart program we construct the equivalent while program by induction on the structure of the blocks. Since the ideas behind the construction are intuitively simpler, we shall not be as formal as in the presentation of ALGORITHM I.

For each entrance b_i to block B we consider that part B_i of the block reachable from b_i . We then recursively construct an equivalent piece of while program $\gamma_{B_i}(\bar{x}, \bar{t})$ ^{**/} as follows. There are two cases to consider:

^{*/} Note that the blocks used here are not related in any way to those used in ALGORITHM I.

^{**/} \bar{t} is a (possibly empty) vector of additional boolean variables introduced by the translation.

Case 1: (a) B_i contains at most one top-level test statement,
or (b) B_i contains no top-level loops.

In both cases $\gamma_{B_i}(\bar{x}, \bar{t})$ is the obvious piece of while program requiring at most one top-level while statement (and no extra boolean variables).

Case 2: B_i contains two or more top-level test statements and at least one loop.

In this case we choose a set of points on top-level arcs of B_i (called 'cut-set' points) such that each loop contains at least one such point. One point on the exit arc of the block is also included in this set. We shall translate B_i into a piece of while program $\gamma_{B_i}(\bar{x}, \bar{t})$ with one top-level while statement in such a way that each iteration of the while statement follows the execution of B_i from one cut-set point to the next. In this case, $\gamma_{B_i}(\bar{x}, \bar{t})$ includes boolean variables introduced to keep track of the current cut-set point. Note that n boolean variables t_1, t_2, \dots, t_n are sufficient to distinguish between k cut-set points, $2^{n-1} < k < 2^n$.

Example: We shall illustrate the method using again the flowchart program P_1 (Figure 1). We decompose P_1 into blocks as shown in Figure 3. Blocks B_1 and B_2 are of type 1 and can each be written as a single while statement. Block B_3 is of type 2 with a single top-level loop. Thus it is sufficient to choose points α and β as the cut-set points. To distinguish between α and β we need one boolean variable, t say. Thus the following while program, using the boolean variable t , can be generated and it is equivalent to the given flowchart program P_1 .

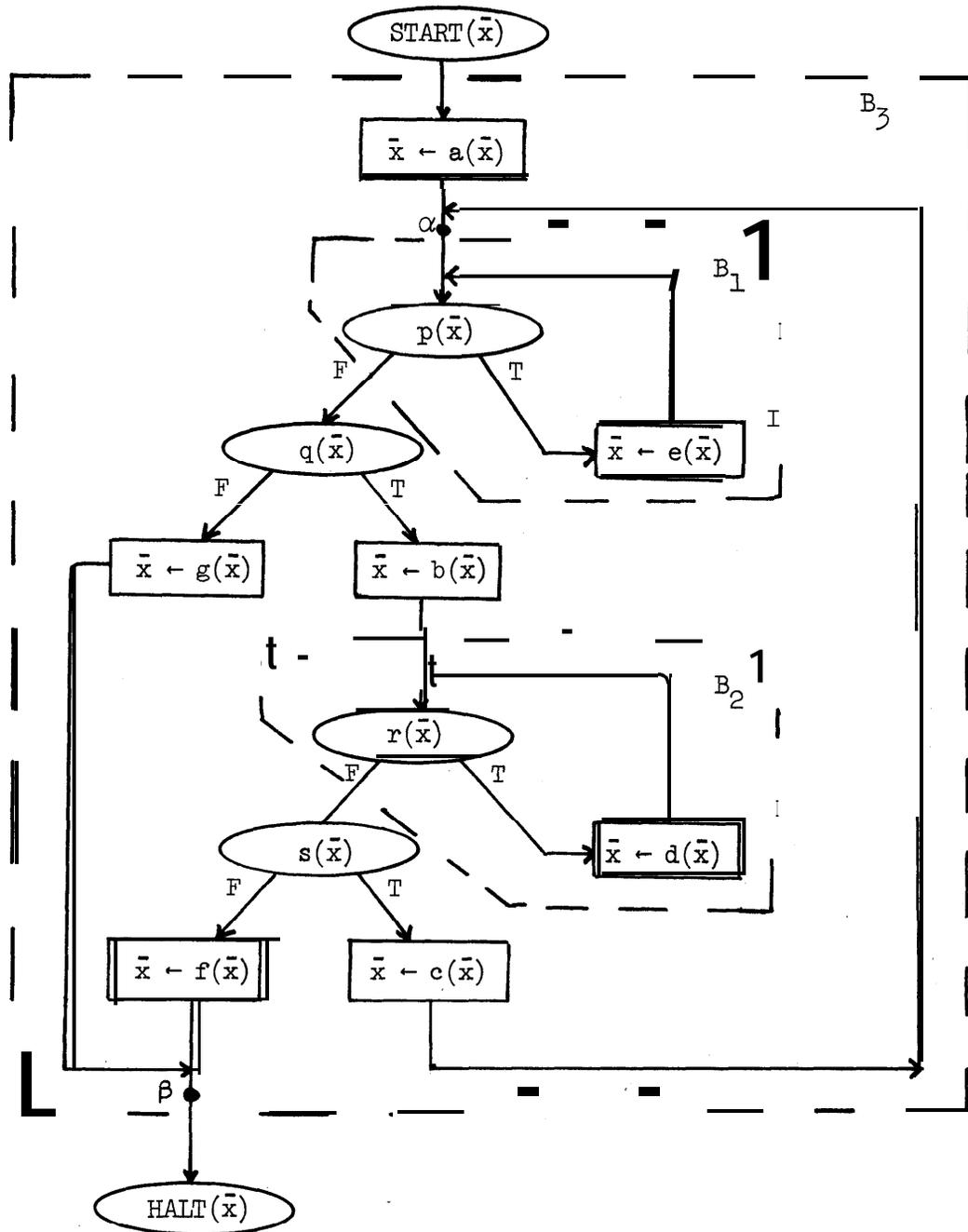


Figure 3. The flowchart program P_1 (for ALGORITHM II).

```

START(%);
 $\bar{x} \leftarrow a(\bar{x});$ 
 $t \leftarrow \underline{\text{true}};$ 
while  $t$  do
  [while  $p(\bar{x})$  do  $\bar{x} \leftarrow e(\bar{x});$ 
  if  $q(\bar{x})$  then [ $\bar{x} \leftarrow b(\bar{x});$ 
    while  $r(\bar{x})$  do  $\bar{x} \leftarrow d(\bar{x});$ 
    if  $s(\bar{x})$  then  $\bar{x} \leftarrow c(\bar{x})$ 
    else [ $\bar{x} \leftarrow f(\bar{x}); t \leftarrow \underline{\text{false}}$ ]]
  else [ $\bar{x} \leftarrow g(\bar{x}); t \leftarrow \underline{\text{false}}$ ]];
HAIT( $\bar{x}$ ) .

```

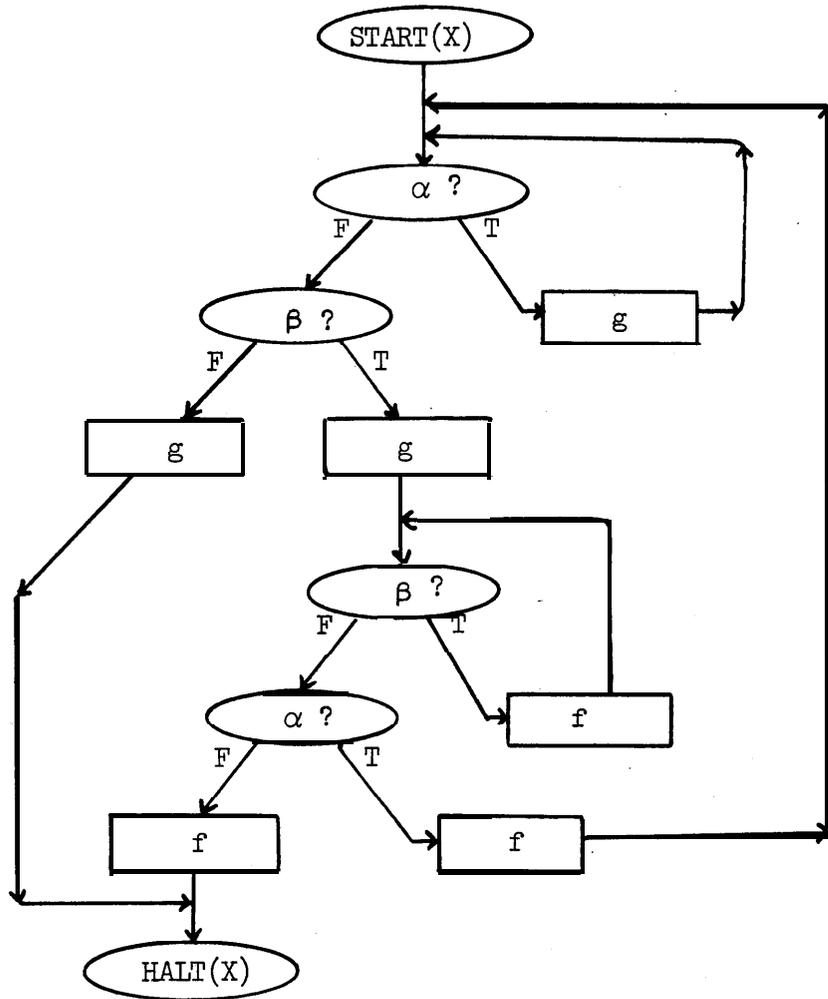
THE NEGATIVE RESULT

We consider the flowchart program P_2 (Figure 4) which has the structure of Figure 1. The domain D is the set of all pairs of strings such that the first string, called 'head', is any finite string over letters $\{f,g\}$, and the second string, called 'tail', is any infinite^{*/} string over letters $\{\alpha,\beta,\gamma\}$ with at most one occurrence of γ .

During a computation of P_2 , the only changes in the value of the program variable are deletion of leftmost letters from the tail and adding letters f or g to the right of the head. The tests in the program simply look at the tail, and-therefore the computation is determined by the tail of the initial value. Thus, since the program terminates if and only if both tests α and β are false, it implies that P_2 terminates if and only if the tail of the initial value contains γ . Another important feature of any computation of P_2 is that whenever the leftmost letter of the tail is α , the next but one operation must be operation g . Similarly, whenever the leftmost letter is β , the next but one operation must be f .

Let us assume that we have a while program P_2^* equivalent to P_2 which also has one variable and the same domain D . Although the assignment statements of P_2^* may use any terms obtained by compositions of the operations f and g , we assume without loss of generality that each assignment statement in P_2^* consists of a single operation f or g . The tests in the conditional and while statements may only use quantifier-free formulas obtained from tests α and β , and operations f and g . Since we use

^{*/} Note that the domain is non-enumerable. However, we can in fact restrict the tails to the enumerable domain of ultimately periodic strings, i.e., infinite strings which eventually repeat some finite substring indefinitely.



where test α means "is letter ' α ' the leftmost letter in tail";
 test β means "is letter ' β ' the leftmost letter in tail";
 operation f means "erase the leftmost letter in tail and add
 letter ' f ' on the right of head"; and
 operation g means "erase the leftmost letter in tail and add
 letter ' g ' on the right of head".

Figure 4. The Flowchart Program P_2 (for negative result)

only one variable, it follows that every sequence of values describing a computation of P_2^* is identical to the corresponding computation of P_2 . Note also that since there is a bound on the depth of terms in the quantifier-free formulas, there is a bound, M say, on the number of leftmost letters in the tail that can affect the decision of any test in P_2^* . Finally, without loss of generality we shall make the restriction that there is no redundant while statement in P_2^* ; i.e., there is no while statement with a uniform bound on the number of its iterations.

Since P_2^* must contain some (non-redundant) while statement, let W be any while statement in P_2^* which is not contained or followed by another while statement. The point in P_2^* immediately after W we shall denote by A .

Lemma

For all n ($n > 0$) there exists strings $a, c \in \{\alpha, \beta\}^*$ and $d \in \{\alpha, \beta\}^\infty$ ($|c| = n$) ^{*/} such that for all strings $b \in \{\alpha, \beta\}^*$ the computation starting with tail $abcyd$ passes A with some tail $\underline{ab}cyd$, where \underline{ab} is some rightmost substring of ab (possibly empty).

From this Lemma we immediately obtain the following corollary.

Corollary

For every n , $n > 0$, there exists a finite computation of P_2^* which passes through A with more than n operations still to be performed.

But this contradicts the fact that, since there is no while statement following A , the number of operations that P_2^* can perform after A is bounded.

^{*/} i.e., a and c are finite strings (possibly empty) over $\{\alpha, \beta\}$, d is an infinite string over $\{\alpha, \beta\}$ and the length of c is n .

Proof of Lemma. By induction on n .

Base step. Choose any computation starting with tail $a'a''b'\gamma d'$ ($a', a'', b' \in \{\alpha, \beta\}^*$, $d' \in \{\alpha, \beta\}^\infty$ and $|a''| = M$) that enters W with tail $a''b'\gamma d'$. (Such computation exists by non-redundancy of W .)

Since at most M leftmost letters of the tail can effect the decision of any test, on entering W the main test can only look at a'' . Therefore the test will be true for any tail starting with a'' .

In particular, the computation starting with tail $a'a''b\gamma a''d'$, for any b in $\{\alpha, \beta\}^*$, also enters W at the same point, i.e., with tail $a''b\gamma a''d'$. It must subsequently pass point A , but (noting that the test in W must be false when passing A) it cannot pass A with tail $a''d'$.

Hence, with $a = a'a''$, $d = a''d'$, for all strings b in $\{\alpha, \beta\}^*$, the computation starting with $abyd$ must pass A with some tail $\underline{ab}yd$ where \underline{ab} is some rightmost substring of ab .

Induction step. Assume we have strings $a, c \in \{\alpha, \beta\}^*$ and $d \in \{\alpha, \beta\}^\infty$, $|c| = n$, such that for all strings b in $\{\alpha, \beta\}^*$ the computation starting with tail $abcyd$ passes A with some tail $\underline{abc}yd$ where \underline{ab} is some rightmost substring of ab .

We find a string $c' \in \{\alpha, \beta\}^*$, $|c'| = n+1$, such that for all strings b' in $\{\alpha, \beta\}^*$ the computation starting with tail $ab'c'\gamma d$ passes A with some tail $\underline{ab'c'}\gamma d$ where $\underline{ab'}$ is some rightmost substring of ab' .

There are three cases to consider:

(i) For all non-empty strings b , the corresponding substring \underline{ab} is non-empty. In this case we take c' to be αc ^{*/}

For any string b' in $\{\alpha, \beta\}^*$ the computation starting with tail $ab'\alpha cyd$, passes A with tail $\underline{ab'\alpha cyd}$, where $\underline{ab'}$ is a rightmost substring of ab' .

(ii) For some non-empty string $b = b''\alpha$ ($b'' \in \{\alpha, \beta\}^*$), the substring \underline{ab} is empty, i.e., there exists computation S starting with $ab''\alpha cyd$ that passes A with tail cyd . In this case we take c' to be βc .

By earlier remarks about P_2 and P_2^* , it follows that the next operation in S after passing A must be g .

Now, for any string b' in $\{\alpha, \beta\}^*$ the computation starting with tail $ab'\beta cyd$ must pass A with some tail $\underline{ab'\beta cyd}$ where $\underline{ab'\beta}$ is some rightmost substring of $ab'\beta$.

$\underline{ab'\beta}$ not be empty because this would mean that this computation passes A with the same tail cyd as for S but in this case the next operation to be performed is f . This is impossible, since the course of computation from A must be determined by the tail at this point.

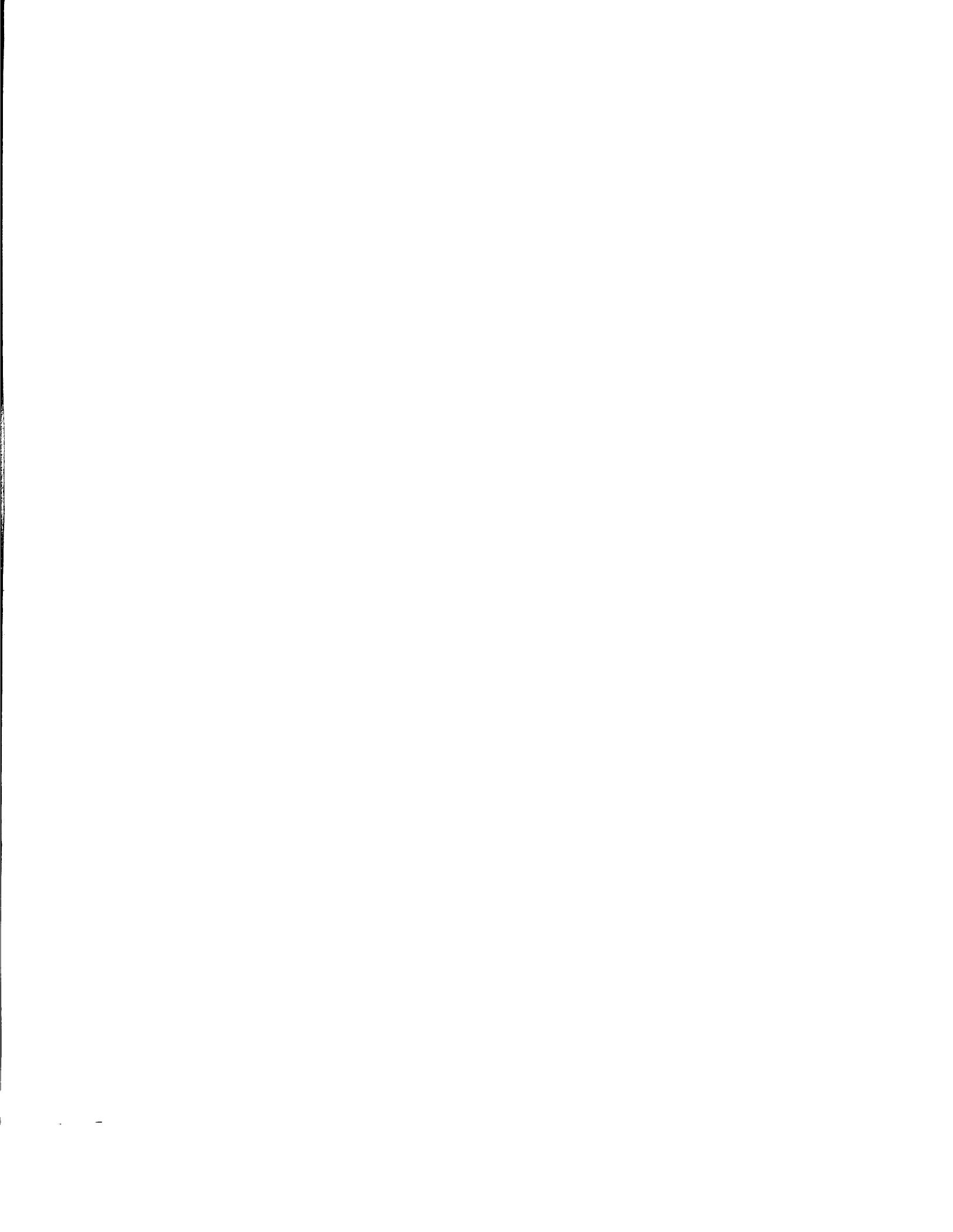
Hence, the computation must pass A with some tail $\underline{ab'\beta cyd}$ (or equivalently $\underline{ab'c'yd}$) where $\underline{ab'}$ is a rightmost substring of ab' .

(iii) For some non-empty string $b = b''\beta$ ($b'' \in \{\alpha, \beta\}^*$), the substring \underline{ab} is empty. In this case we take c' to be αc .

We proceed as in case (ii) with α and β interchanged and f and g interchanged.

Q.E.D.

^{*/} We could equally well take c' to be βc and consider computations starting with tail $ab'\beta cyd$.



Acknowledgment

We are indebted to David Cooper for stimulating discussions and mainly for his idea of using cut-set points which we have adopted in ALGORITHM II.

References

- C. BÖHM and G. JACOPINI [1966]
"Flow Diagrams, Turing Machines and Languages with only Two Formation Rules". CACM, Vol. 9, No. 5, pp. 366-371 (May 1966).
- J. BRUNO and K. STEIGLITZ [1970]
"The Expression of Algorithms by Charts", unpublished memo.
- D. C. COOPER [1967]
"Böhm and Jacopini's Reduction of Flow Charts". Letter to the Editor. CACM, vol. 10, No. 8, pp. 436-4 (August 1967).
- D. C. COOPER [1970]
"Programs for Mechanical Program Verification", in Machine Intelligence 6, Edinburgh University Press.
- E. DIJKSTRA [1968]
"GoTo Statement Considered Harmful", CACM, vol. 11, No. 3, pp. 147-148 (March 1968).
- E. ENGELER [1970]
"Structure and Meaning of Elementary Programs", in Symposium on the Semantics of Algorithmic Languages,
- D. E. KNUTH and R. W. FLOYD [1970]
"Notes on Avoiding 'GO TO' Statements", CS 148, Computer Science Department, Stanford University (January 1970).
- D. C. LUCKHAM, D. M. R. PARK and M. S. PATERSON [1970]
"On Formalized Computer Programs", Journal of Computer and System Sciences (June 1970).
- M. S. PATERSON and C. E. HEWITT [1970]
"Comparative Schematology", Unpublished memo.
- H. R. STRONG [1970]
"Translating Recursion Equations into Flowcharts", Journal of Computer and System Sciences (to appear).

