# STRUCTURABILITY OF PROGRAMS

A thesis submitted

by

Isabel Yea-Ping Chang

to

the School of Graduate Studies
of the University of Ottawa

in partial fulfillment of the requirements
for the degree of
Master of Science
in the subject of
Mathematics

© Isabel Yea-Ping Chang, Ottawa, Canada, 1975.

## ACKNOWLEDGEMENT

Sincere thanks are to be offered to Dr. E. S. Bainbridge for his advice and guidance. Thank with gratitude also for the encouragement I received while the thesis was in process.

CONTENTS

Page

ABSTRACT

CHAPTER I. Introduction ........................................ 1

CHAPTER II. Model of Computer and WHILE Programs.

§ 2.1 The Model of the Memory Unit ........................ 4

§ 2.2 The Model of the Control Unit ........................ 6

§ 2.3 WHILE Programs ........................................ 11

CHAPTER III. Program Equivalence.

§ 3.1 Automata and Machines ................................ 15

§ 3.2 Automata and Machines of Programs .............. 16

§ 3.3 Equivalence of Programs ............................ 18

§ 3.4 The Böhm and Jacopini Construction .............. 22

CHAPTER IV. Strong Equivalence to WHILE Programs.

§ 4.1 Kleene's Theorem ...................................... 27

§ 4.2 Strong Equivalence between Programs and WHILE Programs ........................................ 31

§ 4.3 Algorithm for Strong Equivalence to a WHILE Program ........................................ 34

CHAPTER V. Generalized WHILE Programs with REPEAT-EXIT.

§ 5.1 Examples to Illustrate the Generalized WHILE Algorithm ........................................ 43

§ 5.2 Algorithm for Translating Programs into Generalized WHILE Form with REPEAT-EXIT ........................ 57

BIBLIOGRAPHY ........................................ 67

## ABSTRACT

The purpose of the present work is to study the various techniques in translating computer programs into GO TO-less or structured programs. By strong equivalence between two programs we mean that they have the same execution sequences. Our first algorithm given in Chapter 4, tests whether programs are strongly equivalent to a WHILE program, a kind of structured program defined in § 2.3, and gives the equivalent WHILE program if the answer is 'yes'. The second algorithm translates the computer programs into generalized WHILE programs with REPEAT-EXIT. It is shown that the second algorithm is applicable to all computer programs.

## CHAPTER I

## INTRODUCTION

Structured programming theory is understood to deal with converting arbitrarily large and complex flowcharts into standard forms so that they can be represented by iterating and nesting a small number of basic and standard control logic structures which are, usually, sequencing, alternation and iteration. A structured program is characterized by the absence of GO TO's. It has been pointed out by authors of many articles [4, 10, 14] that a GO TO-free program tends to be not only easier to understand and allows better optimization by the compiler, but also better suited for an eventual proof of correctness. Our present work is to study about the various techniques, (e.g. [1, 2]), involved in translating computer programs into GO TO-less programs.

In this paper, we concentrate on the schema-type of computer programs, as it often turns out that properties of a given program can be proved independent of the exact meaning of its functions and predicates, and only the control structure of the program is really important in this case.

In Chapter 2, we give the definition of a memory unit of a machine and express, as an example, the Turing machine with 0-1 alphabet in the form of a model of the memory unit of the

machine. The control unit of a computer is specified by the definitions of program schemata and programs for a specific memory unit. An equation- form is introduced to describe programs. The definition of a GO TO-less or WHILE program is given followed by an example of a previous program expressed, then, in the WHILE form.

Chapter 3 is concerned with program equivalence. The definitions of automata and machines and automata and machines of programs are given. Then program equivalence of two kinds are defined involving the behaviors of automata and machines of programs. A procedure due to Böhm and Jacopini [ 2 ], for translating any program to a WHILE program for a different memory unit is given in a simplified form. The triviality of the program equivalence in this sense is obvious as the procedure is applicable to any program.

We discuss in Chapter 4, the strong equivalence of programs to WHILE programs. The algorithm which resembles Kleene's Theorem for the strong equivalence is given. A characterization theorem for strong equivalence to a WHILE program is proved with several interesting corollaries.

In Chapter 5, we extend the idea of WHILE programs to one with REPEAT and EXIT in which any program can be related to one in the sense of strong equivalence. The algorithm is given with examples. A theorem is proved showing that the algorithm works for all programs.

The results and the algorithms given in Chapter 4 and 5 for the strong equivalence of programs to WHILE programs and to the generalized WHILE programs with REPEAT-EXIT were developed from the outlines suggested by E. S. Bainbridge, and have not appeared elsewhere.

# CHAPTER II

## MODEL OF COMPUTER AND WHILE PROGRAMS

We will deal with an abstract description of computers so that the type of machines and languages involved are immaterial for the study of the correctness of programs. For a machine of the simplest form, a memory unit and a control unit are the two main components giving the essence of the machine. An abstract description of these two units is to be given in the first two sections.

### § 2.1 The Model of the Memory Unit.

Definition 2.1. A memory unit, $\Delta = (M, A, P)$ is defined to consist of the following entities:

(i) the set of memory states, denoted by $M$;

(ii) the set of operations carried out by the control unit on the memory states, denoted by $A \subset M^M$, where $M^M$ is the set of functions from $M$ to $M$;

(iii) the set of tests which the control unit can perform on the memory states, denoted by $P \subset 2^M$, where $2^M$ is the set of functions from $M$ to the set $2 = \{0, 1\}$.

Note. For notational convenience in the following, we shall assume that $P$ is closed under negation; that is, for each $p \in P$, there is a $\overline{p} \in P$ such that

$$\overline{p}\,(m) = \begin{cases} 0, & \text{if } p(m) = 1 \\ 1, & \text{if } p(m) = 0 \end{cases}$$

Then $\overline{\overline{p}} = p$. In fact, it is not unreasonable to suppose that P is a Boolean algebra, but we shall not use the other Boolean operations.

For example, the tape of a Turing machine can be described by a model of the above form. For simplicity, let us assume that the tape alphabet is { 0, 1 }. Then the memory states are the tape configurations; that is, the content of the tape plus an indication of the square being scanned by the machine head. The operations are the move and the print operations. There is a single test; namely; whether the scanned square is '1'.

For a formal presentation of this example, we need the following:

Definition 2.2. For any finite set X, we denote by $X^*$, the set of all finite sequences of elements of X, including the empty sequence $\Lambda$, with length 0.

$X^*$ is a monoid under concatenation '.', with identity $\Lambda$. That is, for finite sequence $\alpha$, $\beta$ in $X^*$, we have

$$\alpha \, . \, \beta = \alpha\beta$$

and $\alpha \, . \, \Lambda = \alpha = \Lambda \, . \, \alpha$.

We adopt the symbol $( \xi, x, \eta )$ for a tape configuration where $\xi$ and $\eta$ are elements of $X^*$, and $x \in X$ with X as the tape alphabet of the Turing machine (where in this case, X = { 0,1 }); and x is the scanned symbol. Let the squares to the left of $\xi$

and to the right of $\eta$ be filled with 0's. Formally, for a Turing machine over X, the model $\Delta = (M, A, P)$ is of the following form:

$M = \{(\xi, x, \eta) \mid \xi, \eta \in X^* \text{ and } x \in X\}$;

$A = \{R, L, \underline{0}, \underline{1}\} \subset M^M$;

$P = \{p_1, \bar{p}_1\} \subseteq 2^M$;

where 'R' is the movement of the scanner to the right,

i.e. $$R((\xi, x, \eta)) = \begin{cases} (\xi x, y, \eta'), & \text{if } \eta = y\eta' \\ (\xi x, 0, \Lambda), & \text{if } \eta = \Lambda \end{cases}$$

'L' is the movement of the scanner to the left,

i.e. $$L((\xi, x, \eta)) = \begin{cases} (\xi', y, x\eta), & \text{if } \xi = \xi' y \\ (\Lambda, 0, x\eta), & \text{if } \xi = \Lambda \end{cases}$$

'$\underline{0}$' is the printing of '0' at the scanned position,

i.e. $\underline{0}((\xi, x, \eta)) = (\xi, 0, \eta)$

'$\underline{1}$' is the printing of '1' at the scanned position,

i.e. $\underline{1}((\xi, x, \eta)) = (\xi, 1, \eta)$

and $p_1 \colon M \to 2$, such that

$$p_1((\xi, x, \eta)) = \begin{cases} 1, & \text{if } x = 1 \\ 0, & \text{if } x = 0 \end{cases}$$

$$\bar{p}_1((\xi, x, \eta)) = \begin{cases} 1, & \text{if } x = 0 \\ 0, & \text{if } x = 1. \end{cases}$$

## § 2.2 The Model of the Control Unit.

The control unit for a computer with memory unit $\Delta = (M, A, P)$ is specified by a program in the following simple

language.

Definition 2.3. Let A, P be finite sets. Then an (A, P) program schema $\pi = (Q, q_0, h, S)$ is defined to consist of the following entities:

$Q$, the set of statement labels;

$q_0 \in Q$, the entry label;

$h \notin Q$, the exit label; (define $\bar{Q} = Q \cup \{ h \}$)

and a function S assigning to each $q \in Q$, a statement which is one of the following:

(i) DO a GO TO r, where $a \in A$, $r \neq q$ and $r \in \bar{Q}$;

(ii) IF p THEN GO TO r, ELSE GO TO s, where $r, s \in \bar{Q}$, $r, s \neq q$, and $r \neq s$.

Notation. For convenience, we adopt the following notation which corresponds to the above two types of statements.

(i) $ar$

(ii) $pr + \bar{p}s$

Definition 2.4. Let $\Delta = (M, A, P)$ be a memory unit. Then an (A, P) program schema is called a program for $\Delta$. Every program has an underlying program schema, obtained by ignoring the structure of A and P.

Remarks. (1) Without loss of generality, we may assume that distinct programs have distinct sets $\bar{Q}$, of statements labels.

(2) As a convention, S will always be considered as a set of ordered pairs, expressed as

either $q = ar$, or $q = pr + \bar{p}s$,

with $r, s \neq q$ and at least one of $r, s \in Q$.

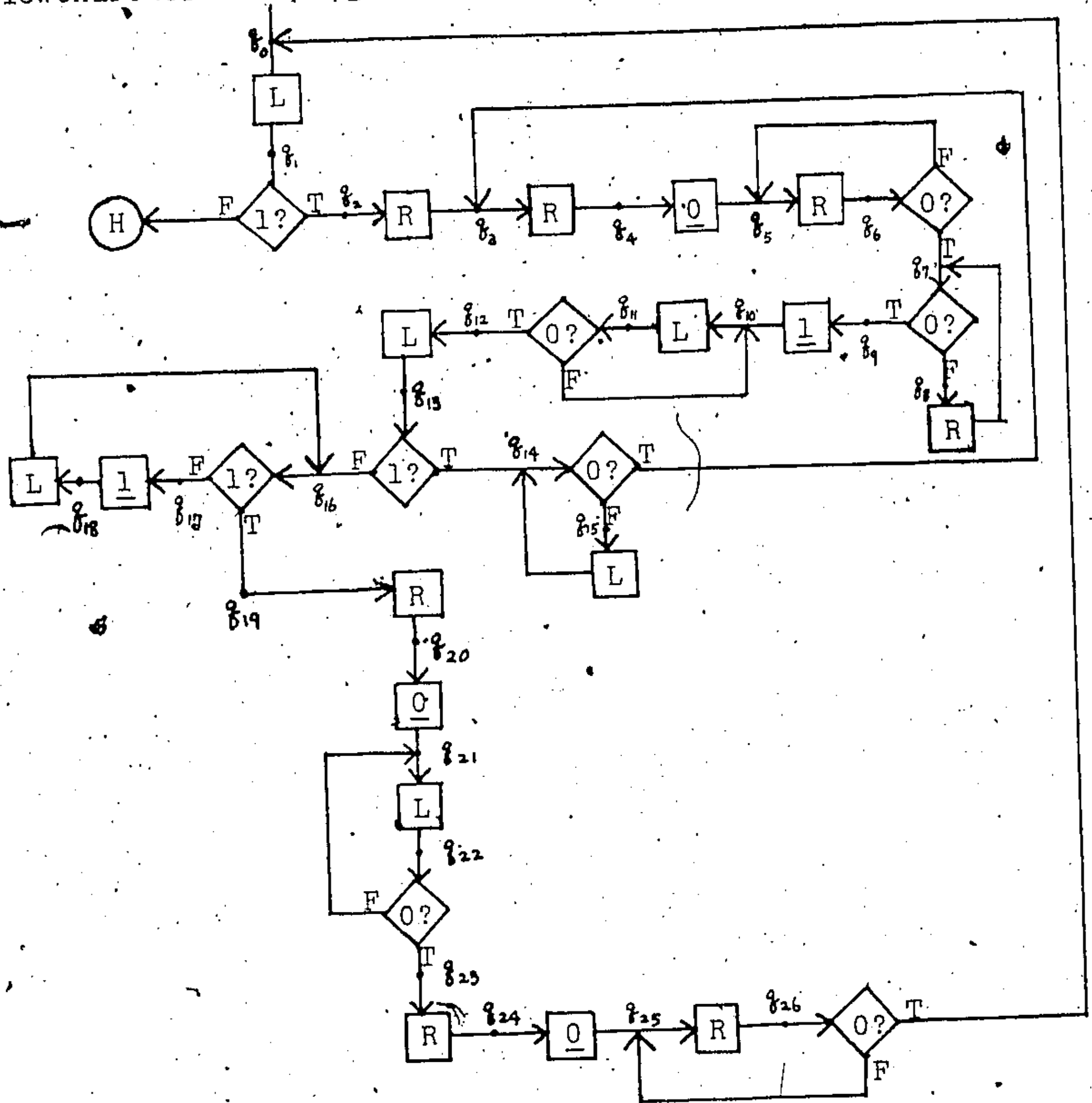
<u>Example 2.1</u>. Consider a unary multiplier where numbers are represented by the number of 1's on a Turing machine of 0-1 alphabet. If the input configuration is : 0...011...1011...10...0

then the output configuration will be of the form:

m n mxn

0...000...0011..,1011.....10...0

The flowchart of the program is:



The following program corresponds to the above flowchart.

$q_0$: DO L GO TO $q_1$

$q_1$: IF 1 THEN GO TO $q_2$ ELSE HALT

$q_2$: DO R GOTO $q_3$

$q_3$: DO R GO TO $q_4$

$q_4$: DO $\underline{0}$ GO TO $q_5$

$q_5$: DO R GO TO $q_6$

$q_6$: IF 0 THEN GO TO $q_7$ ELSE GO TO $q_5$

$q_7$: IF 0 THEN GO TO $q_9$ ELSE GO TO $q_8$

$q_8$: DO R GO TO $q_7$

$q_9$: DO $\underline{1}$ GO TO $q_{10}$

$q_{10}$: DO L GO TO $q_{11}$

$q_{11}$: IF 0 THEN GO TO $q_{12}$ ELSE GO TO $q_{10}$

$q_{12}$: DO L GO TO $q_{13}$

$q_{13}$: IF 1 THEN GO TO $q_{14}$ ELSE GO TO $q_{16}$

$q_{14}$: IF 0 THEN GO TO $q_3$ ELSE GO TO $q_{15}$

$q_{15}$: DO L GO TO $q_{14}$

$q_{16}$: IF 1 THEN GO TO $q_{19}$ ELSE GO TO $q_{17}$

$q_{17}$: DO $\underline{1}$ GO TO $q_{18}$

$q_{18}$: DO L GO TO $q_{16}$

$q_{19}$: DO R GO TO $q_{20}$

$q_{20}$: DO $\underline{0}$ GO TO $q_{21}$

$q_{21}$: DO L GO TO $q_{22}$

$q_{22}$: IF 0 THEN GO TO $q_{23}$ ELSE GO TO $q_{21}$

$q_{23}$: DO R GO TO $q_{24}$

$q_{24}$: DO $\underline{0}$ GO TO $q_{25}$

$q_{25}$: DO R GO TO $q_{26}$

$q_{26}$: IF 0 THEN GO TO $q_0$ ELSE GO TO $q_{25}$

In equation form, the program is:

$$q_0 = Lq_1$$
$$q_1 = 1q_2 + \overline{1}H$$
$$q_2 = Rq_3$$
$$q_3 = Rq_4$$
$$q_4 = \underline{0}q_5$$
$$q_5 = Rq_6$$
$$q_6 = 0q_7 + \overline{0}q_5$$
$$q_7 = 0q_9 + \overline{0}q_8$$
$$q_8 = Rq_7$$
$$q_9 = \underline{1}q_{10}$$
$$q_{10} = Lq_{11}$$
$$q_{11} = 0q_{12} + \overline{0}q_{10}$$
$$q_{12} = Lq_{13}$$
$$q_{13} = 1q_{14} + \overline{1}q_{16}$$
$$q_{14} = 0q_3 + \overline{0}q_{15}$$
$$q_{15} = Lq_{14}$$
$$q_{16} = 1q_{19} + \overline{1}q_{17}$$
$$q_{17} = \underline{1}q_{18}$$
$$q_{18} = Lq_{16}$$
$$q_{19} = Rq_{20}$$
$$q_{20} = \underline{0}q_{21}$$

$$q_{21} = Lq_{22}$$
$$q_{22} = 0q_{23} + \overline{0}q_{21}$$
$$q_{23} = Rq_{24}$$
$$q_{24} = \underline{0}q_{25}$$
$$q_{25} = Rq_{26}$$
$$q_{26} = 0q_{0} + \overline{0}q_{25}$$

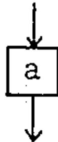
From the above example, we see that in this form, the program is not easy to understand, and hence it is not easy to see that it does what it claimed.

## § 2.3 WHILE Programs.

The concept of WHILE programs, or programs without GO TO statements was extensively discussed in the literature in the last decade [ 4, 10, 14 ], and the purpose of introducing the idea was to give programs which are easier to understand and easier to prove correct.

Definition 2.5. A WHILE program schema with A as the set of operations and P as the set of tests is defined inductively as follows:

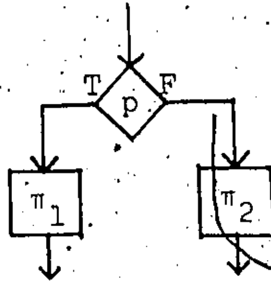
(i) For every $a \in A$, a is a WHILE program schema,




(ii) (Branching) If $\pi_1$ and $\pi_2$ are WHILE program schemata, then for any $p \in P$,

IF p THEN $[\pi_1]$ ELSE $[\pi_2]$

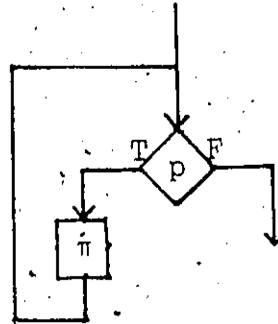
is a WHILE program schema,




(iii) (Looping) If $\pi$ is a WHILE program schema, then for any $p \in P$,

WHILE p DO [$\pi$]

is a WHILE program schema,




(iv) (Concatenating) If $\pi_1$ and $\pi_2$ are WHILE program schemata, then

$\pi_1\pi_2$

is a WHILE program schema,

(v) Only the program schemata that can be derived from (i) by (ii) to (iv) are WHILE program schemata.

Definition 2.6. Let $\Delta = (M, A, P)$ be a memory unit. Then an (A, P) WHILE program schema is called a WHILE program for $\Delta$.

Notation. For convenience in future usage, we adopt the following notation for the cases (i) to (iv) in the previous definition.

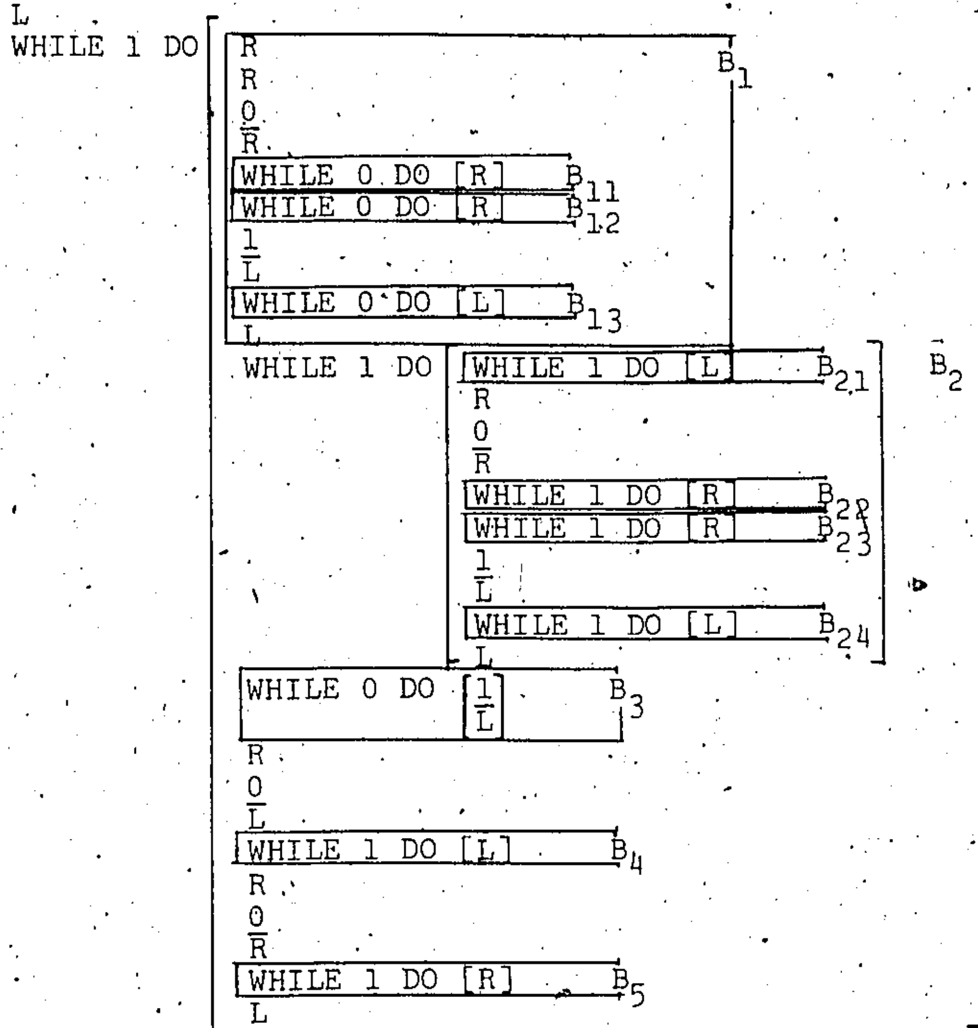
(i) $a$

(ii) $p\pi_1 + \overline{p}\pi_2$

(iii) $(p\pi)^*\overline{p}$

(iv) $\pi_1\pi_2$

Example 2.2 Let us consider the example given in § 2.3. We now express it as a WHILE program.




The WHILE program, as shown in the above, can be considered as consisting of blocks each of which carries out a sub-job. In our case, the blocks and their functions are as indicated below.

$B_1$: Erase an '1' in n and copy it to the right of n.

$B_{11}$: Move right to the first '1' of n.

$B_{12}$: Move right to the space for 'm x n'.

$B_{13}$: Move left to the first '1' in n.

$B_2$: Erase and copy n.

$B_{21}$: Move left to the leftmost '1' in n.

$B_{22}$: Move right to the rightmost '1' in n.

$B_{23}$: Move right to the rightmost '1' in the space for mxn.

$B_{24}$: Move left to pass the space for mxn.

$B_3$: Restore n.

$B_4$: Move left to the leftmost '1' in m.

$B_5$: Move to the '0' between m and n.

# CHAPTER III

## PROGRAM EQUIVALENCE

### § 3.1 Automata and Machines

We introduce the notions of strong and weak equivalence of programs by means of S. Eilenberg's distinction [ 6 ] between automata and machines.

Definition 3.1. An automaton $\mathcal{A}$ over a finite alphabet $\Sigma$, consists of the following entities:

$Q$, a finite set of states;

$I \subseteq Q$, the set of initial states;

$T \subseteq Q$, the set of terminal states;

and $E \subseteq Q \times \Sigma \times Q$, the set of edges of the automaton with elements of the form $(p,\sigma,q)$, denoted as

$$\sigma : p \rightarrow q \qquad \text{or} \qquad p \xrightarrow{\sigma} q$$

with $p, q \in Q$.

Denote $\mathcal{A}$ as $(Q, I, T)$.

Definition 3.2. A path $c$ in $\mathcal{A}$ is a finite sequence of consecutive edges:

$$c = (q_0, \sigma_1, q_1)(q_1, \sigma_2, q_2)\ldots\ldots(q_{k-1}, \sigma_k, q_k).$$

Such a path is said to be of length $k > 0$, and the path begins at $q_0$ and ends at $q_k$. We represent such a path by the abbreviated notation:

$$c = (q_0, \sigma_1, \sigma_2, \ldots\ldots, \sigma_k, q_k).$$

If $k = 0$, then $c$ is the <u>null path</u>; that is, a path from any state $q$ to itself, with label $\Lambda$, and length 0.
The element $s = \sigma_1\sigma_2 \ldots\ldots \sigma_k \in \Sigma^*$, is called the <u>label</u> of $c$, and is denoted by $|c|$.
<u>Definition 3.3</u>. A path $c: i \rightarrow t$ with $i \in I$, $t \in T$, is called a <u>successful path</u>. The <u>behavior</u> of $\mathcal{A}$ is defined to be the set of labels carried by the successful paths in $\mathcal{A}$, and is denoted as $|\mathcal{A}|$.
<u>Definition 3.4</u>. Let $X$ be an arbitrary set and let $\Phi$ be a family of relations $\phi$: $X \rightarrow X$. An <u>X-machine $\mathcal{M}$ of type $\Phi$</u> is defined to be simply a $\Phi$-automaton $(Q, I, T)$.

$$c: \; q_0 \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} \ldots\ldots \xrightarrow{\phi_n} q_n$$

denotes a path in $\mathcal{M}$ of length $n \geq 1$.
<u>Remark</u>. In an automaton, the label $|c|$ of a path $c$ is the 'word' $\phi_1\phi_2 \ldots\ldots \phi_n$ regarded as an element of the free monoid $\Phi^*$ with base $\Phi$. On the other hand, in a machine, the label $|c|$ is the composition $\phi_1\phi_2 \ldots\ldots \phi_n$ viewed as a relation $X \longrightarrow X$.
<u>Definition 3.5</u>. The <u>behavior</u> of $\mathcal{M}$ is the relation $|\mathcal{M}|$ from $X$ to $X$ defined as $|\mathcal{M}| = \cup|c|$, where the union is extended over successful paths $c$ in $\mathcal{M}$.

## § 3.2 Automata and Machines of Programs.

With the notations introduced in the previous section, we shall define the automaton and the machine of a program.

Definition 3.6. The automaton $\mathcal{A}_\pi$ of an (A, P) program schema $\pi = (Q, q_0, h, S)$ is defined over the alphabet $\Sigma = A \cup P$, as the triple (Q', I, T) where

$$Q' = \bar{Q}$$
$$I = \{ q_0 \}$$
$$T = \{ h \}$$

and the set of edges consists of the following:

$$a : q \to r, \quad \text{where } q = ar \text{ is in } S$$

$$\left.\begin{array}{l} p : q \to r \\ \bar{p} : q \to s \end{array}\right\} \text{where } q = pr + \bar{p}s \text{ is in } S.$$

Remark. With the above correspondence between a program schema and the automaton of the program schema, the flowchart of the program schema is essentially the state diagram of its corresponding automaton. The automaton of a program is the automaton of its underlying program schema.

Definition 3.7. The machine $\mathcal{M}_\pi$ of a program $\pi = (Q, q_0, h, S)$ for the memory unit $\Delta = (M, A, P)$ is the above automaton (Q', I, T) over the alphabet $\Sigma = A \cup P$, with the following interpretation of the edges as partial functions on the set of memory states, M:

for an operation $a : M \to M$, 'a' is reinterpreted as a partial function $\underline{a} : M \to M$;

for a test $p : M \to 2$, 'p' is reinterpreted as a partial function $\underline{p} : M \to M$, such that for all m in M, we have

$$\underline{p}(m) = \begin{cases} m, & \text{if } p(m) = 1 \\ \text{undefined}, & \text{if } p(m) = 0; \end{cases}$$

and for a test $p : M \to 2$, '$\bar{p}$' is reinterpreted as a partial function $(\underline{\bar{p}}) : M \to M$, such that for all m in M, we have

$$(\underline{\bar{p}})(m) = \begin{cases} \text{undefined}, & \text{if } p(m) = 1 \\ m, & \text{if } p(m) = 0. \end{cases}$$

We, therefore, extend the association of letters in $\Sigma$ with partial functions from M to M to a function from $\Sigma^*$ to partial functions of M to M by the assignment : $x \mapsto \underline{x}$, defined as:

$$\text{if } x = x_1 x_2 \ldots x_n, \text{ then } \underline{x} = \underline{x}_1 \underline{x}_2 \ldots \underline{x}_n.$$

Notation. For a path $c : q \to r$ in $\mathcal{A}_\pi$, the corresponding path in $\mathcal{M}_\pi$ is denoted as $\underline{c} : q \to r$.

Then, $|\underline{c}| = \underline{|c|}$ .

## 3.3 Equivalence of Programs.

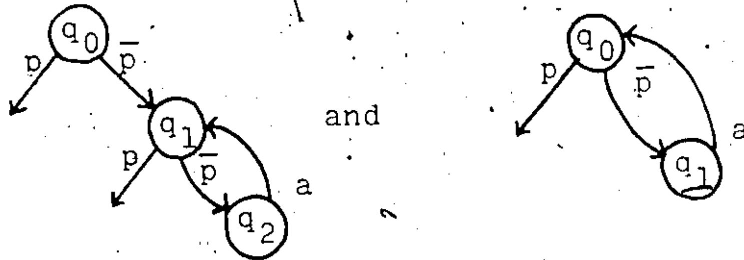
As mentioned in section 2.3, it has been proposed by many authors (Dijkstra, E. W. in " GO TO Statement considered harmful " [4]; Knuth, D. E. and Floyd, R. W. in " Notes on Avoiding GO TO Statements " [10]; Wulf, W. A. in " Programming Without the GO TO " [14]; etc.) that programmers should be trained to write programs without GO TO's; that is, WHILE programs. The reasons for this proposal are that it leads to simpler formal correctness proofs, and experience has shown that fewer programming errors are made.

At this stage, one may be interested in asking " is every program equivalent to a WHILE program ? " The answer is " yes ",

if 'equivalence' means computing the same function; and 'no', if 'equivalence' means having the same execution sequences. We give specific names to the above two kinds of program equivalence, and their definition involving the behaviors of both automata and machines of programs.

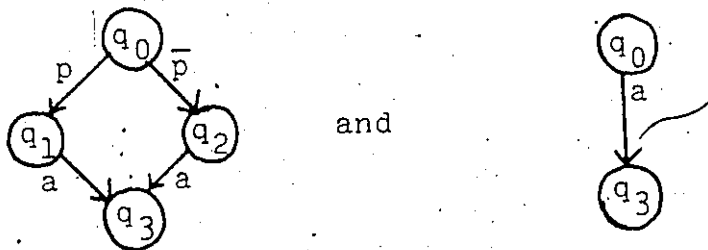
Definition 3.8. Program schemata $\pi_1$ and $\pi_2$ are defined to be strongly equivalent if $|\mathcal{A}\pi_1| = |\mathcal{A}\pi_2|$; and weakly equivalent if $|\mathcal{M}\pi_1| = |\mathcal{M}\pi_2|$ for all memory units $\Delta = (M, A, P)$. We say that programs are strongly or weakly equivalent if their underlying schemata are strongly or weakly equivalent.

Examples (i)



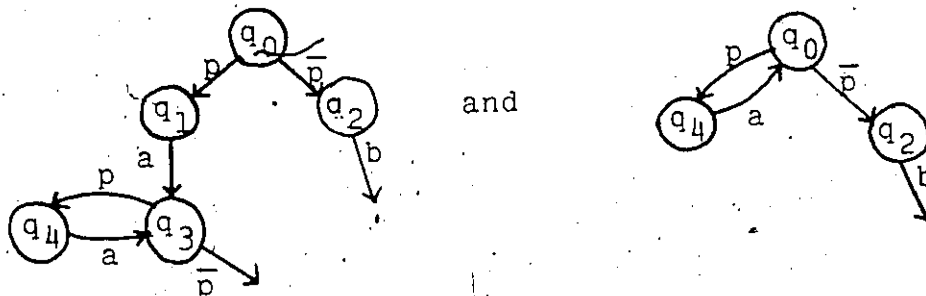

are weakly equivalent.

(ii)




are weakly equivalent.

(iii)

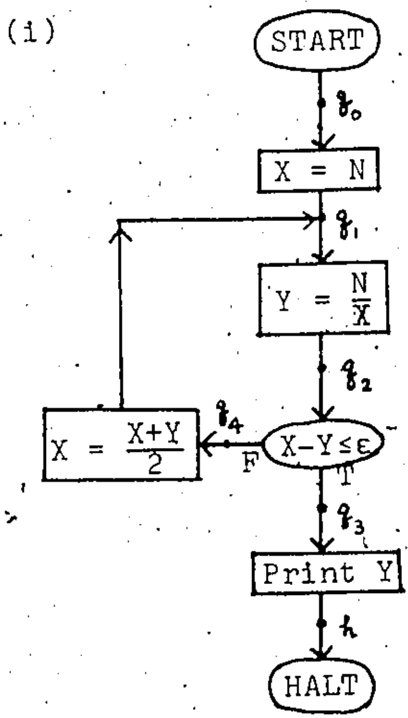


are strongly equivalent.

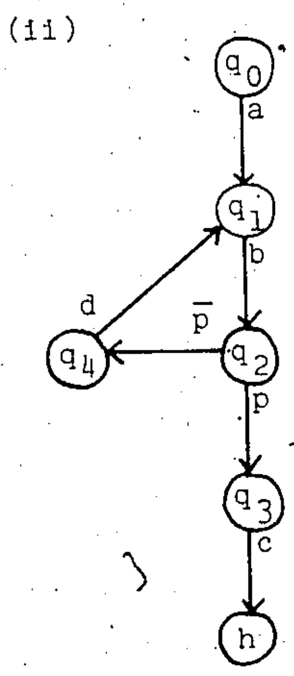
Note. We consider here only strong equivalence of program schemata, but we will later show that if a schema has all the labels distinct (thus excluding examples such as those given in the above), it is strongly equivalent to a WHILE schema if and only if it is weakly equivalent to a WHILE schema, and the WHILE schemata might not be the same in the two cases of equivalence.

Remarks. (1) For any program $\pi$, $\pi$ computes $|\mathcal{M}_\pi|$. Intuitively, for an edge $a : q \rightarrow r$, having $m \in M$ at state $q$ gives us $a(m)$ at state $r$, and for a branching point, one and only one of $p(m)$ and $(\bar{p})(m)$ is defined, and there is no change to the memory state $m$. Consider the following program and its interpretation for finding the square root of a given number N. We have in figure (ii), a, b, c, d representing the corresponding operations and p, the test with $\varepsilon$ as a preassigned value as the tolerance of error by approximation.

(i)




(ii)



Let the triple (n, x, y) be an element of the set of memory states and $\varepsilon = 0.001$ is given. At state $q_0$, we have, say, (25, 0, 0); then at $q_1$ we have (25, 25, 0); at $q_2$, (25, 25, 1). Next we have that " $25 - 1 \leq 0.001$ " is not true, therefore arrive to $q_4$ with (25, 25, 1) unchanged. With (25, $\frac{25+1}{2}$, 1), we come back to $q_1$ and go along the loop consisted of edges: $q_1 \to q_2$, $q_2 \to q_4$, $q_4 \to q_1$, until " $x - y \leq \varepsilon$ " is true, and exit with y as the answer $\approx \sqrt{n}$.

(2) Any two programs being strongly equivalent means they have the same execution sequences; that is, the same order of applying the tests and the operations. On the other hand, two programs are weakly equivalent if they compute the same partial function.

To see that programs do compute partial functions, we have the following lemma and corollaries:

Lemma 3.1. If $c : q \to r$ and $d : q \to r$ are distinct paths in $\mathcal{A}_\pi$, then $\text{Dom}\,|\underline{c}| \cap \text{Dom}\,|\underline{d}| = \emptyset$.

Proof. Let $|c| = c_1 c_2 \cdot \ldots \cdot c_m$ and $|d| = d_1 d_2 \cdot \ldots \cdot d_n$, for $m, n \geq 1$. Let $k \leq m, n$ be the largest integer such that $c_i = d_i$ for all $i \leq k$. Then $c_{i+1} \in P$, as we only have branches for tests. Say $c_{i+1} = \sigma \in P$. Then $d_{i+1} = \overline{\sigma}$, or vice versa. Thus we have $\text{Dom}\,|\underline{c}| \cap \text{Dom}\,|\underline{d}| = \emptyset$.

Corollary 1. For each $m \in M$, and any $q, r \in \overline{Q}$, there is at most one path $\underline{c} : q \to r$ such that $|\underline{c}|(m)$ is defined.

Corollary 2. $|\mathcal{M}_\pi|$ is a partial function.

## § 3.4 The Böhm and Jacopini Construction.

C. Böhm and G. Jacopini [ 2 ], gave a construction which assign to every program $\pi$, a WHILE program $\pi'$ for a different memory unit, such that $|\mathcal{M}_\pi| = |\mathcal{M}_{\pi'}|$. The idea involved in the construction method for the corresponding WHILE program is to include the states as part of the memory states and introduce more tests to direct the transition of states. Let us give a brief account of the construction procedure. In [ 2 ], binary code words are used for the states, but here we use a more 'transparent' construction.

Procedure. Given a program $\pi = (Q, q_0, h, S)$ for a memory unit $\Delta = (M, A, P)$, let the new set of memory states be $M' = M \times \overline{Q}$, then the new set of operations, $A' \subset M'^{M'}$, consists of :

(i) for each $q_i \in \overline{Q}$, $\underline{q}_i \in A'$, defined by :

$$\underline{q}_i(m, q) = (m, q_i), \quad \text{for } m \in M, q \in Q;$$

(ii) for each $a \in A$, the extended operation, $a' \in A'$, defined by :

$$a'(m, q) = (a(m), q), \text{ for } m \in M, q \in Q.$$

The new set of tests, $P' \subset 2^{M'}$, consists of :

(i) the new predicates :

$$p_q(m, q') = \begin{cases} 1, & \text{if } q = q' \\ 0, & \text{otherwise} \end{cases} \quad \text{for } m \in M, q, q' \in \overline{Q}.$$

(ii) the predicates of the original program, extended in the following way : for all $p \in P$, we have $p' \in P'$ such that

$$p'(m, q) = p(m), \quad \text{for } m \in M \text{ and any } q \in Q.$$

Then the corresponding WHILE program takes the following form:

Let $Q = \{ q_0, q_1, q_2, \ldots\ldots, q_n \}$.

WHILE $\bar{p}_h$ IF $p_{q_0}$ THEN ( ..... ) ELSE

IF $p_{q_1}$ THEN ( ..... ) ELSE

⋮

IF $p_{q_n}$ THEN ( ..... ).

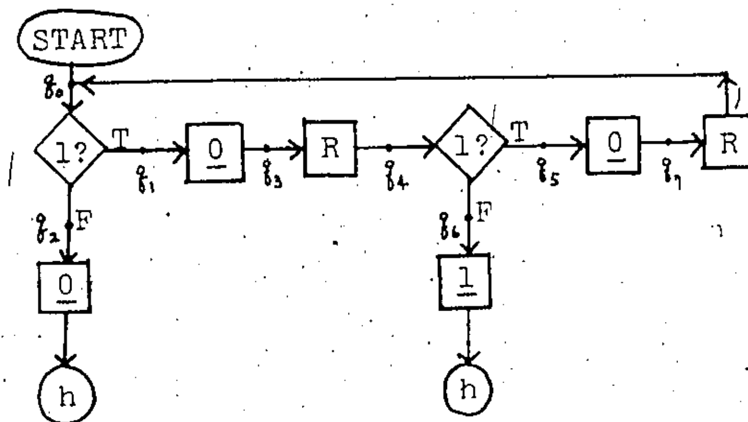
where the brackets correspond respectively to the statememts with labels $q_0, q_1, \ldots\ldots, q_n$, as follows:

There are two possible forms of statements which may occur in the brackets:

(i) 'DO a GO TO $q_j$' is replaced by ' DO a
DO $q_j$.'

(ii) 'IF p THEN GO TO $q_i$ ELSE GO TO $q_j$' is replaced by
'IF p THEN DO $q_i$ ELSE DO $q_j$'.

<u>Example 3.1</u>. The program given here describes the job done by a parity counter. For a string of 0's and 1's as input, we obtain "1" as output indicating that the number of 1's is odd, and we obtain "0" if the case is otherwise.



$q_0$: IF 1 THEN GO TO $q_1$ ELSE GO TO $q_2$

$q_1$: DO $\underline{0}$ GO TO $q_3$

$q_2$: DO $\underline{0}$ GO TO h

$q_3$: DO R GO TO $q_4$

$q_4$: IF 1 THEN GO TO $q_5$ ELSE GO TO $q_6$

$q_5$: DO $\underline{0}$ GO TO $q_7$

$q_6$: DO $\underline{1}$ GO TO h

$q_7$: DO R GO TO $q_0$

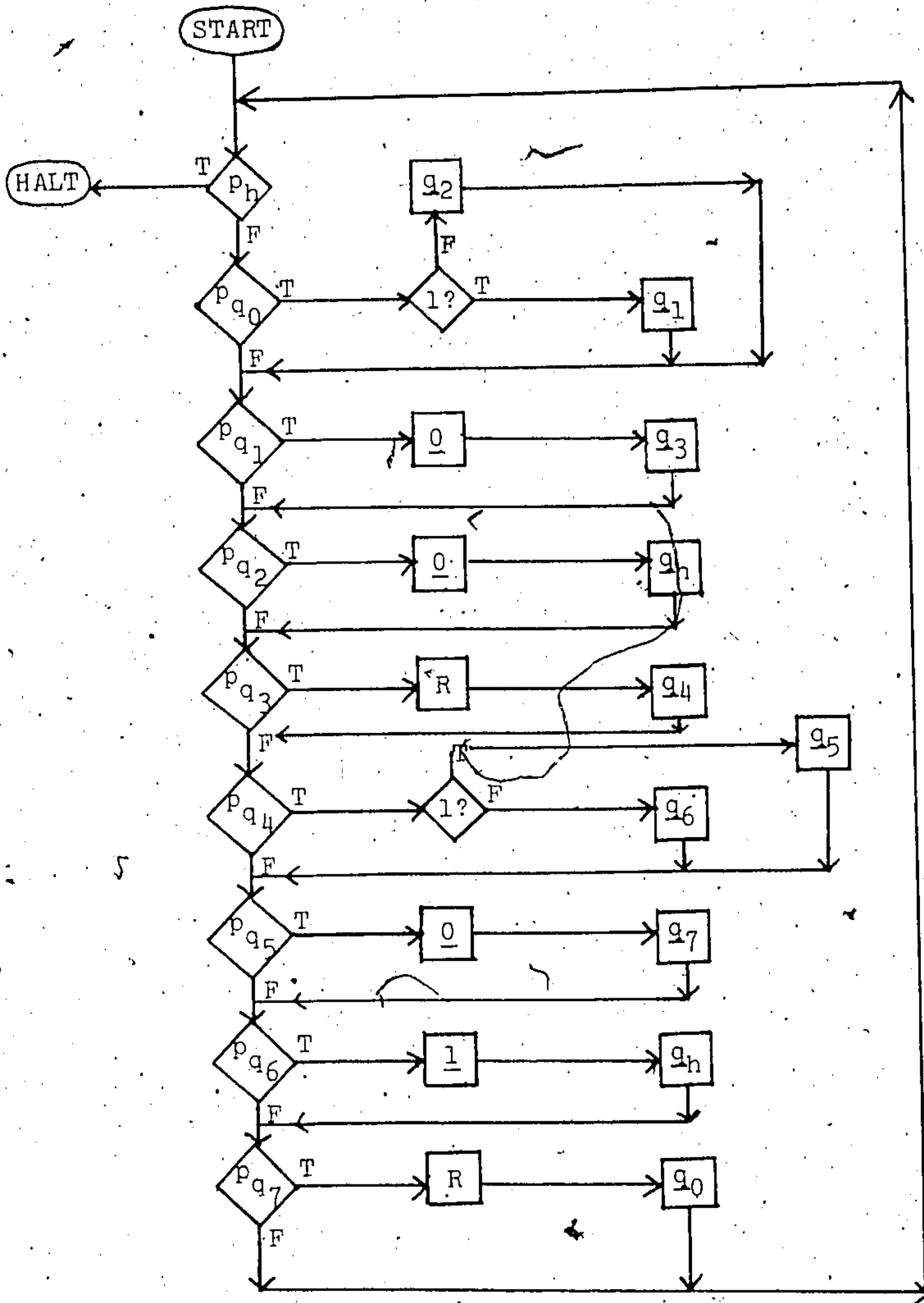
Next, we give a WHILE program which does the same job but is not strongly equivalent to the above program. We need the following symbols:

Let '$p_q$' be the test whether the present state is q, and let '$\underline{q}_i$' be the operation of changing the present state to $q_i$.

The program takes the following form:

WHILE $\overline{P}_h$ IF $p_{q_0}$ THEN IF 1 THEN DO $\underline{q}_1$ ELSE DO $\underline{q}_2$ ELSE
IF $p_{q_1}$ THEN DO $\underline{0}$ DO $\underline{q}_3$ ELSE
IF $p_{q_2}$ THEN DO $\underline{0}$ DO $\underline{q}_h$ ELSE
IF $p_{q_3}$ THEN IF 1 THEN DO R DO $\underline{q}_4$ ELSE
IF $p_{q_4}$ THEN IF 1 THEN DO $\underline{q}_5$ ELSE DO $\underline{q}_6$ ELSE
IF $p_{q_5}$ THEN DO $\underline{0}$ DO $\underline{q}_7$ ELSE
IF $p_{q_6}$ THEN DO $\underline{1}$ DO $\underline{q}_h$ ELSE
IF $p_{q_7}$ THEN DO R DO $\underline{q}_0$

The flowchart of the above WHILE program is as shown in below:



START
HALT
T
p_h
F
q2
F
T
p_q0
1?
T
q1
F
T
p_q1
0
q3
F
T
p_q2
0
q_h
F
T
p_q3
R
q4
F
q5
T
p_q4
1?
F
q6
F
T
p_q5
0
q7
F
T
p_q6
1
q_h
F
T
p_q7
R
q0
F

Remark. Though the two programs involved in the above example do the same job, the flowcharts of the programs do not resemble each other at all. Further, the triviality of the construction suggests that this notion of equivalence to a WHILE program is not of much interest.

# CHAPTER IV

## STRONG EQUIVALENCE TO WHILE PROGRAMS

The fact that every finite automaton recognizes a regular set (one direction of Kleene's Theorem) can be proved by showing that a certain set of equations defining the recognized set is solvable by regular operations,[6]. We show in this chapter that a program is strongly equivalent to a WHILE program if and only if these equations are solvable by a restricted set of operations.

### 4.1 Kleene's Theorem.

Definition 4.1. A subset A of X is regular if either

(i) A is finite or empty;

(ii) for regular sets $A_1$, $A_2$, either

$$A = A_1 . A_2 = \{ a_1 a_2 \mid a_1 \in A_1, a_2 \in A_2 \}$$

or $A = A_1 \cup A_2$

or $A = (A_1)^*$

$$= \{ a_1 a_2 \ldots a_n \mid a_i \in A_1, \text{ for } 0 \le i \le n, \text{ and } n = 0, 1, 2, \ldots \}.$$

(where with $n = 0$, we obtain the empty string $\Lambda$).

Definition 4.2. A regular expression on the set X consists of either (i) $\Lambda$ or $x \in X$; or $\emptyset$;

or (ii) for regular expression $\alpha$, $\beta$, either

$(\alpha + \beta)$ or $\alpha\beta$ or $(\alpha)^*$.

Definition 4.3. If X is a finite set, $\alpha$ is a regular expression over X, then $|\alpha|$ is the regular set constructed recursively as follows:

(i) $|\Lambda| = \{ \Lambda \}$

$|x| = \{x\}$, for $x \in X$

$|\emptyset| = \emptyset$

or (ii) for regular expressions $\alpha$, and $\beta$,

$|(\alpha + \beta)| = |\alpha| \cup |\beta|$

$|\alpha\beta| = |\alpha| . |\beta|$

$|(\alpha)^*| = |\alpha|^*$

The set $|\alpha|$ is called the denotation of the regular expression $\alpha$.

Theorem 4.1. (Kleene) A subset A of $\Sigma^*$, is the behavior of a finite automaton if and only if A is regular.

We are going to state an algorithm justifying the if part of the theorem; that is, by determining for every finite automaton, a regular expression which denotes the behavior of the automaton.

Procedure. Given $\mathcal{A} = (Q, I, T)$, write down equations of the form :

$$R_q = \sum_{a:q\to r} a(R_r) \; [ + \Lambda, \text{ if } q \in T ]$$

(Evidently $R_q = \{ \alpha \mid \exists t \in T \text{ such that } \alpha : q \longrightarrow t \}$ is a solution.)

Then use the following rules to solve the equations :

(i) substitution;

(ii) if $\gamma = \alpha + \beta\gamma$, where $\alpha$, $\beta$, $\gamma$ are any regular expressions and $\Lambda \notin |\beta|$, then $\gamma = \beta^*\alpha$.

Define the regular expression:

$$R = \sum_{q \in I} R_q .$$

Then $|\mathcal{A}| = |R|$.

Proof. We shall show that,

(a) such a system of equations has a unique solution;

(b) the rules preserve equality, so that if one can deduce $R_q = \hat{\rho}$, then $|\rho| = \{ \alpha \mid \exists t \in T \text{ such that } \alpha : q \to t \}$;

(c) one can always solve such a system by rules (i) and (ii).

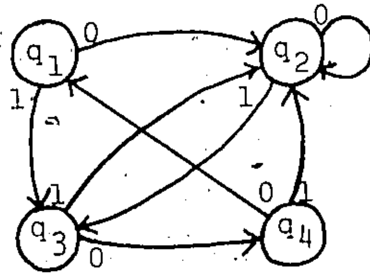
To prove (a), consider the matrix equation $X = AX + B$, over the semiring of subsets of $\Sigma^*$ (where addition is '$\cup$', and product is '.'). If $A = (A_{ij})$ where $A_{ij} \subset \Sigma^*$ and $\Lambda \notin A_{ij}$, then the unique solution is $X = B + AB + A^2B + \ldots\ldots = A^*B$, as follows: If X is a solution, we want to show that $A^*B \subset X$. Since we have that $X = AX + B$, thus $AX \subset X$ and $B \subset X$. Then $AB \subset AX \subset X$. In the same manner, $A(AB) \subset AX \subset X$, ..... We have that $A^*B \subset X$. On the other hand, if $X = (X_i)$ is a solution, and $x \in X_i$, then we want to show that $x \in (A^*B)_i$. Now $x \in X_i = (AX)_i + B_i$, we have case (i), if $x \in B_i$, then $x \in (A^*B)_i$.

case (ii), if $x \in (AX)_i$, we must have $x = uv$, say, with $u \in (A_{ij})$ and $v \in X_j$ for some j, and $|v| < |x|$, as $\Lambda \notin A_{ij}$. If $v \in B_i$, we have case (i); otherwise, we repeat the process for v, with $|v|$ diminishing. Eventually we have $x \in (A.A\ldots AB)_i \subset (A^*B)_i$.

For part (b), it is obvious that the substitution rule preserves equality. Rule (ii) is a special case of part (a) when A is a 1x1 matrix.

For part (c), we will show that using rules (i) and (ii) we can eliminate the variables in turn. If, for some i, variables $R_{q_1}$, $R_{q_2}$, ....., $R_{q_i}$ have been eliminated, then if $R_{q_{i+1}}$ does not appear on the right hand side of the equation for $R_{q_{i+1}}$, we can eliminate it by substitution; otherwise, use rule (ii) to eliminate it.

<u>Example 4.1</u>. Consider $\mathcal{A} = (Q, \{q_1\}, \{q_3, q_4\})$ with the following state diagram :




The equations describing the above diagram are :

$$R_1 = 0R_2 + 1R_3 \quad (1)$$

$$R_2 = 0R_2 + 1R_3 \quad (2)$$

$$R_3 = 1R_2 + 0R_4 + \Lambda \quad (3)$$

$$R_4 = 0R_1 + 1R_2 + \Lambda \quad (4)$$

and $R = R_1$

By rule (ii), (2) becomes : $R_2 = 0^*1R_3$.

Substitute into (3), we have

$$R_3 = 10^*1R_3 + 0R_4 + \Lambda$$

$$= (10^*1)^*(0R_4 + \Lambda) \text{ , by rule (ii)}$$

and $R_2 = 0^*1(10^*1)^*(0R_4 + \Lambda)$.

From (1), $R_1 = 0R_2 + 1R_3$

$$= 00^*1R_3 + 1R_3$$

$$= (00^*1 + 1)(10^*1)^*(0R_4 + \Lambda)$$

Then $R_4 = 0(00^*1 + 1)(10^*1)^*(0R_4 + \Lambda) + 1)^*1)^*(0R_4 + \Lambda) + \Lambda$

$= [0(00^*1 + 1)(10^*1)^*0 + 10^*1(10^*1)^*0]R_4 + 0(00^*1 + 1)$

$.(10^*1)^* + 10^*1(10^*1)^* + \Lambda$

$= [0(00^*1 + 1)(10^*1)^*0 + 10^*1(10^*1)^*0]^*(0(00^*1 + 1)$

$.(10^*1)^* + 10^*1(10^*1)^* + \Lambda)$

Hence $R = R_1$

$= (00^*1 + 1)(10^*1)^* 0[0(00^*1 + 1)(10^*1)^*0 + 10^*1(10^*1)^*0]^*$

$.(0(00^*1 + 1)(10^*1)^* + 10^*1(10^*1)^* + \Lambda) + \Lambda$

is the expression giving the behavior of $\mathcal{A}$.

## 4.2 Strong Equivalence between Programs and WHILE Programs.

Since a WHILE program, as defined in section 2.3, is not, strictly speaking, a program as defined in section 2.2, one cannot say directly when a WHILE program and a program are strongly equivalent. One could define, in an obvious way, an automaton $\mathcal{A}_\pi$ associated with a WHILE program $\pi$, and hence define its strong equivalence to a program, but it is easier to proceed as follows.

We need a class of regular expressions which, for a memory unit $\Delta = (M, A, P)$, describes all computational sequences (or labels of paths in the flowchart) produced by the flowchart of a WHILE program.

Definition 4.4. For a WHILE program $\pi$, define the regular expression $\mathcal{E}_\pi$ of computational sequences of $\pi$ by

(i) if $\pi = a$, then $\mathcal{E}_\pi = a$;

(ii) if $\pi = p\pi_1 + \bar{p}\pi_2$, then $\mathcal{E}_\pi = p\,\mathcal{E}\pi_1 + \bar{p}\,\mathcal{E}\pi_2$;

if $\pi = (p\pi_1)^*\bar{p}$, then $\mathcal{E}_\pi = (p\,\mathcal{E}\pi_1)^*\,\bar{p}$;

if $\pi = \pi_1\pi_2$, then $\mathcal{E}_\pi = \mathcal{E}\pi_1\,\mathcal{E}\pi_2$.

Definition 4.5. A WHILE program $\pi$ is strongly equivalent to a program $\pi'$ if $|\mathcal{E}_\pi| = |\mathcal{A}_{\pi'}|$.

Definition 4.6. (Knuth and Floyd [10]) The set of regular expressions, $W_0$ over $A \cup P$ is defined to consist of :

(i) all $a \in A$;

(ii) if $\sigma_1$, $\sigma_2$, $\sigma \in W_0$, and $p \in P$, then

$p\sigma_1 + \bar{p}\sigma_2 \in W_0$

$(p\sigma_1)^*\bar{p} \in W_0$

$\sigma_1\sigma_2 \in W_0$

Remark. A regular expression $\sigma$ is of the form $\mathcal{E}_\pi$ if and only if $\sigma \in W_0$. Thus a program $\pi$ is strongly equivalent to a WHILE program if and only if there exists $\sigma \in W_0$ such that $|\sigma| = |\mathcal{A}_\pi|$. On the other hand, if $\sigma \in W_0$, then define $W(\sigma)$ to be the WHILE program of the following form:

(i) when $\sigma = a \in A$,

$W(a) = a$

(ii) when $\sigma = p\sigma_1 + \bar{p}\sigma_2$, for $\sigma_1$, $\sigma_2 \in W_0$ and $p \in P$,

$W(p\sigma_1 + \bar{p}\sigma_2) = pW(\sigma_1) + \bar{p}W(\sigma_2)$;

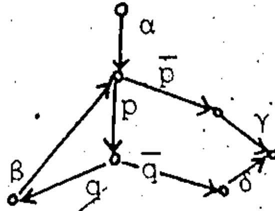
when $\sigma = (p\sigma_1)^*\bar{p}$, for $\sigma_1 \in W_0$ and $p \in P$,

$W((p\sigma_1)^*\bar{p}) = (pW(\sigma_1))^*\bar{p}$;

when $\sigma = \sigma_1\sigma_2$, for $\sigma_1$, $\sigma_2 \in W_0$,

$W(\sigma_1\sigma_2) = W(\sigma_1)W(\sigma_2)$.

<u>Theorem 4.2.</u> (Hopcroft, [10, p. 26]) If a program contains the configuration :




where $\alpha, \beta, \gamma, \delta \in (A \cup P)^*$ and $p, q \in P$, then it is not strongly equivalent to a WHILE program.

<u>Proof.</u> Let $\phi$ denote the computational sequences producible by the above flowchart. Then, $\phi = \alpha(pq\beta)^*(\bar{p}\gamma + p\bar{q}\delta)$.

Suppose to the contrary, there exists $\sigma \in W_0$ such that $|\sigma| = |\phi|$. Since $|\phi|$ contains arbitrarily many 'pq', then $\sigma$ contains $\psi^*$ with pq in $\psi$. Since $\sigma \in W_0$, we have either $(p(q.....))^*\bar{p}$ occurring in $\sigma$ with regular expression $(q.....) \in W_0$ or $(r(...pq...))^*\bar{r}$ with $(...pq...) \in W_0$. We arrive to contradiction, for both cases. Since if q occurs in $\theta \in W_0$, then $\bar{q}$ also occurs in $\theta$, but from the above that $\bar{q}$ has to occur infinitely often in $|\sigma| = |\phi|$, and it is not the case for $\bar{q}$ does not occur arbitrarily often in $|\phi|$.

We give here without proof the theorem due to Kosaraju, [14].

<u>Theorem 4.3.</u> Assume $|\mathcal{A}_\pi| \neq \emptyset$. If $\pi$ has no loop with more than one exit then $\pi$ is weakly equivalent to a WHILE schema. If all labels are different, then if $\pi$ is weakly equivalent to a WHILE schema, then $\pi$ has no loop with more than one exit.

### 4.3 Algorithm for Strong Equivalence to a WHILE Program.

Given a program $\pi = (Q, q_0, h, S)$, consider the following deduction system $D(\pi)$ :

Axioms : S

Deductive Rules :

(I) If $q = \sigma r$, for $\sigma \in (A \cup P)^*$, $r \in \overline{Q}$, we may substitute $\sigma r$ for q in any equations.

(II) If $q = p\sigma_1 r + \overline{p}\sigma_2 r$, for $\sigma_1, \sigma_2 \in (A \cup P)^*$, $r \in \overline{Q}$, $p \in P$, then we have $q = (p\sigma_1 + \overline{p}\sigma_2)r$.

(III) If $q = p\sigma_1 q + \overline{p}\sigma_2 r$, for $\sigma_1, \sigma_2 \in (A \cup P)^*$, $r, s \in \overline{Q}$, $p \in P$, we may deduce $q = (p\sigma_1)^*\overline{p}\sigma_2 r$.

(IV) If $q = p\sigma_1 r + \overline{p}\sigma_2 s$, for $\sigma_1, \sigma_2 \in (A \cup P)^*$, $r, s \in \overline{Q}$, and $p \in P$, we also have $q = \overline{p}\sigma_2 s + p\sigma_1 r$.

Definition 4.7. For any automaton $\mathcal{A}$, let $\mathcal{A}^t$ denote the automaton obtained from $\mathcal{A}$ by removing all states which do not lie on any of the successful paths. $\mathcal{A}^t$ is called the trim part of $\mathcal{A}$ [6].

Lemma 4.1. For any automaton $\mathcal{A}$, we have $|\mathcal{A}| = |\mathcal{A}^t|$.

Proof. By definition, $\mathcal{A}$ is the set of labels carried by successful paths in $\mathcal{A}$. Since $\mathcal{A}^t$ includes all and only those states lying on some successful path, we thus have $|\mathcal{A}| = |\mathcal{A}^t|$.

Theorem 4.4. For any program $\pi = (Q, q_0, h, S)$ with $|\mathcal{A}_\pi| \neq \emptyset$, $D(\pi)$ yields $q_0 = \sigma h$ if and only if $\sigma \in W_0$, and $|\mathcal{A}_\pi| = |\sigma|$; that is, if and only if $\pi$ is strongly equivalent to a WHILE program with computational sequences denoted by $\sigma$.

Proof. Suppose that $D(\pi)$ yields $q_0 = \sigma h$. In the derivation, let $S_0 = S$, and $S_{i+1}$ be the set of equations obtained by applying the next step of the derivation to the equations $S_i$.

We need only to prove by induction on n, that for all n, $S_n$ consists of equations of either of the following forms :

(i) $q = \sigma r$ with $q \in Q$, $\sigma \in W_0$ and $r \in \overline{Q}$;

(ii) $q = p\sigma r + \overline{p}\tau s$ with $\sigma, \tau \in W_0 \cup \{\Lambda\}$, and $r, s \in \overline{Q}$.

When $n = 0$, either $q = ar$ with $a \in A$ or $q = pr + \overline{p}s$, so (i) and (ii) hold.

Assume the truth of the above up to the case n.

Applying rule (I) to (i) $q = \sigma r$ and $r = \tau s$ in $S_n$, we get
$q = \sigma\tau s$ and if $\sigma, \tau \in W_0$, $\sigma\tau \in W_0$.
(ii) $q = p\sigma r + \overline{p}\tau s$ and $r = \delta r'$ in $S_n$, we get $q = p\sigma\delta r' + \overline{p}\tau s$, and $\sigma\delta \in W_0 \cup \{\Lambda\}$.

Applying rule (II) to $q = p\sigma_1 r + \overline{p}\sigma_2 r$ with $\sigma_1, \sigma_2 \in W_0$, we have $q = (p\sigma_1 + \overline{p}\sigma_2)r$, and $p\sigma_1 + \overline{p}\sigma_2 \in W_0$.

Applying rule (III) to $q = p\sigma_1 q + \overline{p}\sigma_2 r$ with $\sigma_1, \sigma_2 \in W_0$, we get $q = (p\sigma_1)^*\overline{p}\sigma_2 r$ and $(p\sigma_1)^*\overline{p}\sigma_2 \in W_0$.

Applying rule (IV) to $q = p\sigma_1 r + \overline{p}\sigma_2 s$ for $\sigma_1, \sigma_2 \in W_0$, the resulting equation : $q = \overline{p}\sigma_2 s + p\sigma_1 r$, is still of the proper form.

The above statement is true for $S_{n+1}$.

Thus if $q_0 = \sigma h \in S_n$ for some n, we have $\sigma \in W_0$.

Note that $|\mathcal{A}_\pi| = |\sigma|$, since rules (I) to (IV) are special cases of Kleene's solution rules, and identities for regular expressions.

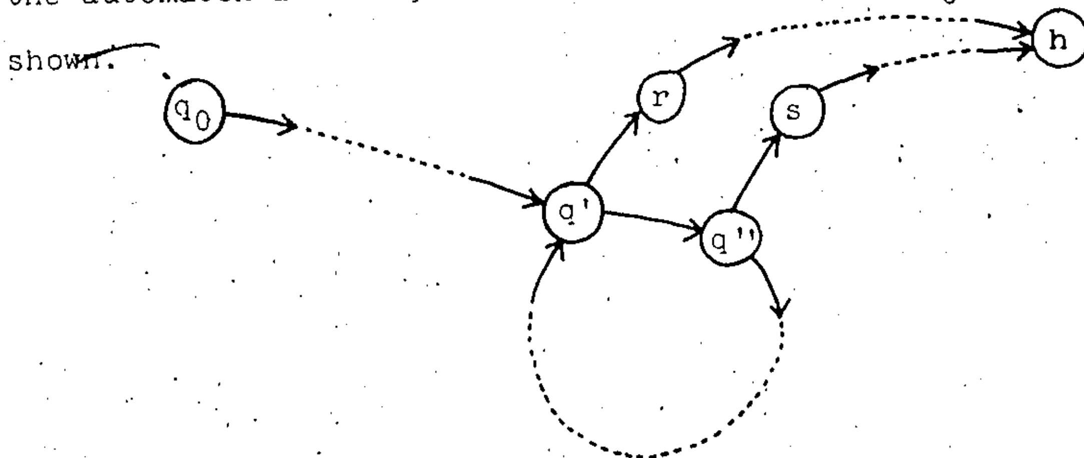
Conversely, to prove that there exists a solution, it is sufficient to prove that when applying the following algorithm to the trim part of $\mathcal{A}_\pi$, we obtain a solution of the form $q_0 = \sigma h$.
Note that since $|\mathcal{A}_\pi| \neq \emptyset$, then $\mathcal{A}_\pi^{\ t}$ is not the empty automaton.

ALGORITHM : (i) Apply rules (II), (III), (IV), if possible.

(ii) Select an equation of the form $q = \sigma r$, with $r \neq q$ and eliminate $q$, by rule (I).

(iii) If more than one variable remain, go to (i); otherwise by iterated use of rule (I), obtain a solution of the form $q_0 = \sigma h$.

The algorithm fails to give a solution if there is no equation of the form $q = \sigma r$ with $r \neq q$, when we come to step (ii). There are no equations $q = \sigma q$, since the automaton is trim, so all equations are of the form : $q = p\sigma r + p\tau s$, with $r \neq s$ (hence one of $r$, $s$ is in Q), otherwise rule (II) can be applied; and $r, s \neq q$, otherwise rule (III) can be applied.
Thus each $q$ in the remaining equations has a distinct successor in Q. Thus there is a non-trivial loop as indicated, and since the automaton is trim, there are connections to $q_0$ and $h$ as shown.



By Theorem 4.2 given in section 4.2 (due to Hopcroft), it contradicts the fact that $\sigma \in W_0$ and $|\sigma| = |\mathcal{A}|$.
Thus we proved the existence of the solution $q_0 = \sigma h$ from $D(\pi)$, if and only if $\sigma \in W_0$ and $|\mathcal{A}_\pi| = |\sigma|$.

Corollary 1. If for any program $\pi$, $D(\pi)$ yields $q_0 = \sigma h$, then $\pi$ is strongly equivalent to $W(\sigma)$.

Corollary 2. If the program $\pi$, is strongly equivalent to a WHILE program, then the following algorithm solves the system $D(\pi)$ :

(i) for each equation, apply rule (II), if possible;

(ii) for each equation, apply rule (III), if possible, using rule (IV) if required to apply rule (III);

(iii) select an equation of the form $q = \sigma r$ and eliminate $q$ by substitution, rule (I);

(iv) if more than one equation remains, go to (i); otherwise by iterated use of rule (I), obtain a solution of the form $q = \sigma h$.

Theorem 4.5. If all the labels of a schema $\pi$ are different, then $\pi$ is weakly equivalent to a WHILE schema if and only if $\pi$ is strongly equivalent to a WHILE schema.

Proof. By Kosaraju's theorem (Theorem 4.3), it is necessary and sufficient for weak equivalence (with all the labels distinct) that $\pi$ contains no loop with two exits. By Hopcroft's theorem (Theorem 4.2), it is necessary for strong equivalence that $\pi$ contains no loop with two exits, and by the above theorem it is sufficient.

Example 4.2. Consider Example 3.1, given in section 3.4, of a parity counter. We want to show that this program is not strongly equivalent to a WHILE program.

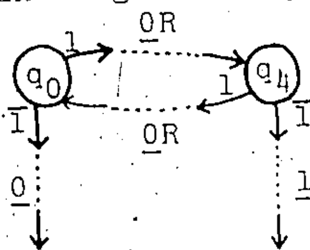
$$q_0 = 1q_1 + \bar{1}q_2$$
$$q_1 = \underline{0}q_3$$
$$q_2 = \underline{0}h$$
$$q_3 = Rq_4$$
$$q_4 = 1q_5 + \bar{1}q_6$$
$$q_5 = \underline{0}q_7$$
$$q_6 = \underline{1}h$$
$$q_7 = Rq_0$$

Solve by the algorithm, we proceed as follows.

By substitution, the equations are reduced to :

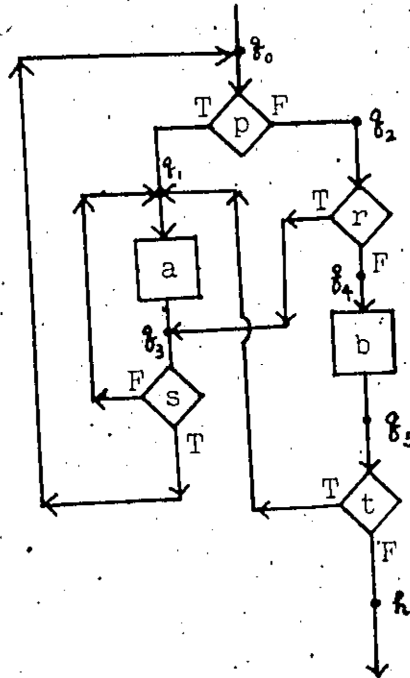
$$q_0 = 1\underline{0}Rq_4 + \bar{1}\underline{0}h$$
$$q_4 = 1\underline{0}Rq_0 + \bar{1}\underline{1}h$$

We then observe that this is actually the situation of a loop with two exits. In diagram form, we have :




By Theorem 4.3, there is no solution of the form $q_0 = \sigma h$ with $\sigma \in W_0$. Thus the program is not strongly equivalent to any While program.

<u>Example 4.3</u>. We want to apply our algorithm to determine if there is a WHILE program which is strongly equivalent to the following program:




The equations describing the above flowchart are :

$$q_0 = pq_1 + \bar{p}q_2$$

$$q_1 = aq_3$$

$$q_2 = rq_3 + \bar{r}q_4$$

$$q_3 = sq_0 + \bar{s}q_1$$

$$q_4 = bq_5$$

$$q_5 = tq_1 + \bar{t}h$$

Applying rule (I), the equations become :

$$q_0 = paq_3 + \bar{p}q_2$$

$$q_2 = rq_3 + \bar{r}bq_5$$

$$q_3 = sq_0 + \bar{s}aq_3$$

$$q_5 = taq_3 + \bar{t}h$$

Applying rule (III) to the third equation, we get :

$$q_3 = (\bar{s}a)^*sq_0$$

Thus, the rest of the equations can be simplified to the following form :

$$q_0 = pa(\bar{s}a)^*sq_0 + \bar{p}q_2$$

$$q_2 = r(\bar{s}a)^*sq_0 + \bar{r}bq_5$$

$$q_5 = ta(\bar{s}a)^*sq_0 + \bar{t}h$$

By iterated use of rules (I) to (IV), we proceed as follows :

$$q_0 = [pa(\bar{s}a)^*s]^*\bar{p}q_2$$

$$q_2 = r(\bar{s}a)^*[pa(\bar{s}a)^*s]^*\bar{p}q_2 + \bar{r}bq_5$$

Thus, $q_2 = \{r(\bar{s}a)^*[pa(\bar{s}a)^*s]^*\bar{p}\}^*\bar{r}bq_5$

Then, $q_5 = ta(\bar{s}a)^*s[pa(\bar{s}a)^*s]^*\bar{p}\{r(\bar{s}a)^*s[pa(\bar{s}a)^*s]^*\bar{p}\}^*\bar{r}bq_5 + \bar{t}h$

or $q_5 = (ta(\bar{s}a)^*s[pa(\bar{s}a)^*s]^*\bar{p}\{r(\bar{s}a)^*s[pa(\bar{s}a)^*s]^*\bar{p}\}^*\bar{r}b)^*\bar{t}h.$

Hence, we have the solution as :

$$q_0 = [pa(\bar{s}a)^*s]^*\bar{p}\{r(\bar{s}a)^*s[pa(\bar{s}a)^*s]^*\bar{p}\}^*\bar{r}b(ta(\bar{s}a)^*s [pa(\bar{s}a)^*s]^*\bar{p}\{r(\bar{s}a)^*s[pa(\bar{s}a)^*s]^*\bar{p}\}^*\bar{r}b)^*\bar{t}h.$$

The WHILE program obtained is as follows :

```
WHILE p DO a
        WHILE s̄ DO a
WHILE r WHILE s̄ DO a
        WHILE p DO a
                WHILE s̄ DO a
DO b
WHILE t DO a
        WHILE s̄ DO a
        WHILE p DO a
                WHILE s̄ DO a
        WHILE r WHILE s̄ DO a
                WHILE p DO a
                        WHILE s̄ DO a
        DO b
```

The state diagram of the above WHILE program takes the following form :

Example 4.4. Let us consider another program in which the algorithm fails to work.

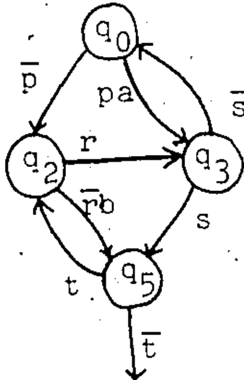
Equations in the set S are :

$$q_0 = pq_1 + \bar{p}q_2$$
$$q_1 = aq_3$$
$$q_2 = rq_3 + \bar{r}q_4$$
$$q_3 = \bar{s}q_0 + sq_5$$
$$q_4 = bq_5$$
$$q_5 = tq_2 + \bar{t}h$$

By rule (I), the substitution rule, we reduce the number of equations by two.

$$q_0 = paq_3 + \bar{p}q_2$$
$$q_2 = rq_3 + \bar{r}bq_5$$
$$q_3 = \bar{s}q_0 + sq_5$$
$$q_5 = tq_2 + \bar{t}h.$$

We arrive now to a situation where none of the four deductive rules can be applied. The diagram form of the situation is :




For the loop $q_0 \to q_2 \to q_3 \to q_0$, we have two exits from $q_2$ and $q_3$ both to $q_5$. Thus, this program is not strongly equivalent to any WHILE program.

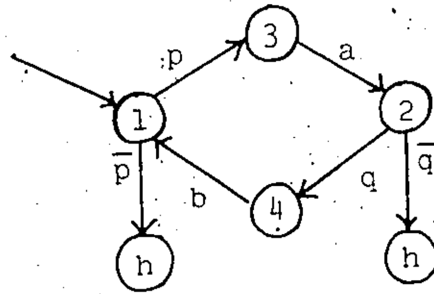
# CHAPTER V

## GENERALIZED WHILE PROGRAMS WITH REPEAT-EXIT

### 5.1 Examples to Illustrate the Generalized WHILE algorithm.

In order to give the idea involved in the generalized WHILE algorithm, which is rather involved with notation, we first illustrate the procedure by some examples.

Example 5.1. We want to translate the following program into the generalized WHILE form.




Equations representing the above program are as follows : (Note that for simplicity, we use numbers 1, 2, 3, ... to represent the statement labels $q_0$, $q_1$, $q_2$, ...)

$$1 = p3 + \overline{p}h$$

$$2 = q4 + \overline{q}h$$

$$3 = a2$$

$$4 = b1$$

By rule (I) of the algorithm given in section 4.3 for translating programs into the WHILE form, we may eliminate statement labels 3 and 4 by substituting them into the other two.

Thus we get,

$$1 = pa2 + \bar{p}h$$

$$2 = qb1 + \bar{q}h$$

None of the rules in WHILE algorithm can be applied. The next step is to replace 'h' by 'H' and simplify the equation

$$q = p\alpha r + \bar{p}\beta H \quad \text{to} \quad q = (p\alpha + \bar{p}\beta H)r.$$

Formally, H will act as a left zero (i.e. for all x, Hx = H), corresponding to the fact that the execution stops at H.

We thus have,

$$1 = (pa + \bar{p}H)2$$

$$2 = (qb + \bar{q}H)1$$

The substitution rule is used to obtain the following :

$$1 = (pa + \bar{p}H)(qb + \bar{q}H)1$$

The equation is now of the form $q = \alpha q$. We write $q = (\ \alpha\ )^{\infty}$, with $(\ \alpha\ )^{\infty}$ ~~to be~~ interpreted as :

'WHILE TRUE DO $\alpha$'

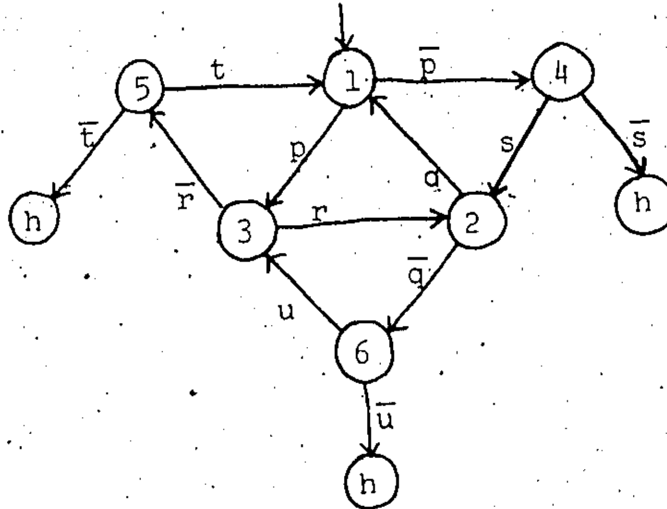
Hence, the solution is of the following form :

$$1 = ((pa + \bar{p}H)(qb + \bar{q}H))^{\infty}.$$

With the interpretations for $(\ \alpha\ )^{\infty}$ and 'H' as absolute halt, the generalized WHILE program is as follows :

```
WHILE TRUE DO IF p THEN DO a
                   ELSE HALT
              IF q THEN DO b
                   ELSE HALT
```

<u>Example 5.2</u>. In this example the following program is translated into a generalized WHILE program with REPEAT-EXIT, (to be explained).




Note that, for simplicity, we purposely omit the edges for operations, since they can be represented by equations of the form $q = ar$, with $q \neq r$. Then by the substitution rule of the algorithm given in section 4.3, we can always eliminate equations of the form $q = ar$.

Equations representing the above program are as follows :

$$1 = p3 + \bar{p}4$$

$$2 = q1 + \bar{q}6$$

$$3 = r2 + \bar{r}5$$

$$4 = s2 + \bar{s}h$$

$$5 = t1 + \bar{t}h$$

$$6 = u3 + \bar{u}h$$

Replace the h's by the H's and simplify to the following form :

$$4 = (s + \bar{s}H)2$$

$$5 = (t + \bar{t}H)1$$

$$6 = (u + \bar{u}H)3$$

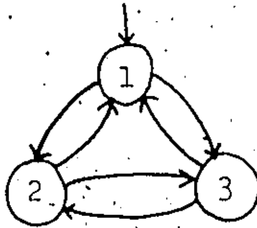
Substituting the above into the rest of the equations we get :

$$1 = p3 + \bar{p}(s + \bar{s}H)2$$

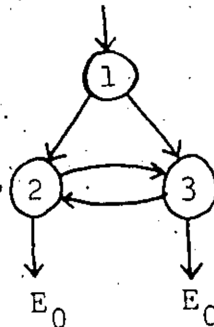
$$2 = q1 + \bar{q}(u + \bar{u}H)3$$

$$3 = r2 + \bar{r}(t + \bar{t}H)1$$

The transition of states can be shown by a simplified diagram.




There are paths from the initial state, 1, to itself, we replace all the occurrences of '1' on the right-hand-sides of the equations by '$E_0$' which behaves similarly as H's, i.e. for all x, $E_0x = E_0$, (the following diagram justifies the above) and is interpreted as the exit and the return to the initial state.




The collection of equations now has the following form :

$$1 = p3 + \bar{p}(s + \bar{s}H)2$$

$$2 = qE_0 + \bar{q}(u + \bar{u}H)3 = (qE_0 + \bar{q}(u + \bar{u}H))3$$

$$3 + r2 + \bar{r}(t + \bar{t}H)E_0 = (r + \bar{r}(t + \bar{t}H)E_0)2$$

Now there is no path from the initial state to itself, we take its successor-states '2' and '3' as the new initial states for

the two subprograms, and repeat again the process of checking whether there is a path from the present initial state to itself. For the part with the state '2', as the initial state, we have,

$$2 = (qE_0 + \bar{q}(u + \bar{u}H))(r + \bar{r}(t + \bar{t}H)E_0)2$$

Thus, $$2 = ((qE_0 + \bar{q}(u + \bar{u}H))( r + \bar{r}(t + \bar{t}H)E_0))^\infty$$

For the part with the state '3', as the initial state, we have,

$$3 = (r + \bar{r}(t + \bar{t}H)E_0)(qE_0 + \bar{q}(u + \bar{u}H))3$$

Thus, $$3 = ((r + \bar{r}(t + \bar{t}H)E_0)(qE_0 + \bar{q}(u + \bar{u}H)))^\infty$$

Substituting the expressions for '2' and '3' into that for '1', we obtain the following :

$$1 = p((r + \bar{r}(t + \bar{t}H)E_0)(qE_0 + \bar{q}(u + \bar{u}H)))^\infty + \bar{p}(s + \bar{s}H)((qE_0 + \bar{q}(u + \bar{u}H))(r + \bar{r}(t + \bar{t}H)E_0))^\infty$$

To indicate the closing of the loop, we write

$$1 = R[p((r + \bar{r}(t + \bar{t}H)E_0)(qE_0 + \bar{q}(u + \bar{u}H)))^\infty + \bar{p}(s + \bar{s}H)((qE_0 + \bar{q}(u + \bar{u}H))(r + \bar{r}(t + \bar{t}H)E_0))^\infty],$$

where $R[\alpha]$ is interpreted in the generalized WHILE programming language as " REPEAT $\alpha$", with '$E_0$' interpreted as the exit and the return to the beginning of the 'REPEAT'.

The program of the expression for '1' takes the form given laterwith the interpretation given in the next remark.

<u>Remark</u>. In the generalized WHILE programming language, we have the following correspondence between the expression and the language.

$(\alpha)^\infty$ means the same as 'WHILE TRUE DO $\alpha$';

$R[\alpha]$ means the same as 'REPEAT $\alpha$';

'NULL' means that no operation is carried out.

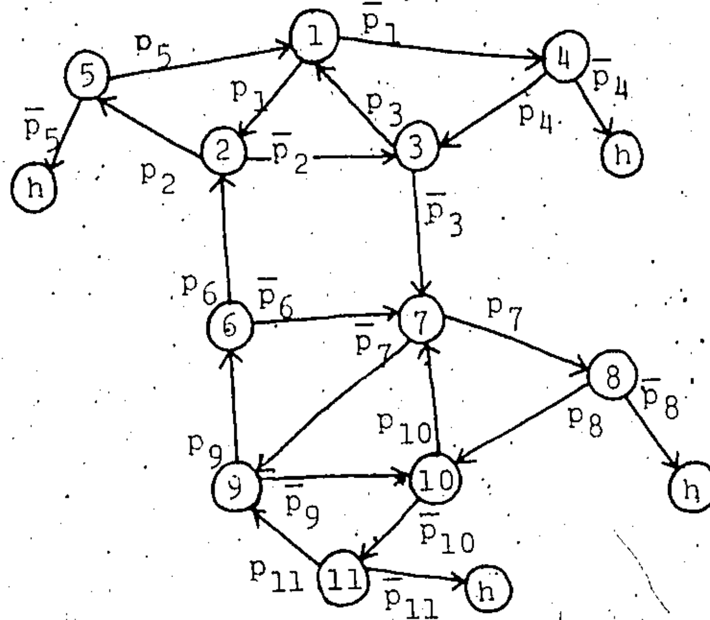
We now give the program in the generalized WHILE programming language.

```
REPEAT [IF p THEN WHILE TRUE DO [IF r THEN NULL
                                         ELSE IF t THEN NULL
                                                   ELSE HALT
                                             EXIT(0)
                                  IF q THEN EXIT (0)
                                         ELSE IF u THEN NULL
                                                   ELSE HALT
                                  ]
              ELSE IF s THEN NULL
                        ELSE HALT
                   WHILE TRUE DO [IF q THEN EXIT(0)
                                         ELSE IF u THEN NULL
                                                   ELSE HALT
                                  IF r THEN NULL
                                         ELSE IF t THEN NULL
                                                   ELSE HALT
                                             EXIT(0)
                                 ]
       ]
```

<u>Note</u>. The statement 'WHILE TRUE DO α' is to be repeated until we reach the possible exits, such as 'HALT' and 'EXIT(0)'.

In order to give the whole picture about the generalized WHILE programming language, we give the next example to illustrate the nested REPEAT-EXIT structure.

Example 5.3. In this example we illustrate the more involved concept of REPEAT-EXIT to an n-th level, for n = 0, 1, 2, ....




The set S of equations describing the program is as follows.

$$1 = p_1 2 + \bar{p}_1 4$$
$$2 = p_2 5 + \bar{p}_2 3$$
$$3 = p_3 1 + \bar{p}_3 7.$$
$$4 = p_4 3 + \bar{p}_4 h.$$
$$5 = p_5 1 + \bar{p}_5 h$$
$$6 = p_6 2 + \bar{p}_6 7$$
$$7 = p_7 8 + \bar{p}_7 9$$
$$8 = p_8 10 + \bar{p}_8 h$$
$$9 = p_9 6 + \bar{p}_9 10$$
$$10 = p_{10} 7 + \bar{p}_{10} 11.$$
$$11 = p_{11} 9 + \bar{p}_{11} h$$

None of the rules of the algorithm given in section 4.3 for translating programs into WHILE form can be applied. We replace

the h's by H's, and simplify by the fact that 'H' is a left zero. Thus we have,

$$4 = (p_4 + \bar{p}_4 H)3$$

$$5 = (p_5 + \bar{p}_5 H)1$$

$$8 = (p_8 + \bar{p}_8 H)10$$

$$11 = (p_{11} + \bar{p}_{11} H)9$$

The above can be eliminated by substitution into the rest of the equations. We now have the following as the set of equations to be further manipulated.

$$1 = p_1 2 + \bar{p}_1 (p_4 + \bar{p}_4 H)3$$

$$2 = p_2 (p_5 + \bar{p}_5 H)1 + \bar{p}_2 3$$

$$3 = p_3 1 + \bar{p}_3 7$$

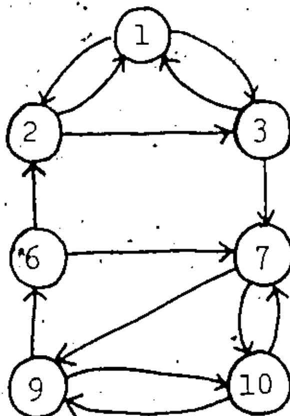
$$6 = p_6 2 + \bar{p}_6 7$$

$$7 = p_7 (p_8 + \bar{p}_8 H)10 + \bar{p}_7 9$$

$$9 = p_9 6 + \bar{p}_9 10$$

$$10 = p_{10} 7 + \bar{p}_{10} (p_{11} + \bar{p}_{11} H)9$$

The transition of states is shown by the following simplified diagram.




There are paths from the initial state, 1, to itself, we replace all the occurrence of 1 on the right-hand sides of the equations

by $E_0$ which is a left zero and is interpreted as the exit and the return to the initial state.

Then,

$$2 = p_2(p_5 + \bar{p}_5H)E_0 + \bar{p}_2 3$$

or $$2 = (p_2(p_5 + \bar{p}_5H)E_0 + \bar{p}_2)3$$

and $$3 = p_3E_0 + \bar{p}_3 7$$

or $$3 = (p_3E_0 + \bar{p}_3)7$$

Thus, $$2 = (p_2(p_5 + \bar{p}_5H)E_0 + \bar{p}_2)(p_3E_0 + \bar{p}_3)7$$

By substitution again, we reduce the number of equations by two as the above two equations are eliminated by substitution.

$$1 = p_1(p_2(p_5 + \bar{p}_5H)E_0 + \bar{p}_2)3 + \bar{p}_1(p_4 + \bar{p}_4H)(p_3E_0 + \bar{p}_3)7$$

or $$1 = p_1(p_2(p_5 + \bar{p}_5H)E_0 + \bar{p}_2)(p_3E_0 + \bar{p}_3)7 + \bar{p}_1(p_4 + \bar{p}_4H) .(p_3E_0 + \bar{p}_3)7$$

or $$1 = (p_1(p_2(p_5 + \bar{p}_5H)E_0 + \bar{p}_2)(p_3E_0 + \bar{p}_3) + \bar{p}_1(p_4 + \bar{p}_4H) .(p_3E_0 + \bar{p}_3))7$$

and $$6 = p_6(p_2(p_5 + \bar{p}_5H)E_0 + \bar{p}_2)(p_3E_0 + \bar{p}_3)7 + \bar{p}_6 7$$

or $$6 = (p_6(p_2(p_5 + \bar{p}_5H)E_0 + \bar{p}_2)(p_3E_0 + \bar{p}_3) + \bar{p}_6)7.$$

Thus, the set of equations becomes,

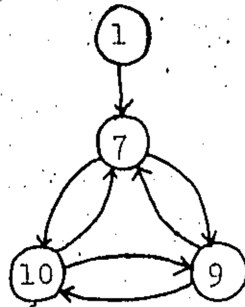
$$1 = (p_1(p_2(p_5 + \bar{p}_5H)E_0 + \bar{p}_2)(p_3E_0 + \bar{p}_3) + \bar{p}_1(p_4 + \bar{p}_4H) .(p_3E_0 + \bar{p}_3))7$$

$$7 = p_7(p_8 + \bar{p}_8H)10 + \bar{p}_7 9$$

$$9 = p_9[p_6(p_2(p_5 + \bar{p}_5H)E_0 + \bar{p}_2)(p_3E_0 + \bar{p}_3) + \bar{p}_6]7 + \bar{p}_9 10$$

$$10 = p_{10}7 + \bar{p}_{10}(p_{11} + \bar{p}_{11}H)9$$

The diagram form of the above is :




Since there is no path from the initial state, 1, to itself, we take its successor-state, 7, as the new initial state of the subprogram and repeat again the process of checking whether there is path from the initial state to itself.

We replace all the occurrence of $E_0$'s by $E_1$'s which are interpreted as the exit and the return to the previous initial state, and replace all the occurrence of the present initial state, 7, by $E_0$ on the right-hand-sides of all the equations.

$$9 = p_9[p_6(p_2(p_5 + \bar{p}_5 H)E_1 + \bar{p}_2)(p_3E_1 + \bar{p}_3) + \bar{p}_6]E_0 + \bar{p}_9 10$$

or $$9 = (p_9[p_6(p_2(p_5 + \bar{p}_5 H)E_1 + \bar{p}_2)(p_3E_1 + \bar{p}_3) + \bar{p}_6]E_0 + \bar{p}_9)10$$

and $$10 = p_{10}E_0 + \bar{p}_{10}(p_{11} + \bar{p}_{11}H)9$$

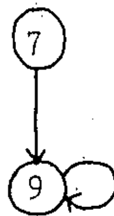
or $$10 = (p_{10}E_0 + \bar{p}_{10}(p_{11} + \bar{p}_{11}H))9 .$$

Thus, we have,

$$9 = (p_9[p_6(p_2(p_5 + \bar{p}_5 H)E_1 + \bar{p}_2)(p_3E_1 + \bar{p}_3) + \bar{p}_6]E_0 + \bar{p}_9)(p_{10}E_0 + \bar{p}_{10}(p_{11} + \bar{p}_{11}H))9$$

$$7 = (p_7(p_8 + \bar{p}_8 H)[p_{10}E_0 + \bar{p}_{10}(p_{11} + \bar{p}_{11}H)] + \bar{p}_7)9$$

with the simplified state diagram :



Now, since there is no path from 7 to itself, we take 9 as the new initial state. Further, there is a path from 9 to itself, and the equation is of the form $q = \alpha q$. We therefore obtain $q = (\ \alpha\ )^{\infty}$ as the result.

Hence, we get what we called a generalized regular expression representing the given program.

First,

$$9 = [(p_9[p_6(p_2(p_5 + \overline{p}_5H)E_1 + \overline{p}_2)(p_3E_1 + \overline{p}_3) + \overline{p}_6]E_0 + \overline{p}_9) .(p_{10}E_0 + \overline{p}_{10}(p_{11} + \overline{p}_{11}H))]^{\infty}$$

Next,

$$7 = R((p_7(p_8 + \overline{p}_8H)[p_{10}E_0 + \overline{p}_{10}(p_{11} + \overline{p}_{11}H)] + \overline{p}_7) .[(p_9[p_6(p_2(p_5 + \overline{p}_5H)E_1 + \overline{p}_2)(p_3E_1 + \overline{p}_3) + \overline{p}_6]E_0 + \overline{p}_9)(p_{10}E_0 + \overline{p}_{10}(p_{11} + \overline{p}_{11}H))]^{\infty})$$

Thus,

$$1 = R[p_1(p_2(p_5 + \overline{p}_5H)E_0 + \overline{p}_2)(p_3E_0 + \overline{p}_3) + \overline{p}_1(p_4 + \overline{p}_4H) .(p_3E_0 + \overline{p}_3))R[(p_7(p_8 + \overline{p}_8H)[p_{10}E_0 + \overline{p}_{10}(p_{11} + \overline{p}_{11}H)] + \overline{p}_7)[(p_9[p_6(p_2(p_5 + \overline{p}_5H)E_1 + \overline{p}_2)(p_3E_1 + \overline{p}_3) + \overline{p}_6]E_0 + \overline{p}_9)(p_{10}E_0 + \overline{p}_{10}(p_{11} + \overline{p}_{11}H))]^{\infty}]]$$

Let us express the above in the form of programming language of the generalized WHILE form with REPEAT-EXIT. We need the following to give the correspondence between the expression and the programming language.

$a$ ⟷ DO a

$p\alpha + \overline{p}\beta$ ⟷ IF p THEN [DO α] ELSE [DO β]

$(p\alpha)^*\overline{p}$ ⟷ WHILE p DO[α]

$\alpha\beta \longleftrightarrow$ DO $\alpha$, DO $\beta$

$H \longleftrightarrow$ HALT

$E_i \longleftrightarrow$ EXIT (i)

$(\alpha)^{\infty} \longleftrightarrow$ WHILE TRUE DO[$\alpha$]

$R[\alpha] \longleftrightarrow$ REPEAT[$\alpha$]

$\Lambda \longleftrightarrow$ NULL (meaning 'NO OPERATION')

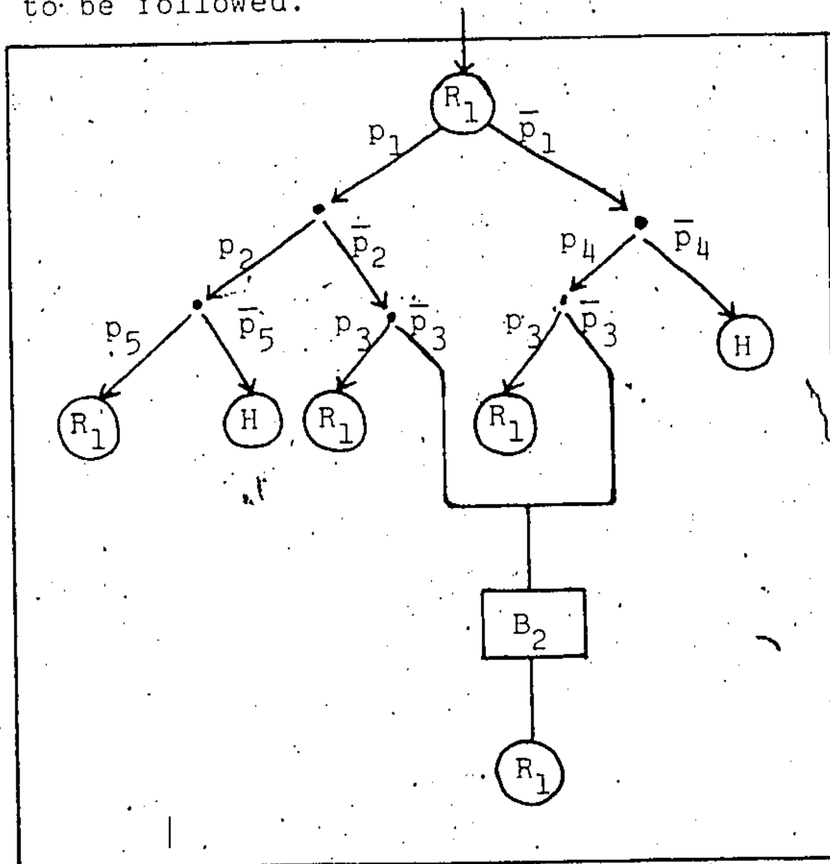
The program is then of the following form :

```
REPEAT[IF p1 THEN IF p2 THEN IF p5 THEN NULL
      |                             ELSE HALT
      |                     EXIT (0)
      |                 ELSE NULL
      |          IF p3 THEN EXIT (0)
      |                ELSE NULL
      |      ELSE IF p4 THEN NULL
      |                 ELSE HALT
      |           IF p3 THEN EXIT (0)
      |                 ELSE NULL
      | REPEAT[IF p7 THEN IF p8 THEN NULL
      |       |                  ELSE HALT
      |       |           IF p10 THEN EXIT (0)
      |       |                  ELSE IF p11 THEN NULL
      |       |                              ELSE HALT
      |       |       ELSE NULL
      |       |[WHILE TRUE DO IF p9 THEN IF p6 THEN IF p2 THEN IF p5 THEN NULL
                                                                    ELSE HALT
```

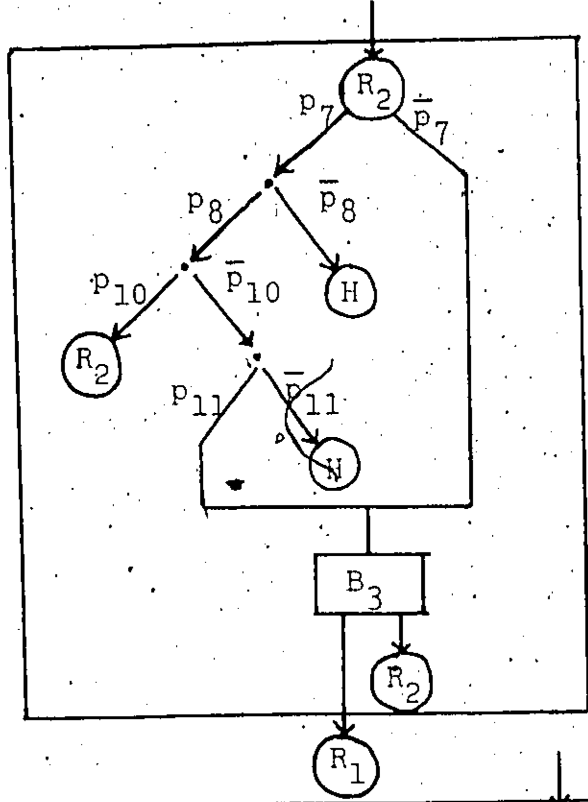
```
                                  EXIT (1)
                              ELSE NULL
                        IF p3 THEN EXIT (1)
                              ELSE NULL
                   ELSE NULL
              EXIT (0)
          ELSE NULL
     IF p10 THEN EXIT (0)
          ELSE IF p11 THEN NULL
                     ELSE HALT .
```

We now give the block-diagram of the above program with '$R_1$' interpreted as the return to the node named $R_1$, and for the other symbols the interpretations are as given in the remarks to be followed.

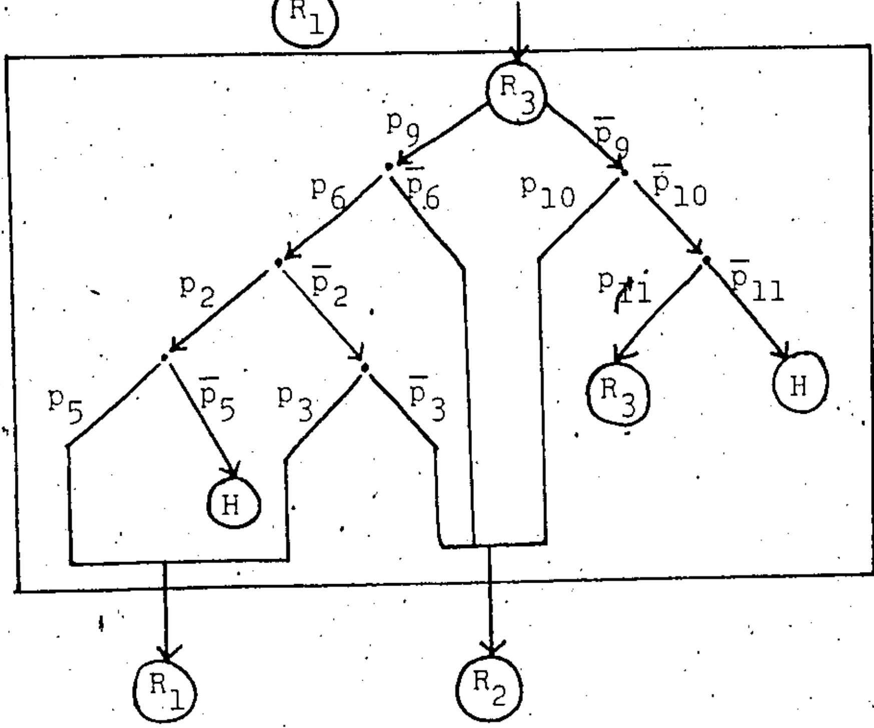
$B_1$ :



$B_2$ :




$B_3$ :




Note. The only looping allowed is the return to the entry of each block, and each block has only one entry.

Remarks. (1) The 'HALT' means absolute halt; that is, the execution of the whole program stops.

(2) The 'EXIT (i)' is the exit from the program and the return to the next i-th 'REPEAT' by going from the inner to the outer.

(3) The 'NULL' means no operation is to be carried out.

## 5.2 Algorithm for Translating Programs into Generalized WHILE Form with REPEAT-EXIT.

We give first, the definitions needed for the generalized WHILE algorithm which translates any given program to a generalized WHILE program with REPEAT-EXIT, given in the previous section.

Definition 5.1. The set $G(\Sigma)$ of generalized regular expressions over the alphabet $\Sigma$ is defined to be as follows :

(i) $\emptyset, \Lambda \in G(\Sigma)$

$H, E_0, E_1, E_2, \ldots\ldots \in G(\Sigma)$

$\Sigma \subset G(\Sigma)$

(ii) If $\alpha, \beta \in G(\Sigma)$, then

$\alpha + \beta$

$\alpha\beta$

$(\alpha)^*$

$(\alpha)^\infty$

$R[\alpha]$

are all in $G(\Sigma)$.

<u>Definition 5.2</u>. Let $\overline{\Sigma}^*$ denote the set, $\Sigma^* \cup \Sigma^*H \cup \Sigma^*E_0 \cup \Sigma^*E_1 \cup \ldots$ with multiplication given by concatenation and simplification by the relations :

$$Hx = H, \quad \text{for all } x \in \overline{\Sigma}^*$$

$$E_i x = E_i, \text{ for all } x \in \overline{\Sigma}^* \text{ and } i = 0, 1, 2, \ldots\ldots$$

Then $\overline{\Sigma}^*$ is the monoid $\Sigma^*$ with left zeros $H, E_0, E_1, \ldots,$ adjoined.

<u>Definition 5.3</u>. Let us define $|\ | : G(\Sigma) \longrightarrow 2^{\overline{\Sigma}^*}$ in the following way :

(i) $|\emptyset| = \emptyset, \quad |\Lambda| = \{\Lambda\};$

$|H| = \{H\}, \quad |E_i| = \{E_i\}, \quad$ for $i = 0, 1, 2, 3, \ldots\ldots$

$|a| = \{a\}, \quad$ for all $a \in \Sigma$.

(ii) $|\alpha + \beta| = |\alpha| \cup |\beta|;$

$|\alpha\beta| = |\alpha||\beta|$

$\underline{\underline{\text{def}}} \{ xy \in \overline{\Sigma}^* \mid x \in |\alpha|, y \in |\beta| \};$

$|(\alpha)^*| = (|\alpha|)^*$

$\underline{\underline{\text{def}}} \{\Lambda\} \cup |\alpha| \cup |\alpha||\alpha| \cup \ldots\ldots\ldots$

$|(\alpha)^\infty| = (|\alpha|)^\infty$

$\underline{\underline{\text{def}}} (|\alpha| \cap \Sigma^*)^*(|\alpha| \cap (\Sigma^*H \cup \Sigma^*E_0 \cup \Sigma^*E_1 \cup \ldots))$

$|R[\alpha]| = R[|\alpha|]$

$= (\Delta|\alpha|)^\infty$

where $\Delta X = (X \cap \Sigma^*H) \cup \{ xE_i \mid xE_{i+1} \in X \}$.

<u>Example 5.4</u>. Let us consider the following program in the generalized WHILE form with REPEAT-EXIT. The generalized regular expression of the program is :

$$R[aR[(pE_1 + \overline{p}b)(qH + \overline{q}E_0)]].$$

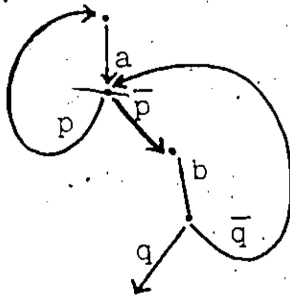
The program takes the following form :

```
REPEAT DO a
        REPEAT IF p THEN EXIT (1)
                    ELSE DO b
                IF q THEN HALT
                    ELSE EXIT (0)
```

By the above definition let us find the denotation of the regular expression of this program.

$$
\begin{aligned}
&|R[aR[(pE_1 + \bar{p}b)(qH + \bar{q}E_0)]]| \\
&= |R[a(\Delta(pE_1 + \bar{p}b)(qH + \bar{Q}E_0))^{\infty}]| \\
&= |R[a((pE_0 + \bar{p}b)(qH + \bar{q}))^{\infty}]| \\
&= |R[a(\bar{p}b\bar{q})^*(pE_0 + \bar{p}bqH)]| \\
&= |(\Delta a(\bar{p}b\bar{q})^*(pE_0 + \bar{p}bqH))^{\infty}| \\
&= |(a(\bar{p}b\bar{q})^*(p + \bar{p}bqH))| \\
&= |(a(\bar{p}b\bar{q})^*p)^*a(\bar{p}b\bar{q})^*\bar{p}bqH| .
\end{aligned}
$$

Let us have the diagram form of the above program.




Note that the execution sequence of the generalized WHILE program with REPEAT-EXIT agrees with the usual way of interpretation.

In the rest of this section, we give the algorithm for translating any program into the generalized WHILE with REPEAT -EXIT. Then a characterization theorem is proved.

The algorithm consists of the following sub-algorithms in the order as appears. The first part of the algorithm covers the algorithm given in section 4.3.

Generalized WHILE Algorithm with REPEAT-EXIT.

--STRUCTURED PROGRAM ALGORITHM : -

Data: STATES : a finite set,

INITIAL : an element of STATES,

EQUATIONS : a set of equations, one for each q in STATES, of the following form :

$q = ar$, where $r \in$ STATES $\cup \{h\}$, $r \neq q$, $a \in A$;

or $q = pr + \overline{p}s$, with $r, s \in$ STATES $\cup \{h\}$, $p \in P$, and $q, r, s$ are distinct.

WHILE Rules:

1. substitute $q = \alpha r$, for $\alpha \in W_0$;
2. $p\alpha q + \overline{p}\beta q = (p\alpha + \overline{p}\beta)q$, for $\alpha, \beta \in W_0$;
3. $q = p\alpha q + \overline{p}\beta r$, then $q = (p\alpha)^*\overline{p}\beta r$, for $\alpha, \beta \in W_0$;
4. $p\alpha q + \overline{p}\beta r = \overline{p}\beta r + p\alpha q$, for $\alpha, \beta \in W_0$.

Use WHILE rules until one of the following cases holds :

Case (i), INITIAL $= \alpha h$, with $\alpha \in W_0$.

Case (ii), for some set $Q_1 \subset$ STATES, there is, for each $q \in Q_1$, an equation : $q = p\alpha r + \overline{p}\beta s$, with $r, s \in Q_1 \cup \{h\}$, $\alpha, \beta \in W_0 \cup \{\Lambda\}$, and $q, r, s$ are distinct; and

either (a) INITIAL $= \alpha r$, $\alpha \in W_0$, $r \in Q_1$,

or (b) INITIAL $= p\alpha r + \overline{p}\beta s$, with $r, s \in Q_1 \cup \{h\}$, $\alpha, \beta \in W_0 \cup \{\Lambda\}$, and $q, r, s$ are distinct.

If, we have case (i), then the output is $\alpha H$. [A]

if, we have case (ii), then replace every occurrence of 'h' on the right-hand side of an equation by 'H'. [B]

if (a), call the Extended WHILE Algorithm, with

STATES = $Q_1$

INITIAL = r

EQUATIONS : the set of equations for each $q \in Q_1$,

if $r = \beta$ is obtained, then the output is $\alpha\beta$. [C]

if (b), call the Extended WHILE Algorithm, with

first,

STATES = $Q_1$

INITIAL = r

EQUATIONS : the set of equations for $Q_1$,

next,

STATES = $Q_1$

INITIAL = s

EQUATIONS : the set of equations for $Q_1$,

if $r = \gamma$ and $s = \delta$ are obtained, then the output is $p\alpha\gamma + \bar{p}\beta\delta$. [D]

--<u>EXTENDED WHILE ALGORITHM</u>. :-

<u>Data</u>: STATES : a finite set,

INITIAL : an element of STATES,

EQUATIONS : a set of equations, one for each $q \in$ STATES, of the following form:

$q = \alpha r$, for $r \in$ STATES, $r \neq q$ and $|\alpha| \cap \Sigma^* \neq \emptyset$

or $q = p\alpha r + \bar{p}\beta s$, with r, s $\in$ STATES, q, r, s are distinct and $|\alpha| \cap \Sigma^* \neq \emptyset$, $|\beta| \cap \Sigma^* \neq \emptyset$.

Extended WHILE Rules:

1. substitute $q = \alpha r$, with $|\alpha| \cap \Sigma^* \neq \emptyset$;
2. $p\alpha q + \bar{p}\beta q = (p\alpha + \bar{p}\beta)q$, with $|\alpha| \cap \Sigma^* \neq \emptyset$, $|\beta| \cap \Sigma^* \neq \emptyset$;
3. $q = p\alpha q + \bar{p}\beta r$, then $q = (p\alpha)^*\bar{p}\beta r$, with $|\alpha| \cap \Sigma^* \neq \emptyset$ and $|\beta| \cap \Sigma^* \neq \emptyset$;
4. $p\alpha q + \bar{p}\beta r = \bar{p}\beta r + p\alpha q$, with $|\alpha| \cap \Sigma^* \neq \emptyset$, $|\beta| \cap \Sigma^* \neq \emptyset$;
5. $q = p\alpha r + \bar{p}\beta$, then $q = (p\alpha + \bar{p}\beta)r$, with $|\alpha| \cap \Sigma^* \neq \emptyset$ and $|\beta| \cap \Sigma^* = \emptyset$;
6. $q = \alpha q$, then $q = (\alpha)^\infty$, with $|\alpha| \cap \Sigma^* \neq \emptyset$.
   (Note: $|\alpha^\infty| \cap \Sigma^* = \emptyset$.)

Use the Extended WHILE Rules until one of the following cases:

Case (i), INITIAL $= \beta$, with $|\beta| \cap \Sigma^* = \emptyset$.

Case (ii), for some $Q_1 \subset$ STATES, there is, for each $q \in Q$,
an equation : $q = p\alpha r + \bar{p}\beta s$.
with $r, s \in Q_1$, $|\alpha| \cap \Sigma^* \neq \emptyset$, $|\beta| \cap \Sigma^* \neq \emptyset$, and
$q, r, s$ are distinct, and
either (a) INITIAL $= \alpha r$, $|\alpha| \cap \Sigma^* \neq \emptyset$, $r \in Q_1$,
or (b) INITIAL $= p\alpha r + \bar{p}\beta s$, with $r, s \in Q_1$,
$q, r, s$ are distinct and $|\alpha| \cap \Sigma^* \neq \emptyset$,
$|\beta| \cap \Sigma^* \neq \emptyset$.

If we have case (i), then return with $\beta$ as the output. [E]

If we have case (ii), then

if (a), call the REPEAT-EXIT Algorithm, with

STATES $= Q_1$

INITIAL $= r$

EQUATIONS : the set of equations for $Q_1$.

if we obtain $r = \beta$, then return with $\alpha\beta$. [F]

a

if (b), call the REPEAT-EXIT Algorithm, with

first,

STATES = $Q_1$

INITIAL = r

EQUATIONS : the set of equations for $Q_1$.

next,

STATES = $Q_1$

INITIAL = s

EQUATIONS : the set of equations for $Q_1$.

if $r = \gamma$ and $s = \delta$ are obtained, then return with $p\alpha\gamma + \bar{p}\beta\delta$, as the output. [G]

--<u>REPEAT-EXIT ALGORITHM</u> :-

<u>Date</u>: STATES : a finite set,

INITIAL : an element of STATES,

EQUATIONS : a set of equations, one for each q in STATES, of the following form, for some values of n,

$$q \doteq p\alpha r + \bar{p}\beta s,$$

with r, s $\varepsilon$ STATES, q, r, s are distinct and $|\alpha| \cap \Sigma^* \neq \emptyset$, $|\beta| \cap \Sigma^* \neq \emptyset$.

If there is no path from the INITIAL to itself, and we have INITIAL = $p\alpha r + \bar{p}\beta s$, then let $Q_1$ = STATES - {INITIAL}, and call the REPEAT-EXIT Algorithm, for

first,

STATES = $Q_1$

INITIAL = r

EQUATIONS : the set of equations for $Q_1$

next,

STATES = $Q_1$

INITIAL = s

EQUATIONS : the set of equations for $Q_1$.

If we have r = $\gamma$ and s = $\delta$ as results, we return with $p\alpha\gamma + \bar{p}\beta\delta$ as the output. [H]

Otherwise, if there is a path from the INITIAL to itself, we replace every occurrence of '$E_i$' in all the equations by '$E_{i+1}$', and then replace every occurrence of the INITIAL on the right-hand-sides of all the equations by $E_0$. Then call the Extended WHILE Algorithm, with

STATES = STATES

INITIAL = INITIAL

EQUATIONS : the set of equations as modified.

If we have INITIAL = $\alpha$ as the result, then return with R[$\alpha$] as the output. [I]

Examples. The examples given in the previous section were solved by the Generalized WHILE Algorithm with REPEAT-EXIT as decribed in the above.

Next, we want to show that the Generalized WHILE Algorithm actually works for any given program. We firstprove the following lemma.

Lemma 5.1. Let $A \subset \bar{\Sigma}^*$ with $\Lambda \not\in A$. Then the only non-empty solution of the equation $X = AX$ is $X = A^\infty$.

Proof. Let $x \in X$. Thus $x = a_1 x_1$, for some $a_1 \in A$, $x_1 \in X$. Then, for the same reason, $x_1 = a_2 x_2$, for some $a_2 \in A$, $x_2 \in X$; ....... Thus, $x = a_1 a_2 \ldots\ldots a_i$, where $i$ is the smallest integer such that $a_i \not\in \Sigma^*$. Hence, $x \in A^\infty$, so $X \subset A^\infty$.

Conversely, if $b_1, \ldots\ldots, b_{n-1} \in A \cap \Sigma^*$ and $b_n \in A - \Sigma^*$, then since $X \neq \emptyset$, let $x_0 \in X$, then we have

$$b_1 b_2 \ldots\ldots b_n x_0 = b_1 b_2 \ldots\ldots b_n \in X,$$

thus $A^\infty \subset X$, completing the proof.

Theorem 5.1. For any program $\pi$, the generalized structured program algorithm produces a solution $q_0 = \alpha$ such that $|\alpha| = |\mathcal{A}_\pi| H$.

Proof. If the algorithm stops at [A], then $|\alpha| = |\mathcal{A}_\pi|$, by Theorem 4.4 in section 4.3, so $|\alpha H| = |\mathcal{A}_\pi| H$. If not, the solution for INITIAL of the EQUATIONS at point [B] is $|\mathcal{A}_\pi| H$.

If the algorithm stops at [C] or [D], then the result is correct assuming that the extended WHILE program works. To see that this is the case, argue by induction as follows.

If the extended WHILE algorithm stops at point [E], then the result is obtained by rules (1) to (6) from the data EQUATIONS. Rules (1) to (4) preserve equality, as we have seen. Rule (5) preserves equality, since for any $X, Y \subset \overline{\Sigma}^*$, if $X = pAY + \bar{p}B$, where $|B| \cap \Sigma^* = \emptyset$, then $X = (pA + \bar{p}B)Y$ because $BY = B$. Rule (6) preserves equality by Lemma 5.1. Thus the expression $\beta$ obtained by rules (1) to (6) denotes the solution of EQUATIONS for the INITIAL.

If the extended WHILE algorithm stops at points [F] or [G], then the result is correct assuming that the REPEAT-EXIT algorithm works. What remains to be checked is the REPEAT-EXIT algorithm. If the REPEAT-EXIT algorithm stops at point [H], then we may always obtain the result by assuming calls to the REPEAT-EXIT algorithm with a smaller set of STATES give proper results. If the algorithm stops at point [I], then assuming extended WHILE algorithm returns a solution $\alpha$ of the modified equations, then $|R[\alpha]| = (\Delta|\alpha|)^{\infty}$ is a solution of the equations before modification. This is justified since the modification of the equations consists of increasing the level of the nested REPEAT-EXIT structure by one.

Finally, the algorithm always terminates, as follows: The extended WHILE program calls only the REPEAT-EXIT program. The REPEAT-EXIT program calls either itself with fewer states as data, or the extended WHILE program with fewer loops in the data, or the extended WHILE input program. Thus, the sequence of calls to the extended WHILE program has as data programs with fewer and fewer loops. Since the extended WHILE program always solves programs with no loops, eventually this process must terminate.

# BIBLIOGRAPHY

[1] Ashcroft, E. and Manna, Z., The Translation of GO TO Programs to WHILE Programs, Proc. IFIP Cong., North-Holland Pub. Com., Amsterdam, (1971), 250-255.

[2] Böhm, C. and Jacopini, G., Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, Comm. ACM, 9 No. 5, (1966).

[3] Cooper, D. C., Böhm and Jacopini Reduction of Flow Charts, Letter to the Editor, Comm. ACM, 10, NO. 8, (1967).

[4] Dijkstra, E. W., GO TO Statement Considered Harmful, Comm. ACM, 11, March, (1968), 147-148.

[5] Dijkstra, E. W., Notes on Structured Programming, in 'Structured Programming' by Dahl, O. S. and Dijkstra, E. W. and Hoare, C. A. R., Academic Press, New York, (1972).

[6] Eilenberg, S., Automata, Languages and Machines, Vol. A, Academic Press, New York, (1974).

[7] Elspas, B.; Levitt, K. N.; Waldinger, R. J. and Waksman, A., An Assessment of Techniques for Proving Program Correctness, ACM Computing Surveys, Vol. 4, No. 2, June, (1972), 97-147.

[8] Gries, D., On Structured Programming, ACM Forum, Vol. 17, No. 11, Nov., (1974), 655-657.

[9] Hoare, C. A. R., An Axiomatic Basis for Computer Programming, Comm. ACM, Vol. 12, No. 10, Oct., (1969), 576-581.

[10] Knuth, D. E. and Floyd, R. W., Notes on Avoiding GO TO Statements, Information Processing Letters, 1, North-Holland Publishing Company, (1971), 23-31.

[11] Kosaraju, S. R., Analysis of Structured Programs, Journal of Computer and System Science, No. 9, (1974), 232-255.

[12] Manna, Z., Mathematical Theory of Computation, McGraw-Hill Inc., (1974).

[13] Wirth, N., Systematic Programming -- An Introduction, Prentice-Hall Inc., (1973).

[14] Wulf, W. A., Programming Without the GO TO, Information Processing 71, North-Holland Publishing Company, (1972)

[15] Kasai, T., Translatability of Flowcharts into WHILE Programs, Journal of Computer and System Science, No. 9, (1974), 177-195.