TRANSLATING PROGRAM SCHEMAS TO WHILE SCHEMAS

by

E.A. Ashcroft
University of Waterloo

and

Zohar Manna
Weizmann Institute, Rehovot, Israel

CS-73-22

TRANSLATING PROGRAM SCHEMAS TO WHILE SCHEMAS*

Edward Ashcroft
Computer Science
University of Waterloo
Waterloo, Canada

and

Zohar Manna
Applied Mathematics
Weizmann Institute
Rehovot, Israel

Abstract

While-schemas are defined as program schemas without go to statements, in which iteration is achieved using while statements. We present two translations of program schemas into equivalent while-schemas, the first one by adding extra variables, and the second one by adding extra logical variables. In both cases we aim to preserve as much of the structure of the original program schemas as possible.

We also show that in general any translation must add variables.

---

* An earlier version of part of this paper was presented at IFIP 1971 (Ljubliana, Yugoslavia). The research reported here was supported in part by the Advanced Research Projects Agency of the Office of the U.S. Secretary of Defence (SD-183).

INTRODUCTION

The program schema approach makes it meaningful to consider the relative 'power' of programming language constructs. Most work in this area [12, 4] has considered adding features to program schemas such as recursion and arrays. Here we consider removing, or at least restricting, a feature of program schemas: the go to statement.

There has been much interest lately, following observations by Dijkstra [7], in the possibility and desirability of removing go-to statements from programming languages, using instead such statements as the while statement. Programs in such languages should be better structured, easier to understand and, hopefully, easier to prove correct. The elegant formal system of Hoare [9] for proving programs correct requires programs with the sort of 'nested' structure that while statements provide. Goto-less programs are clearly an interesting class of programs to study.

We therefore define a class of while-schemas, and show that while-schemas are as powerful as program schemas by giving a translation, ALGORITHM I, of program schemas to equivalent while-schemas. This translation is interesting in that it preserves most of the 'loop structure' of the program schemas, and gives while-schemas of the same order of efficiency.

Bohm and Jacopini [3] have shown that program schemas can be translated into while-schemas, with the addition of extra

logical variables. A modification of a technique in Brown et al [2] would show (by a further translation) that the additional logical variables add no extra power to while-schemas. However, applying these two translations to a program schema would give a while-schema that would not resemble the original program schema at all. In any practical application, ALGORITHM I would be preferable.

Nevertheless, ALGORITHM I itself has some impractical features; the resulting while-schemas tend to be long, and, applied to a program schema that corresponds directly to a while-schema, it does not give us back that while-schema. We therefore present an improvement on Bohm and Jacopini's reduction to while-schemas with logical variables, which we call ALGORITHM II. This doesn't give us 'pure' while-schemas, but the schemas produced are often more 'readable' than those produced by ALGORITHM I. The method preserves whatever 'while-structure' already exists in a program schema, and when applied to a program schema corresponding directly to a while-schema, ALGORITHM II gives us back that while-schema.

Both ALGORITHM I and ALGORITHM II give while-schemas that use more variables in general than the original program schemas. It is natural to ask whether this is necessary feature of any translation. We show that this is the case by giving a program schema, with one variable, for which there is no equivalent one-variable while-schema. This also means, of course, that program schemas are more powerful than while schemas in the restricted sense that they need fewer variables in general.

PROGRAM SCHEMAS

A program schema consists of a finite sequence of statements, separated by semicolons. This sequence must start with a Start statement, e.g. START $(x_4, x_2, \cdots)$, designating input variables, and end with a Halt statement, e.g. HALT $(x_1, x_3, \cdots)$, designating output variables. The other statements may be of the following types (other types will be allowed in later types of schemas):

i) null statements i.e. null

ii) assignment statements

e.g. $x_1 \leftarrow f(x_2, x_1, x_3)$

iii) conditional statements

e.g. if ψ then S_1 else S_2

where S_1 and S_2 are statements

and ψ is a formula.

iv) compound statements

e.g. $[S_1; S_2; \cdots; S_n]$

where $S_1, S_2, \cdots, S_n$ are statements.

v) goto statements

e.g. goto L_i

where L_i is a label.

Any statement can be labelled, by preceding it with a label followed by a colon. A formula is any quantifier-free formula of predicate calculus.

The statement if ··· then ··· else null can be written if ··· then ··· provided no confusion results.

Example: The following is a program schema P_1, with one variable, that will be used often throughout the paper:

Schema P_1: START (x) ;
x ← a(x) ;
L : if p(x) then [x ← e(x) ; goto L] ;
A : if q(x) then x ← b(x) else [x ← g(x) ; goto N];
M : if r(x) then [x ← d(x) ; goto M] ;
B : if s(x) then [x ← c(x) ; goto L] else x ← f(x) ;
N : null ;
HALT (x)

A, B, L, M and N are labels (A and B will be used in later discussions). The symbols a, b, c, d, e, f and g denote functions and the symbols p, q, r and s denote predicates or tests. The expressions p(x), q(x) etc. are (simple) formulas).

WHILE-SCHEMAS

A while-schema is a program schema using only statements of types i), ii), iii) and iv), and type vi) below:

vi) while statement

e.g. while ψ do S

Such statements are to be considered as abbreviations for the equivalent statements

L : if ψ then [S; goto L] else null

Example: The following is a while-schema P_2, with two variables:

```
Schema P2: START (x) ;
           x ← a(x) ;
           while p(x) do x ← e(x) ;
           y ← x ;
           if q(x) then [x ← b(x) ; while r(x) do x ← d(x)] ;
           while q(y) ∧ s(x) do
                  [x ← c(x) ;
                   while p(x) do x ← e(x) ;
                   y ← x ;
                   if q(x) then [x ← b(x); while r(x) do x ← d(x)]];
           if q(y) then x ← f(x) else x ← g(x) ;
           HALT (x)
```

The schema uses the same symbols as P_1, denoting functions and predicates, and here we have the (more complicated) formula $q(y) \wedge s(x)$.

WHILE SCHEMAS WITH LOGICAL VARIABLES

A while-schema with logical variables is a program schema using only statements of types i), ii), iii), iv) and vi), and type vii) below:

vii) logical assignment statements

e.g. $t_i \leftarrow$ true

or $t_i \leftarrow$ false

The variables appearing in logical assignment statements are called logical variables, and they may not appear in ordinary assignments. They may appear, however, in formulas, as if they were propositions, i.e. 0-ary predicates.

Example: The following schema P_3 is a while-schema with one logical variable (and one 'ordinary' variable).

```
Schema P3 : START (x) ;
            x ← a(x)  ;
            t ← true  ;
            while t do
                [while p(x) do x ← e(x) ;
                 if q(x) then [x ← b(x) ;
                               while r(x) do x ← d(x) ;
                               if s(x) then x ← c(x)
                                       else [x ← f(x); t ← false]]
                         else [x ← g(x) ; t ← false]] ;
            HALT (x)
```

P_3 uses the same symbols as P_1 and P_2 denoting functions and predicates, and here the expression t is a formula.

EQUIVALENCE OF SCHEMAS

Two schemas, having the same numbers of input variables and of output variables, are said to be equivalent if they compute the same function (from input variable values to output variable values), no matter what functions or predicates are denoted by the symbols in the schema. (Of course, the same symbol appearing in the two schemas must denote the same function or predicate.)

More formally, we can first give meaning to the symbols in a schema by using an interpretation. An interpretation I consists of a domain D_I from which the variables in the schema may take values, and a specification of the functions and predicates over D_I denoted by the function and predicate symbols in the schema. The interpretation also supplies initial values (from D_I) for the input variables. Given an interpretation I, a schema S becomes a program (S, I). The program has a finite or infinite computation in the usual way, and if this is finite we let val (S, I) denote the final values of the output variables. If the computation is infinite, val (S, I) is undefined.

Two schemas S_1 and S_2 are then equivalent if, for all interpretations I, val (S_1, I) = val (S_2, I), i.e. both are undefined, or both are defined and have the same values.

In most of the paper we do not need the formal definition of equivalence. In these sections we will use simple equivalence-preserving transformations which are clearly correct. However we do

use interpretations in the last section.

Examples: The schemas P_1, P_2 and P_3 are all equivalent. (In fact P_2 is the result of applying ALGORITHM I to P_1 and P_3 is the result of applying ALGORITHM II to P_1.)

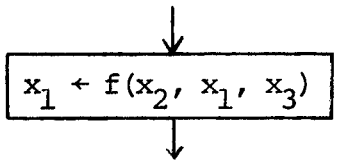
To see informally that P_1 is equivalent to P_2, note that each iteration of the main while statement in P_2 corresponds in P_1 to going from label B back to label B. The variable y in P_2, at the beginning of each iteration, holds the value that x previously held in P_1, the last time computation reached label A.

To see informally that P_1 is equivalent to P_3, note that each iteration of the main while statement in P_3 corresponds in P_1 to going from label L back to label L (the long way, via statement labelled B) or to label N. In the latter case, t is made false in P_3, and we subsequently exit from the main while statement.

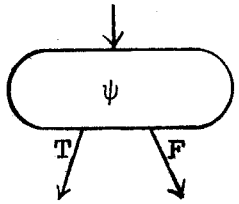
FLOWCHARTS

We will find it useful to consider the flowchart representations of schemas. Program schemas clearly correspond to arbitrary flowcharts, with one Start node and one Halt node, using the following types of statements:

i) assignments

e.g. 

ii) tests

e.g. 

where ψ is a formula.

We shall be more concerned with normal forms for such flowcharts.

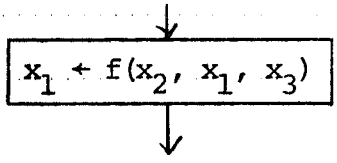
While-chart form

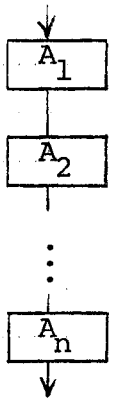
Firstly, it is clear that while-schemas have more restricted 'structure' than program schemas, and we define a correspondingly restricted class of flowcharts:

A while-chart is a one-entrance, one-exit piece of flowchart constructed inductively as follows:

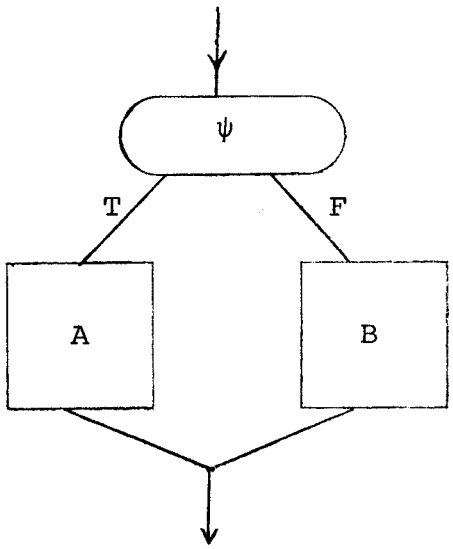
i) an empty edge is a while-chart i.e. 

ii) an assignment statement is a while-chart.

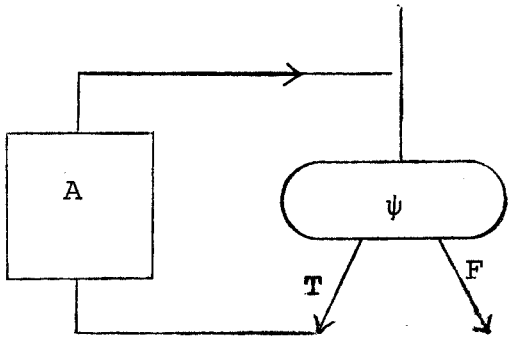
e.g. 

iii) 

is a while-chart, where $A_1, A_2, \cdots, A_n$ are while-charts.

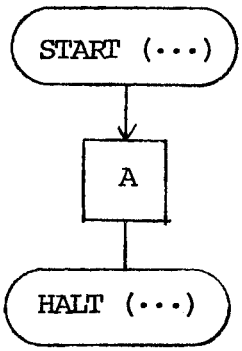
iv) 

is a while-chart, where A and B are while-charts and ψ is a formula.

v) 

is a while-chart, where A is a while-chart and ψ is a formula.

The various cases correspond to the types of statement allowed in while-schemas. Thus for any flowchart in while-chart form:



where A is a while-chart, there is an equivalent while-schema, and vice versa.

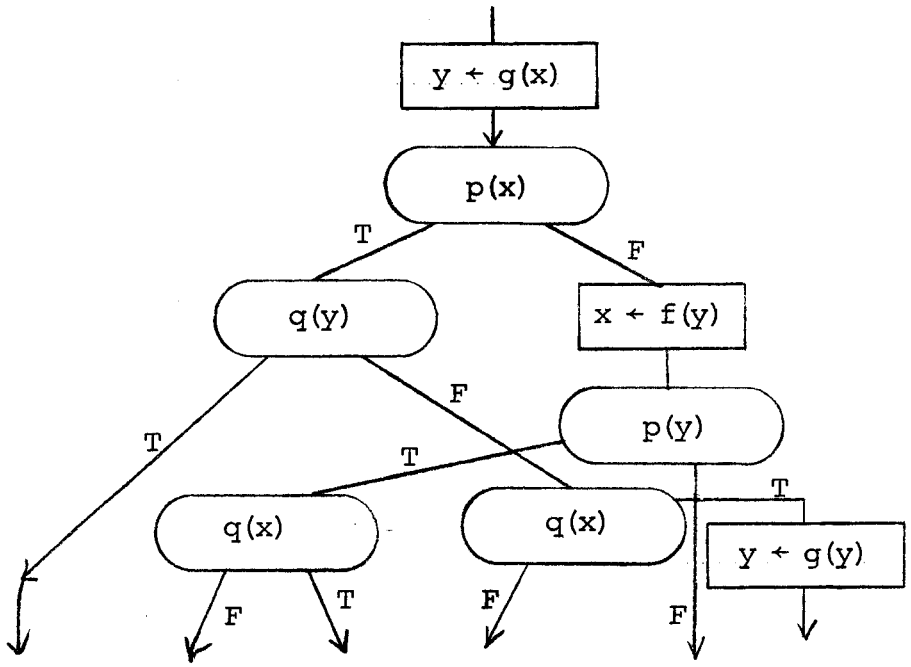
Block-form

Even general flowcharts can be put into normal forms, by such methods as duplicating nodes, unwinding loops etc. One such is the block-form of Cooper [6] and Engeler [8].

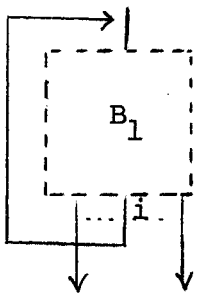
A block is a one-entrance, many-exit piece of flowchart constructed inductively as follows (we occasionally number the exits from a block, starting at the left):

i) A basic block is a block. A basic block is a one-entrance, many-exit tree-like piece of flowchart,

e.g.

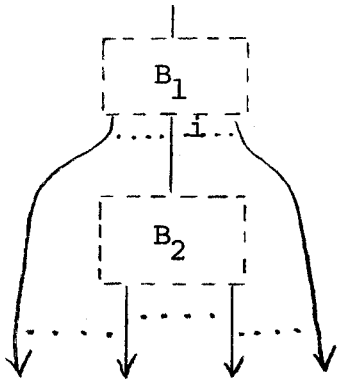


ii) (Looping on the i-th exit)



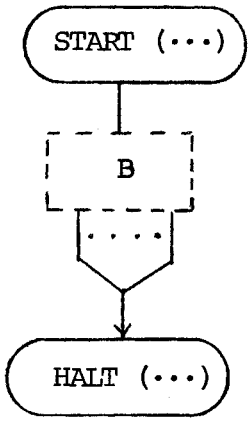
is a block where B_1 is a block.

iii) (Concatenating with the i-th exit)



is a block, where B_1 and B_2 are blocks.

A flowchart is in block form if it is of the form



where B is a block.

Clearly every flowchart in block form is equivalent to some program schema. The result of Engeler and Cooper is that for every program schema we can find an equivalent flowchart in block form.

Example: Figure 1 shows a flowchart P_1' in block form which is equivalent to the program schema P_1. The blocks are indicated by broken lines. B_0, B_1 and B_3 are basic blocks. B_2, B_4 and B_6 are constructed by looping, and B_5 and B_7 by concatenation.

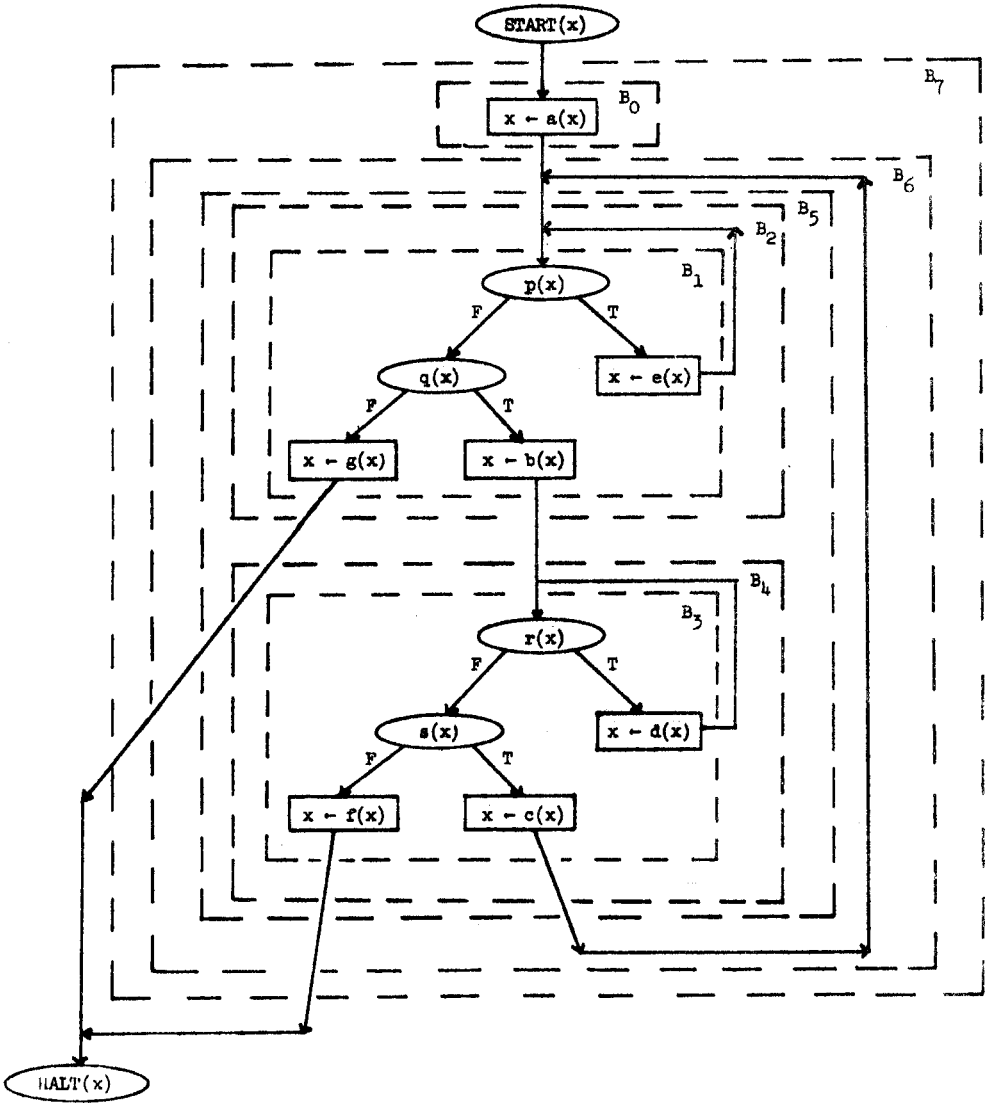
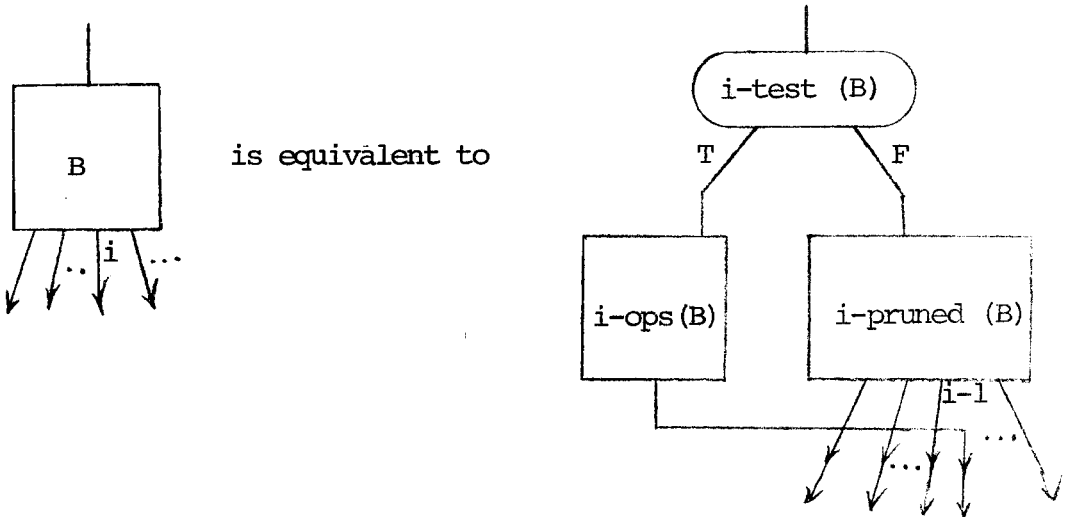


Figure 1. Block form flowchart P_1' .

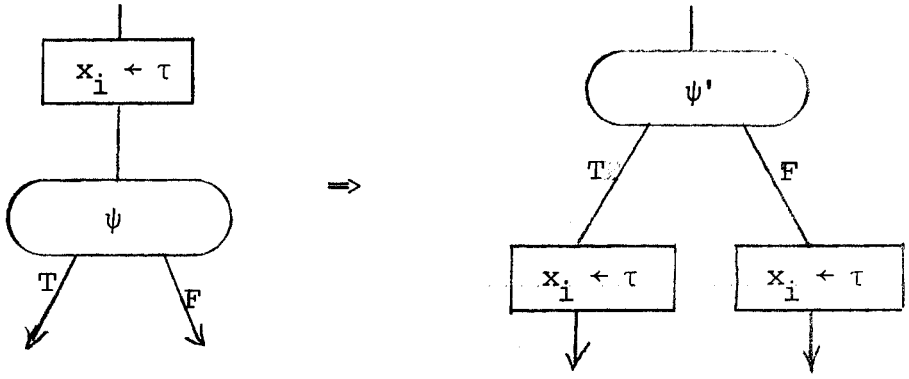
Properties of Basic Blocks

Before we consider our last normal form, we observe two useful properties of basic blocks.

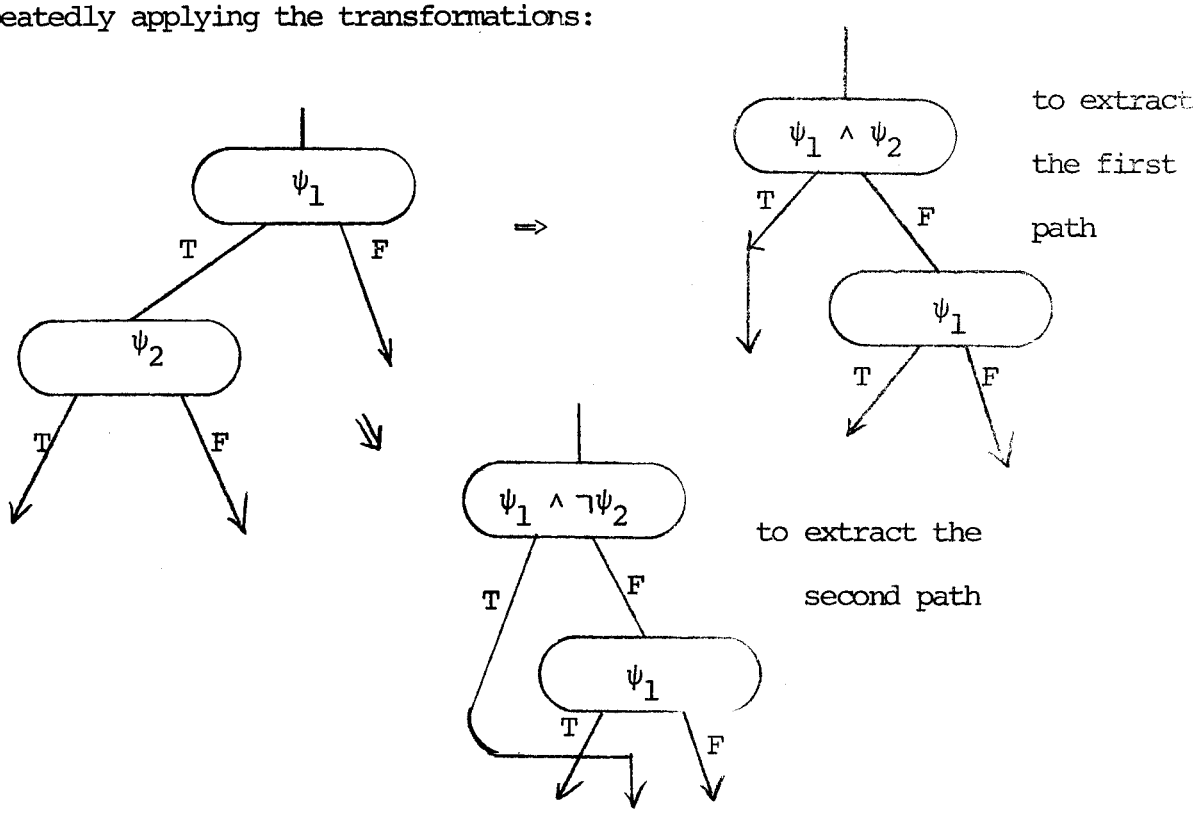
1. Given any basic block B and some i-th exit of B there exist a formula i-test (B) , a basic block i-pruned (B) and a sequence of assignment statements i-ops (B) such that



To see this, note first that the basic block can be put into a form in which the tests on the path to the i-th exit precede the assignments, by repeated application of the transformation:



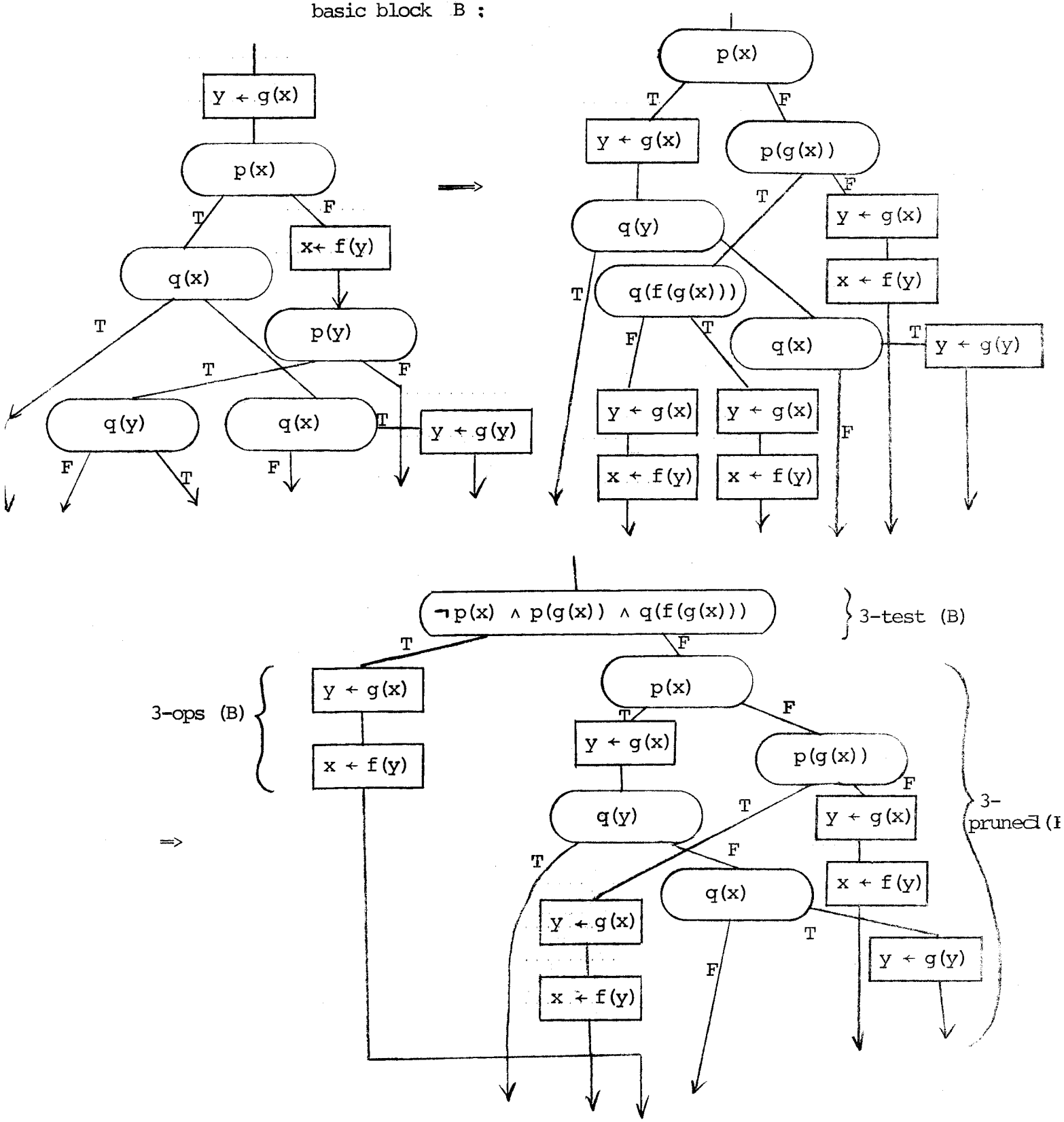
where ψ' is like ψ but with x_i replaced by τ . It is then a simple matter to find a single test to 'extract' the i-th path by repeatedly applying the transformations:

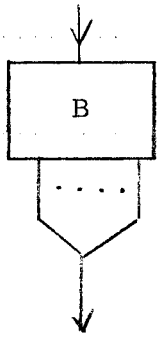


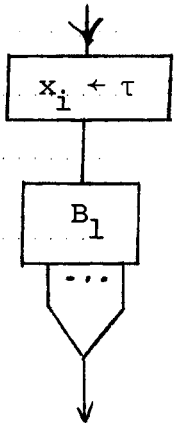
This eventually gives us the desired form for the basic block. It will be called the 'i-extracted form'.

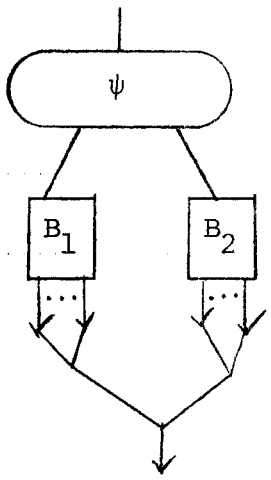
Example: obtaining the 3-extracted form of the following

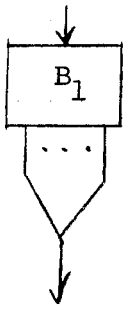
basic block B :

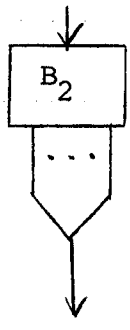


2. The second property of basic blocks that we need is that every piece of flowchart of the form 

, where B is a basic block, is a while-chart. This can be seen very easily by induction on the number of statements in B . If there are no statements, we have an empty edge, which is a while chart. If there are $n > 0$ statements, we have either 

or 

, where B_1 and B_2 are basic blocks. In both cases we have while charts, since 

and 

are while charts by the induction hypothesis.

Module form

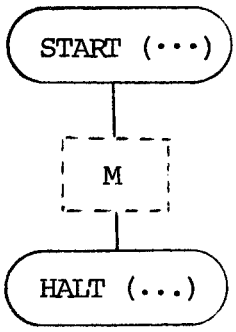
The final normal form for flowcharts which we will consider is module form.

A module is either

i) an assignment statement

or ii) a one-entrance,one-exit piece of flowchart constructed from tests and modules.

A flowchart is in module form if it is of the form



where M is a module.

This definition may appear surprising since we immediately have that any flowchart is in module form, by taking each assignment statement as a module, at one level, and then taking the whole flowchart as a module at the next level. However, we can find an interesting subclass of module-form flowcharts.

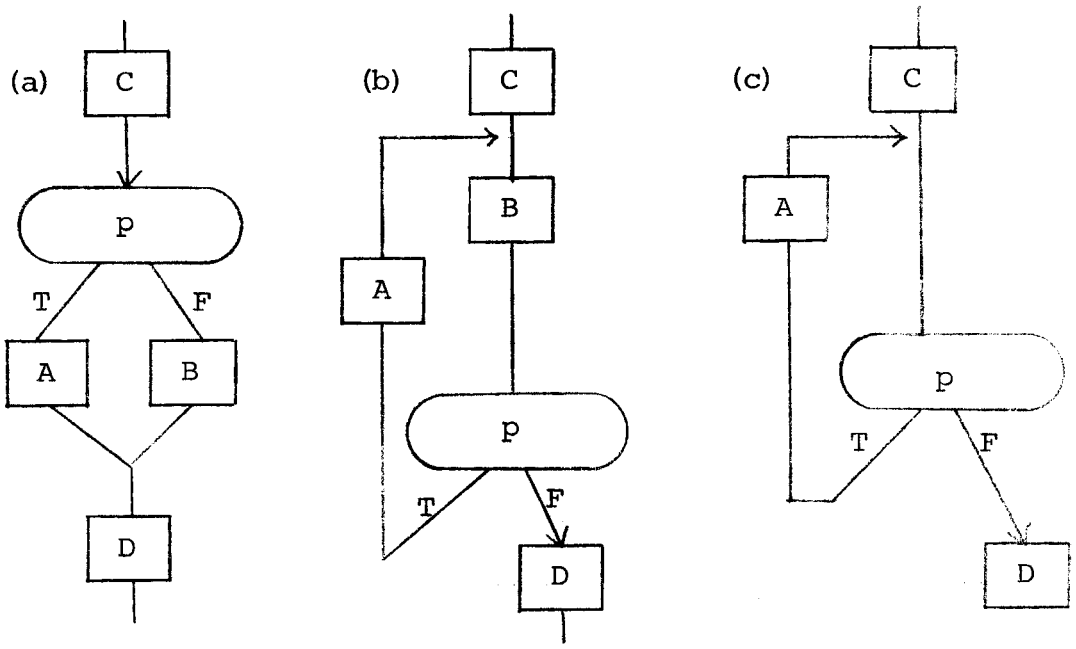
A simple module is either

i) an assignment statement

or ii) a one-entrance,one-exit piece of flowchart constructed from modules and at most one test.

A flowchart is in simple module-form if it is in module form, and each module is simple.

A simple module either has no tests, and is thus either an empty edge or the concatenation of modules, or it has one test and can only be of the following forms:



The analogy with while-schemas is obvious:

(a) is equivalent to [C ; if p then A else B ; D]

(b) is equivalent to [C ; B ; while p do [A; B]; D]

(c) is equivalent to [C ; while p do A ; D]

For every flowchart in simple module form there is an equivalent while-schema, and vice versa.

The motivation for module form now becomes clear. At one extreme we can take any flowchart as a module, whose sub-modules are simply assignment statements. However, if by ingenuity, and equivalence-preserving transformations, we can get many levels of modules, with fewer tests per module, then we get closer to simple module form, and hence closer to while-schemas.

Example: In Figure 2 we give a module form flowchart P_1'' for the program schema P_1. Modules M_1 and M_2 are simple, but module M_3 contains two tests $q(x)$ and $s(x)$.

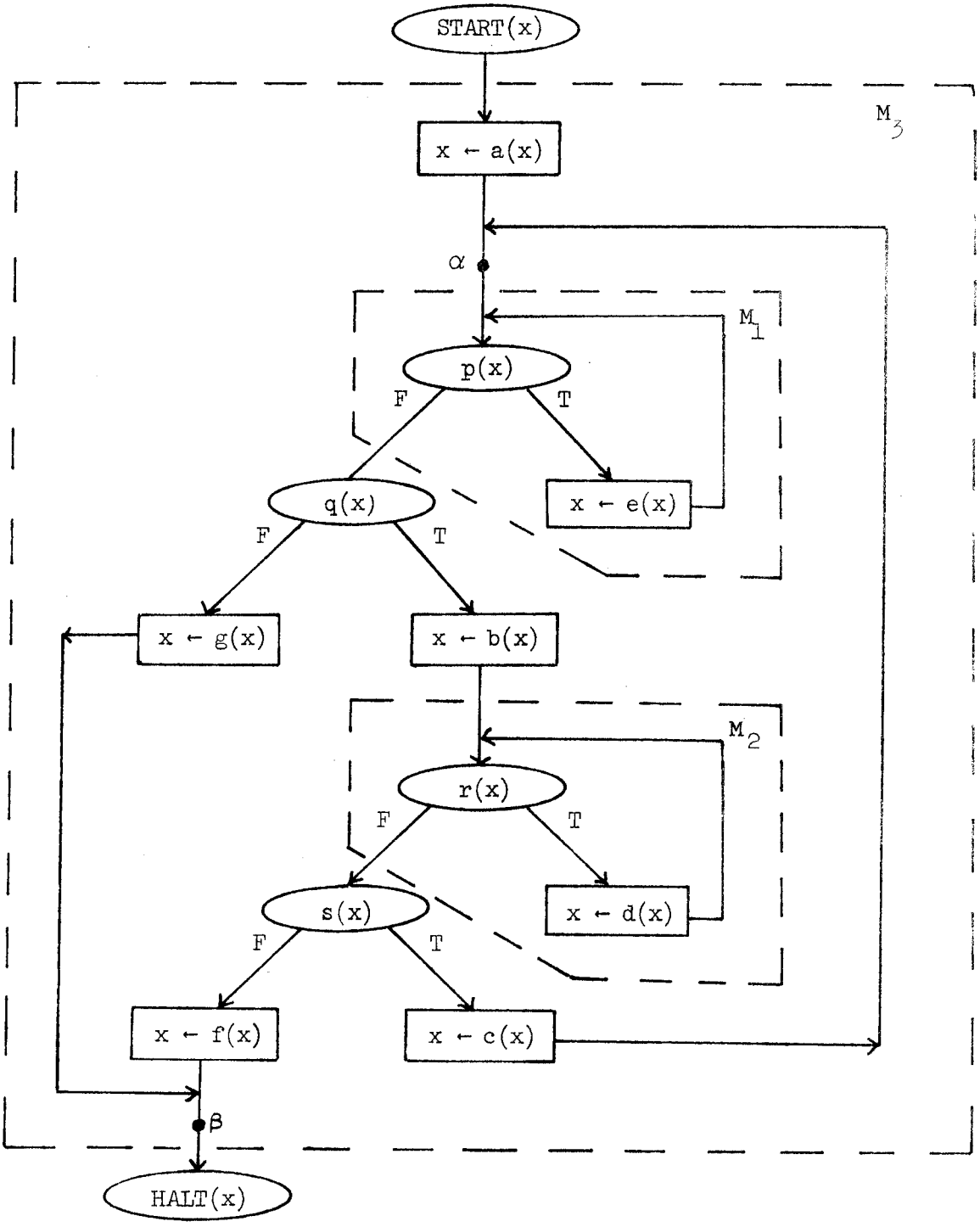
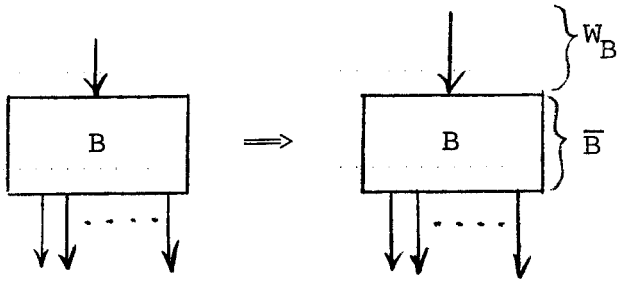


Figure 2. The module form flowchart P_1^m .

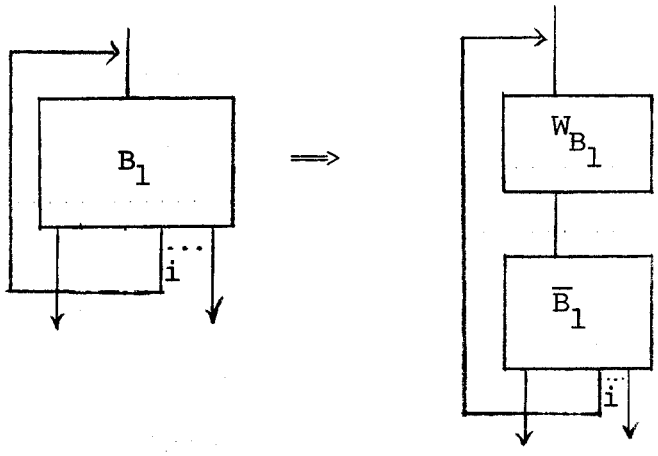
ALGORITHM I

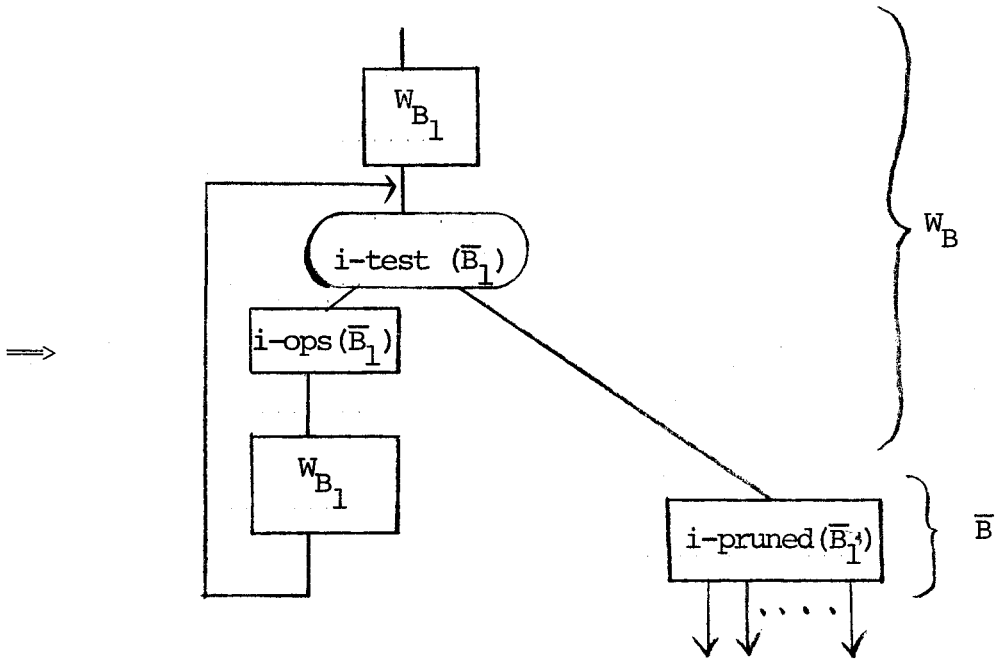
To translate program schemas to while schemas it suffices to consider flowcharts in block form. We show how to transform each block B into an equivalent piece of flowchart consisting of a while-chart W_B followed by a basic block $\overline{B}$. We do this by induction on the block structure as follows:

i) B is a basic block:

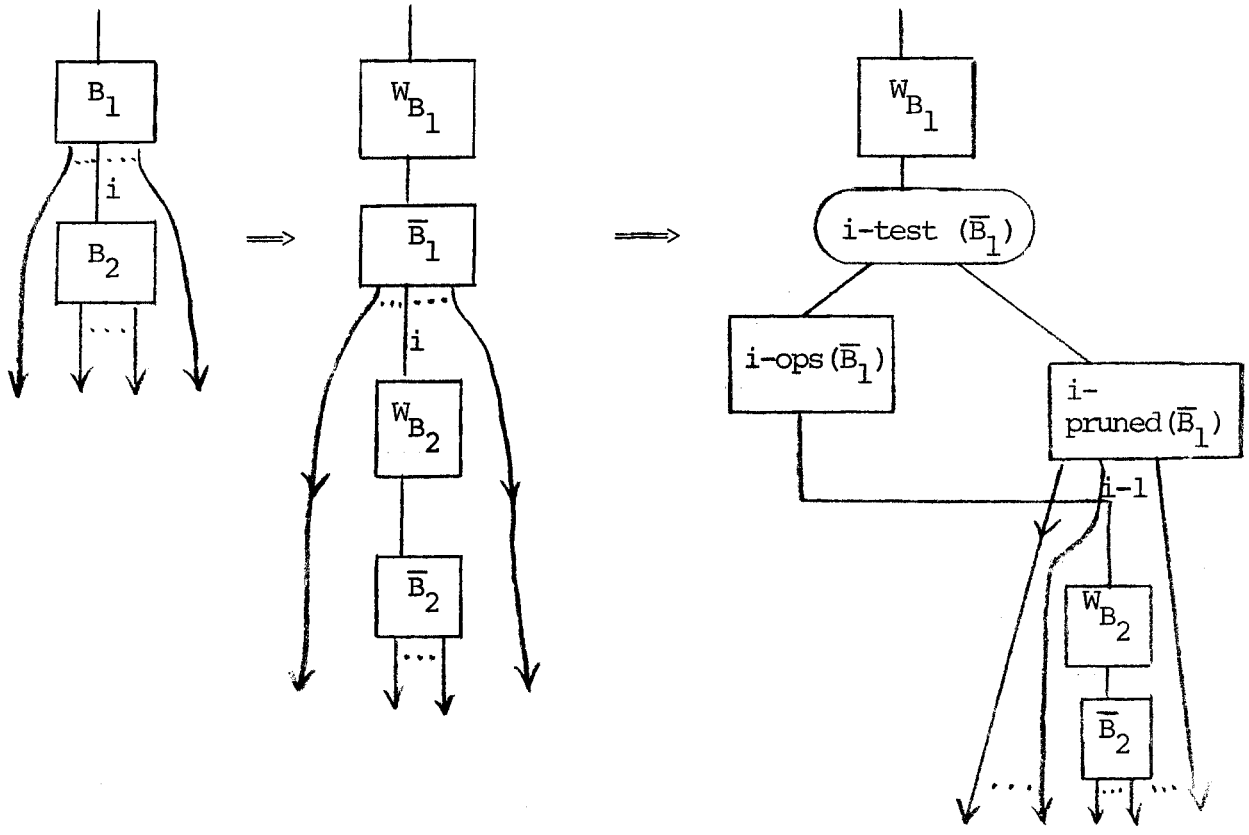


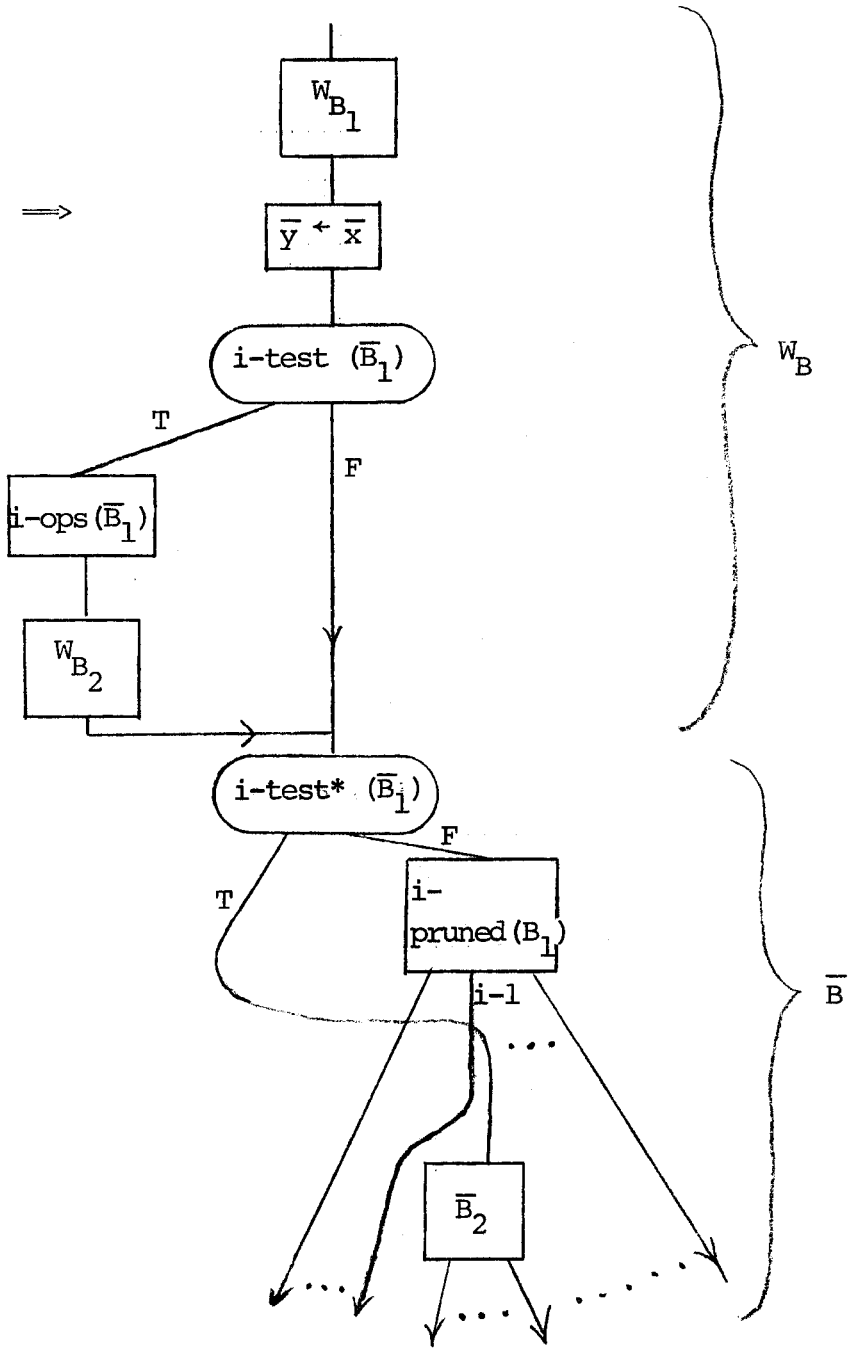
ii) B is constructed by looping on the i-th exit of B_1 :





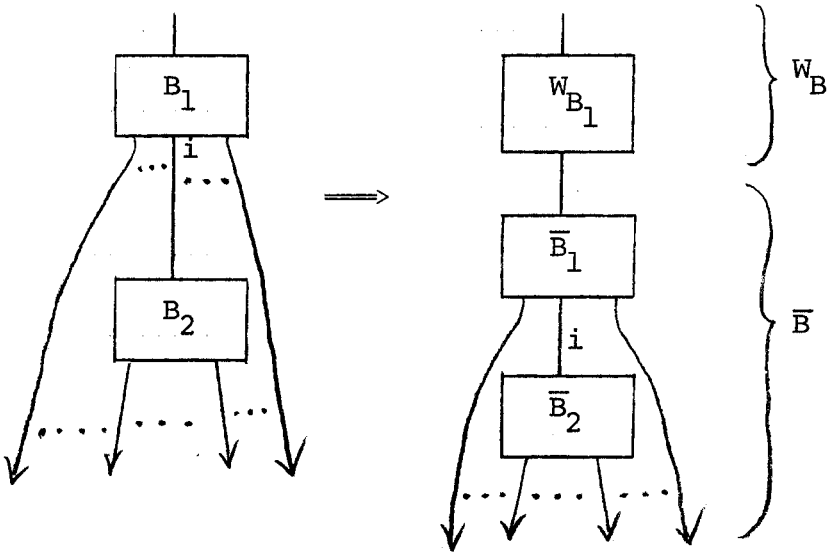
iii) B is the concatenation of B_1 and B_2, using the i-th exit of B_1 :





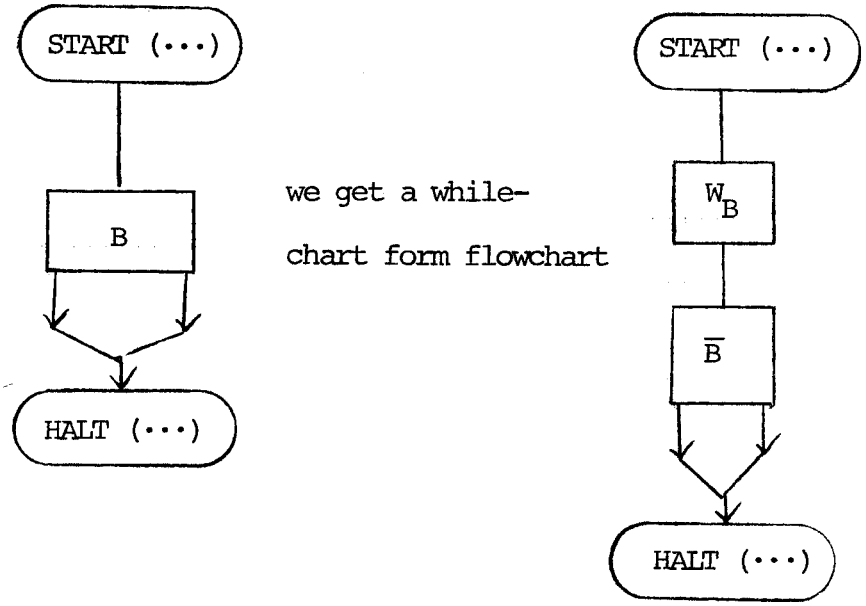
$\bar{x}$ is a vector of the variables occurring in i-test$(\bar{B}_1)$; $\bar{y}$ is a vector of the same length of <u>new</u> variables; i-test* $(\bar{B}_1)$ is the same as i-test$(\bar{B}_1)$ but with variables $\bar{x}$ replaced by $\bar{y}$, so that any computation must take the same branch out of each test.

Note: If B_2 is a basic block, we can simply make the transformation



No new variables are needed in this case.

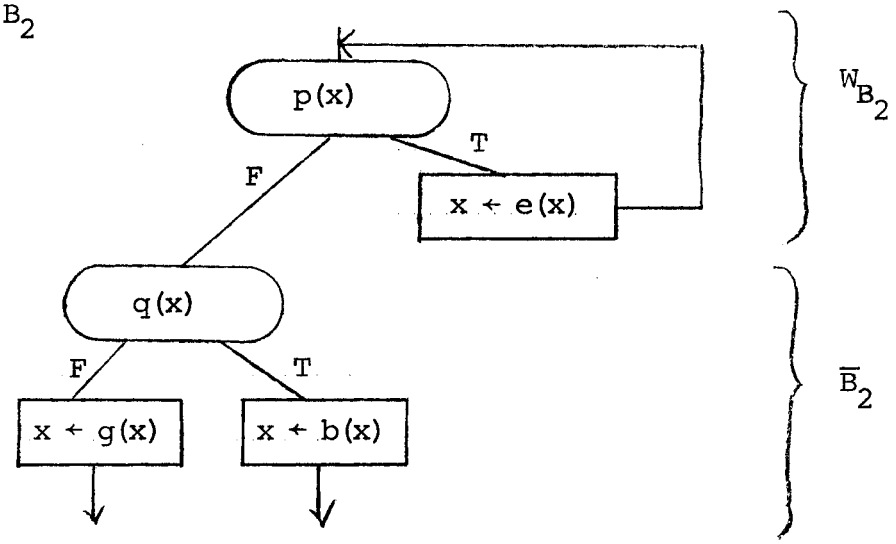
Thus for every block form flowchart

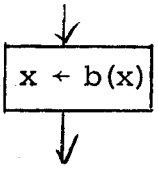
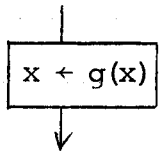


and hence a while schema.

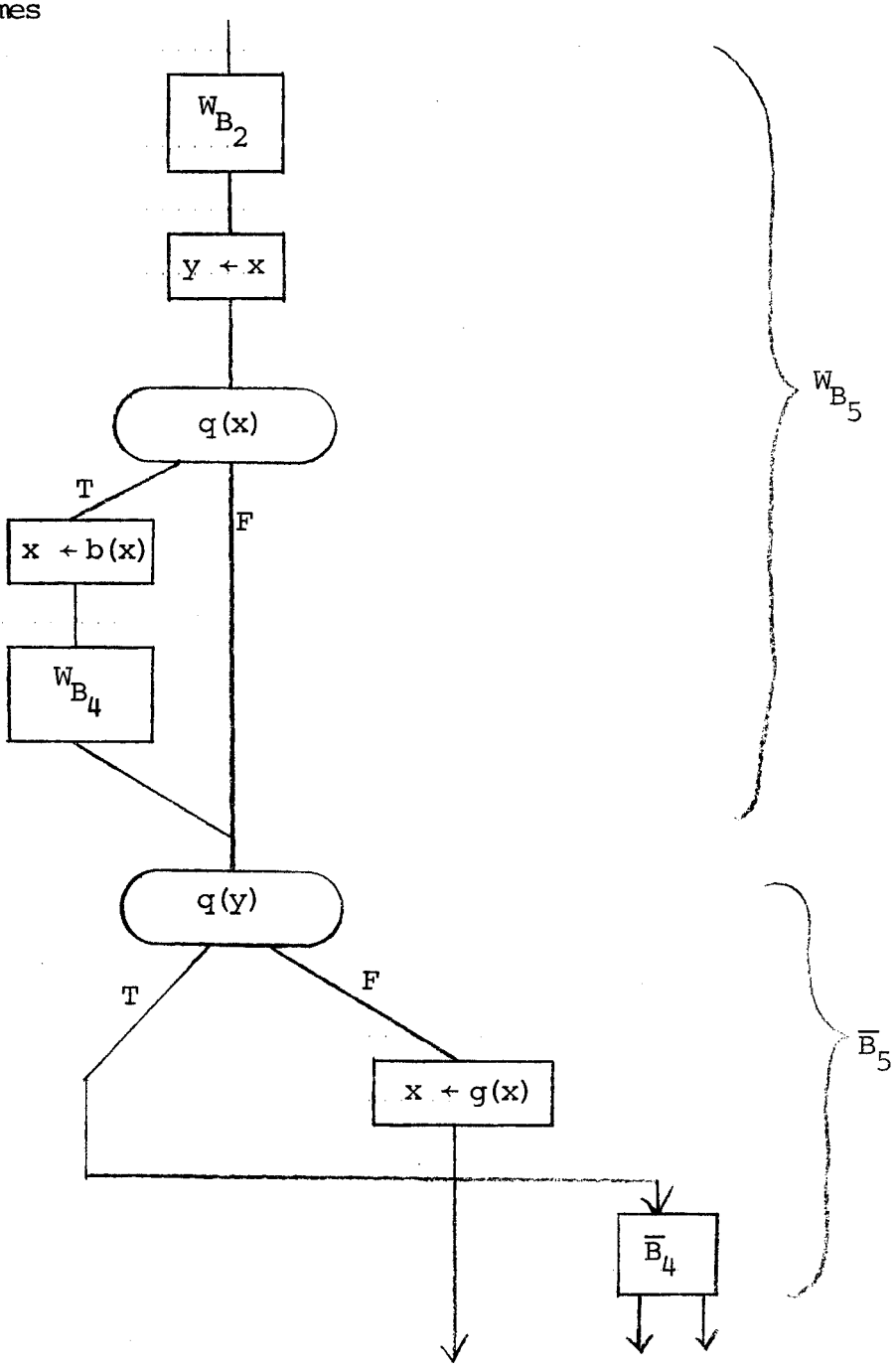
Example: We take flowchart P_1' of Figure 1. Blocks B_2 and B_4 are already of the required form:

e.g. B_2

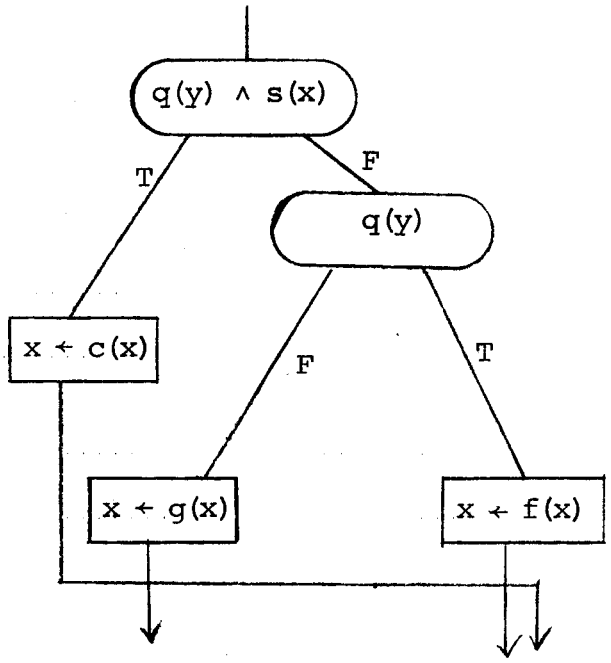


2-test($\overline{B}_2$) is q(x) , 2-ops($\overline{B}_2$) is  and 2-pruned($\overline{B}_2$) is simply 

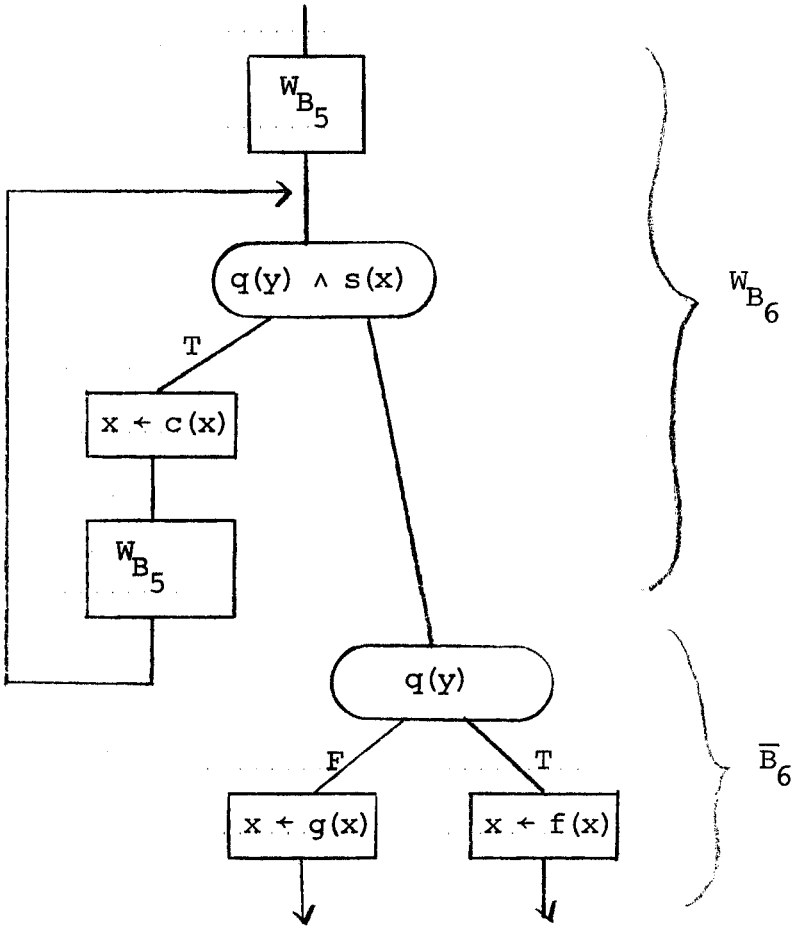
Hence B_5 becomes



The 3-extracted form of $\overline{B}_5$ is then



and thus B_6 becomes



The final while-chart form flowchart equivalent to P_1' is

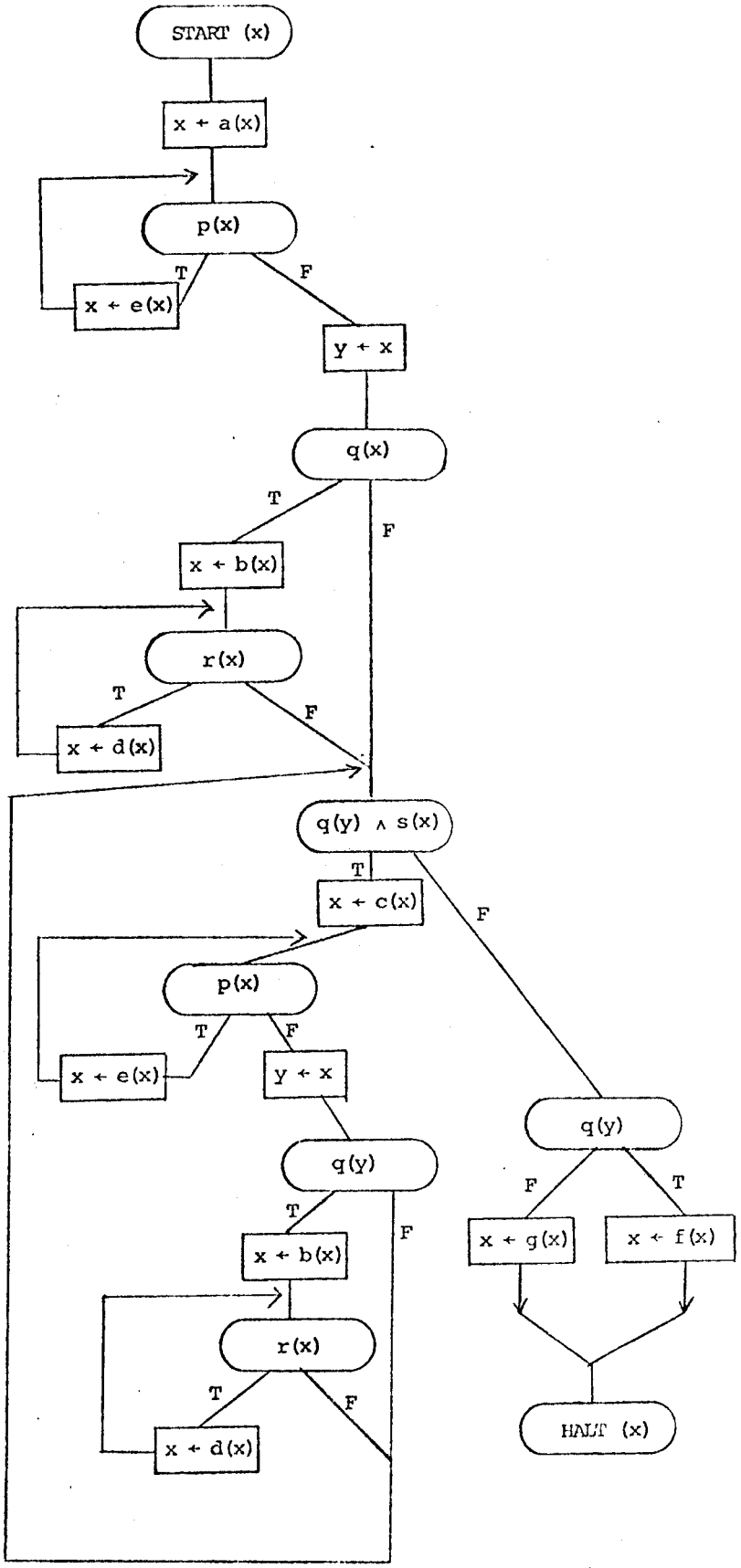


Figure 3: While-chart form flowchart equivalent to P_1' .

Comments:

i) Putting a flowchart into block form in general requires some increase in the size of the flowchart. This can be reduced by allowing the exits of any block to be joined together in arbitrary ways and still be a block. In the same spirit we would allow basic blocks that were not tree-like but merely loop-free. ALGORITHM I will work just as well for such block form. The only change needed is in defining the i-extracted form for basic blocks; for example i-ops (B) becomes a one-exit basic block rather than a sequence of assignment statements.
This modified block form corresponds to interval analysis (see [1]).

ii) To minimise the number of new variables added by ALGORITHM I, we must find block form flowcharts which avoid concatenating blocks except when the second block is basic. Even for while-schemas it is not clear how to do this, and so ALGORITHM I is not an identity mapping on while-schemas.

ALGORITHM II

The idea of Bohm and Jacopini's translation of program schemas to while-schemas with logical variables [3] (see also [5]) can be expressed as follows. Suppose the given program schema has n statements including the Halt statement, numbered, for our convenience, 1 to n. We construct a while-schema using k additional logical variables, where $2^n \leq k < 2^{n+1}$. Each statement of the original program schema then corresponds to a particular pattern of values for the k variables - the number of the statement written in binary notation. The while-schema consists of a single while statement, the formula of which will be true provided the 'pattern' of logical variable values does not correspond to the Halt statement. If the formula is true, we enter the body of the while statement, where a series of tests decides to which statement in the program schema the logical variable values correspond. The operation of that statement is then performed, and the values of the logical variables are changed so that their 'pattern' corresponds to the next statement to be executed in the program schema. The body of the while statement is repeatedly executed, until we reach the pattern for the Halt statement in the program schema. When this happens we exit from the while statement, and reach the Halt statement of the while-schema.

The while-statement simply acts as a one-loop interpreter, performing one operation of the original program schema on each iteration. The logical variables simply represent a 'program counter'.

An improvement upon this method, due to Cooper [private communication], reduces the number of logical variables required. We take the flowchart representation of the program schema and choose a 'cut-set' of the edges between the assignment and test statements from which it is composed, i.e. we choose one edge per loop. We add the edge leading to the Halt statement to this set. These edges are then numbered, and coded up as logical variable value patterns as before. The while statement 'interpreter', on each iteration, now performs the operations of the original program schema from one cut set edge to the next cut set edge, and updates the logical variables accordingly. This technique is used in ALGORITHM II, below.

We consider flowcharts in module form, and, for good results, we try to get as many simple modules as possible. We then translate each module M into a statement of while-schema W_M by induction on the module structure.

i) If M is an assignment statement, W_M is that assignment statement.

ii) If M is a simple module, W_M is the corresponding statement of while-schema (as indicated in the section FLOWCHARTS).

iii) If M is a non-simple module, then we apply Cooper's version of the Bohm and Jacopini reduction. We choose a cut set of the edges between modules and tests comprising the module M, and add the single exit edge of M. We then take sufficient 'new' logical variables to represent these positions in M, and construct a statement of while-schema. This statement

will be a compond statement $[S_1; S_2]$. Statement S_1 will perform the operations from the entrance of M up to the first cut-set edge, and set the logical variables to correspond to that edge. Statement S_2 is then the while statement which 'interprets' the module M . It's formula checks that the current pattern of logical variable values does not correspond to the exit edge. The body determines the current cut set edge, performs the operations to the next cut set edge (using the while-schema statements corresponding to the modules of which M is composed) and updates the logical variable values accordingly. This is possible as a statement of while-schema since the use of a cut set of edges ensures that there is a bound on the number of tests and modules that can be performed between one cut set edge and the next.

Example: The modules of flowchart P_1'' (Figure 2) correspond to statements of while schema as follows:

M_1 and M_2 are simple modules and correspond to

$$\underline{\text{while}}\ p(x)\ \underline{\text{do}}\ x \leftarrow e(x)$$

and $$\underline{\text{while}}\ r(x)\ \underline{\text{do}}\ x \leftarrow d(x)$$ respectively.

M_3 is non-simple, so we choose cut set edges, for example α and β in Figure 2 (there is only one loop in M_3 and we must add the exit edge of M_3). We then need one logical variable, t say, to keep track of the cut set edge - true corresponds to α , false

corresponds to β . W_{M_3} is then the following statement of while-schema

```
[[x ← a(x); t ← true];
  while  t  do  [W_M1;
                 if  q(x)  then  [x ← b(x);
                                  W_M2;
                                  if  s(x)  then  x ← c(x)
                                            else  [x ← f(x);
                                                   t ← false]]
                           else  [x ← g(x); t ← false]]]
```

Enclosing W_{M_3} between Start and Halt statements then gives us the while-schema P_3 .

Comment: No reasonable algorithm is known for finding the optimal equivalent module form for a program schema, optimal in the sense that ALGORITHM II adds the smallest number of logical variables. However, it is clear that the flowcharts of while-schemas are in simple module form, so that ALGORITHM II is the identity mapping on while-schemas.

THE NECESSITY OF ADDING VARIABLES

We show that any translation from program schemas to while-schemas must in general add variables. We prove that for a particular one-variable program schema there is no equivalent while-schema that also uses only one variable.

A similar result has been demonstrated by Knuth and Floyd [10] and Scott [private communication]. However, the notion of equivalence used by those authors is more restrictive than ours, in that it requires the equivalence of computation sequences (i.e. the sequences of assignments and tests in order of execution) and not just the equivalence of final results. For example, the following program schema has no 'equivalent' while-schema if we consider execution sequences:

```
START (x) ;
x ← a(x)  ;
L : if  p(x)  then  [x ← b(x); if  q(x)  then  [x ← c(x); go to  L]
                                          else  x ← d(x)]
              else  x ← e(x);
HALT (x)
```

However, if we apply ALGORITHM I, we get an equivalent while-schema, which happens to use only one variable:

```
START (x) ;
x ← a(x)  ;
while  p(x) ∧ q(b(x))  do  x ← c(b(x)) ;
if  p(x)  then  x ← d(b(x))  else  x ←  e(x);
HALT (x)
```

Clearly our result is stronger than the previous results, and needs a more complicated program schema to demonstrate it. The one we use is the following program schema P_5 :

```
Schema P5 : START (X) ;
            L : if  p(X)  then  [X ← e(X); go to  L];
                if  q(X)  then  X ← e(X)  else  [X ← e(X); go to  N];
            M : if  q(X)  then  [X ← d(X); go to  M];
                if  p(X)  then  [X ← d(X); go to  L]  else  X ← d(X);
            N : null;
            HALT (X)
```

This schema is similar to P_1 , but is simpler since it only uses two functions and two predicates. It is expecially interesting because for most other simpler versions of P_1 there are equivalent one-variable while schemas. For example, the program schema below

```
START (x);
x ← a(x) ;
L : if  p(x)  then  [x ← e(x); go to  L];
    if  q(x)  then  x ← b(x)  else  [x ← g(x); go to  N];
M : if  q(x)  then  [x ← b(x); go to  M];
    if  s(x)  then  [x ← c(x); go to  L]  else  x ← f(x);
N : null;
HALT (x)
```

is equivalent to the following one-variable while-schema

START (x) ;

$x \leftarrow a(x)$;

while $p(x)$ do $x \leftarrow e(x)$;

while $q(x) \wedge q(b(x))$ do $x \leftarrow b(x)$;

while $q(x) \wedge s(b(x))$ do

[$x \leftarrow c(b(x))$;

while $p(x)$ do $x \leftarrow e(x)$;

while $q(x) \wedge q(b(x))$ do $x \leftarrow b(x)$] ;

if $q(x)$ then $x \leftarrow f(b(x))$ else $x \leftarrow g(x)$;

HALT (x) .

Our proof that there is no one-variable while-schema P_5' equivalent to P_5 must therefore depend crucially on special features of P_5. The essential property of P_5 is the following:

> in any unfinished computation of P_5, if p is true and q is false, then the next-but-one function that will be applied is e, whereas if q is true and p is false then the next-but-one function that will be applied is d. If both p and q are false, the computation will terminate after applying one more function.

Let $D = \{d, e\}^*$. We shall consider the interpretations I_z, where $z \in D \cdot h \cdot D$, defined as follows:

i) $D_{I_z} = D$

ii) for $y \in D_{I_z}$, $d(y) = yd$

$e(y) = ye$

$p(y) = |y| < |z| \ \& \ z(|y| + 1) = e$ *)

$q(y) = |y| < |z| \ \& \ z(|y| + 1) = d$

iii) The initial value of the input variable X is Λ , the empty string. **)

Note that the predicates p and q are mutually exclusive, and, from the essential property of P_5 , the computation of (P_5, I_z) , where $z = uhv$ $(u, v \in D)$, must terminate with $\underline{val}\ (P_5, I_z) = eu$ (the symbol h makes both p and q false and makes the computation halt). Also for any interpretation $I_{w\alpha uhv}$ $(\alpha \in \{d, e\}, w,u,v \in D)$, when the value of X becomes ew , the future course of the computation is determined by uhv , since this substring will determine the possible future values of the predicates p and q . This property also holds for any <u>one-variable</u> schema P_5' equivalent to P_5 (it will be called the main property of P_5'), and will be used to show that such a schema cannot exist.

Let us assume therefore that there exists a one-variable while-schema P_5' equivalent to P_5 . Without loss of generality we can assume there is some while statement S in P_5' , say $\underline{while}\ \psi'\ \underline{do}\ S_1$, which is not contained in or followed by any other while statement, and for which S_1 is executed in the computation for some I_z . We can also assume that there is no bound on the number of

---

*) $|y|$ denotes the length of string y ; $z(i)$ denotes the i-th symbol in string z .

**) This is a 'Herbrand' or 'free' interpretation (see [11]).

iterations of S for computations for such interpretations I_z. (All such bounded while statements could be 'unwound' the corresponding number of times, leaving only 'unbounded' while statements and while statements never entered for any I_z.)

Let the maximum 'depth' of functional composition in any formula in P_5' be M. Then in computation of (P_5', I_z), if we evaluate a formula ψ for value w of variable x then the outcome of ψ is determined by $z(|w| + 1), z(|w| + 2), \cdots, z(|w| + M + 2)$. We define <u>visible</u> (z, w) as this substring of z starting at $z(|w| + 1)$ and ending at $z(|w| + M + 2)$.

<u>Lemma:</u> For all $n \geq 0$ there exist strings $u,w,y \in D$, with $|w| = n$, such that for all $v \in D$ the computation of (P_5', I_{uvwhy}) exits from S with a proper prefix of euv as the value of variable x.

This technical Lemma has the following informal corollary:

<u>Corollary:</u> For every $n \geq 0$ there exists a computation of P_5' which exits from S with more than n functions still to be applied.

This corollary contradicts the fact that S is not followed by or contained in another while-statement; the number of functions that can be applied after exiting from S is bounded. Hence while-schema P_5' cannot exist.

Proof of Lemma: By induction on n.

Base step ($n = 0$) Since S is unbounded there exists an interpretation I_z, whose computation enters S_1 before the end of the computation is 'visible', i.e. more than M function applications from the end. In other words $z' = u'\alpha y v' h y'$, where $u', y, v', y' \in D$, $\alpha \in \{d, e\}$ and $|y| = M + 1$, and the computation of $(P_5', I_{z'})$ reaches S with eu' as the value of X. Moreover, since S_1 is entered, the formula ψ' must be true for this value of X. Note that the truth of ψ' is determined by visible $(z', eu) = y$.

Consider now the interpretation $I_z = I_{uvhy}$, where $u = u'\alpha y$ and v is any string from D. The computation must reach S as for $I_{z'}$, i.e. with value eu' for variable X, since the changes in the interpretation are not 'visible' by this point. However when it subsequently exits from S it can not do so with value euv (the final value) for X, since visible $(z, euv) = y$, and for this value the formula ψ' must be true. Thus it must exit from S with a proper prefix of euv as the value of X.

Induction step Assume we have strings $u, w, y \in D$, with $|w| = n$, such that for all $v \in D$, the computation of (P_5', I_{uvwhy}) exits from S with a proper prefix of euv as the value of X.

We shall find a string $w' \in D$, $|w'| = n + 1$, such that for all $v' \in D$, the computation of $(P_5', I_{uv'w'hy})$ exits from S with a proper prefix of euv' as the value of X.

There are three cases to consider (in order):

i) for all $v = v'e$ (for all $v' \in D$) in the induction hypothesis, the corresponding proper prefix of euv is also a proper prefix of euv'. In this case we take $w' = ew$.

ii) for all $v = v'd$ (for all $v' \in D$) in the induction hypothesis, the corresponding proper prefix of euv is also a proper prefix of euv'. In this case we take $w' = dw$.

iii) for some $v = v''e$ in the induction hypothesis, the corresponding proper prefix of euv is euv'', i.e. the computation C of $(P_5', I_{uv''ewhy})$ exits from S with value euv'' for variable X. Note that the rest of the computation adds ew to the value of X.

Consider now the interpretations $I_{uv'dwhy}$, for all $v' \in D$. By the induction hypothesis, the value of X on exiting from S must in each case be a proper prefix of $euv'd$. But the main property of P_5' ensures that in no case can this value of X be euv', otherwise the future course of this computation, being determined by why, would be the same as for C, giving X a final value of $euv'ew$ instead of $euv'dw$.

Thus with $w' = dw$, the computations of $(P_5', I_{uv'w'hy})$ (for all $v' \in D$) exit from S with a proper prefix of euv' as the value of X.

□

ACKNOWLEDGEMENTS

We are indebted to David Cooper for stimulating discussions and for his modification of the Bohm and Jacopini's reduction, which we have used in ALGORITHM II. We are also grateful to Donald Knuth for his critical reading of an earlier version of this paper, and subsequent helpful suggestions.

REFERENCES

[1] Allen F. E. "A Basis for Program Optimization". Proc. IFIP Congress, Ljubliana, Yugoslavia, 1971.

[2] Brown S., Gries D. and Szymanski T. "Program Schemes with Push-down Stores". SIAM J. Comput.,1, No. 3, 1972.

[3] Bohm C. and Jacopini G. "Flow Diagrams, Turning Machines and Languages with only Two Formation Rules". Comm. ACM, 9, No. 5, 1966.

[4] Constable R.L. and Gries D. "On Classes of Program Schemata". SIAM J. Comput., 1, 1972.

[5] Cooper D.C. "Bohm and Jacopinis Reduction of Flow Charts". Letter to the Editor. Comm. ACM, 10, No. 8, 1967.

[6] Cooper D.C. "Programs for Mechanical Program Verification". Machine Intelligence 6, Edinburgh University Press, 1970.

[7] Dijkstra E. "Goto Statement Considered Harmful". Comm. ACM, 11, No. 3, 1968.

[8] Engeler E. "Structure and Meaning of Elementary Programs". In Symposium on Semantics of Algorithmic Languages, Springer-Verlag, 1971.

[9] Hoare C.A.R. "An Axiomatic Approach to Computer Programming". Comm. ACM, 12, No. 10, 1969.

[10] Knuth D.E. and Floyd R.W. "Notes on Avoiding Goto Statements". Information Processing Letters, 1, North-Holland Publishing Co., 1971.

[11] Luckham D., Park D. and Paterson M. "On Formalized Computer Programs". J. Comput. and Sys. Sci., 4, No. 3, 1970.

[12] Paterson M. and Hewitt C. "Comparative Schematology". Conference Record of Project MAC Conference on Current Systems and Parallel Computation, ACM, New York, 1970.