ON SELF-MODIFYING PROGRAMS

by

Raymond Reiter

Department of Computer Science
University of British Columbia
Vancouver 8, British Columbia
Canada

May, 1972

ABSTRACT

A model for self-modifying program schemata is presented. Essentially this augments Ianov program schemata with certain self-modifying primitives. An appropriate algorithm for the decomposition of a directed graph is developed and applied to yield a Ianov schema equivalent to a given self-modifying schema. Generalizations of the model are considered. In particular, certain growing programs are seen to lie outside the class of Ianov schemata.

On Self-Modifying Programs

0. Introduction

The usual models of algorithmic processes in the theory of computation (e.g. Turing Machines, partial recursive functions, etc.) deal with fixed structures by means of which a computation is effected. Similarly, more realistic models of programs (e.g. program schemata [3,5,7]) deal with fixed structure flowcharts which characterize the flow of control in the execution of an algorithm. However, all assembly languages, as well as certain high level languages in interpretive mode (e.g. LISP) admit programs which dynamically modify themselves. Curiously enough there has been very little research reported on a theory of self-modifying programs. A very early reference to the problem may be found in [2] where, in their description of a flowcharting language, Goldstine and Von Neumann introduce a flowchart primitive, called a variable remote connection, which functions like the label variable of PL/1. In artificial intelligence circles there has been continual speculation over the role of program self-modification in learning, and in the attendant restructuring of intelligent systems. In [10] Winograd describes a system for understanding natural language which dynamically generates programs (in MICROPLANNER) for subsequent execution. The work on automatic program writing by Waldinger and Lee [9] and more recently, Manna and Waldinger [6], can also be viewed in this light.

Of course no form of self-modification can enlarge the class of computable functions. What does appear to be affected, in a fundamental way, is the nature of the control of a computation. Intuitively one feels that suitable self-modification primitives introduce control structures very different from the usual program loops, branches, and recursion.

Conceivably this additional descriptive power can lead to languages more suitable for describing large scale intelligent systems.

One approach to the study of control structures in programming languages is through the introduction of appropriate program schemata. In a schema, the control is formalised while the primitive computations and tests are left uninterpreted. Examples of this approach are: [4] for parallel control, [8] for recursive control and [3,5] for iterative control. Given such models, one can then investigate the relative "powers" of different control mechanisms. For example, it is known [8] that recursion is strictly more powerful than iteration.

This paper presents a simple model of program self-modification, embedded in the language of Ianov program schemata [3,7]. We ask whether this additional control structure yields a more powerful class of schemata than those of Ianov. Because our model is essentially finite state, the answer turns out to be no. However, by slightly generalizing the self-modification primitives so as to permit programs to grow, we can easily obtain programs lying outside the class of Ianov schemata.

Section 1 presents an algorithm which, under appropriate conditions, decomposes a node labelled directed graph in such a way that we can associate a unique state with each node of the resulting graph. This procedure is applied to the model for self-modifying schemata of Section 2 to prove Ianov closure. Section 3 briefly discusses generalizations of the model.

The reader is assumed to be familiar with the basic notions of Ianov schemata as described in, say [7].

1. Finite State Decomposition of Graphs

We consider finite directed graphs G such that

1. G has a distinguished initial node n_0.
2. Every node of G is accessible from n_0.
3. Each node of G is labelled. We denote the labelling of node n_i by λ_i.

A walk is a sequence of node labellings $W = \lambda_{i_1}, \lambda_{i_2}, \ldots, \lambda_{i_r}$ such that, for $1 \leq j < r$, $n_{i_{j+1}}$ is an immediate successor of n_{i_j}. n_{i_1} is the initial node of W, and n_{i_r} its terminal node. W is a walk to n_{i_r} if $n_{i_1} = n_0$. W will sometimes be called a walk from n_{i_1} when its terminal node is irrelevant to the discussion.

We assume that associated with G is a function σ which maps the set of all walks from n_0 into a finite set of states. σ must have the property

π: If W_1 and W_2 are walks to the same node m, if (m,n) is an edge of G, and if W is an arbitrary walk from n, then $\sigma(W_1) = \sigma(W_2) \Rightarrow \sigma(W_1W) = \sigma(W_2W)$.

Let G and G' be node labelled directed graphs, with distinguished nodes n_0 and n_0' respectively. G is a homomorphic image of G' if there exists a function

h: Node set of G' $\xrightarrow{\text{onto}}$ Node set of G

such that

1. $h(n_0')=n_0$
2. (n_i',n_j') is an edge of G' iff $(h(n_i'), h(n_j'))$ is an edge of G.
3. n' and h(n') have the same label.

The function h is a homomorphism.

It is clear that for every walk W from n_0 in G there is an equal walk from n_0' in G' and conversely. This remark is the basis for our applications of the algorithm of this section to flowcharts as in Section 2.

In this section we show how to construct, given G and a function σ

satisfying property π, a finite node labelled directed graph G' such that

1. G is a homomorphic image of G'.
2. For any pair of walks W_1' and W_2' to the same node n' of G', $\sigma(W_1')=\sigma(W_2')$

The Algorithm

Essentially, the algorithm does "node splitting" on G. A node is split whenever two walks to it define different states. Property π together with the existence of finitely many states guarantees termination.

We construct a sequence $G_0, G_1, \ldots, G_r, \ldots,$ of directed graphs, each of which contains copies of nodes of G. A copy in G_r of the node n of G will be denoted by n'. There may be several copies of the same node n of G, in which case they will all be ambiguously denoted by n'.

1. Let G_0 be the edge-free graph with single node n_0' labelled λ_0. This node is the distinguished initial node of G_r, for each r. Let $r=0$.
2. Suppose, for all nodes m' of G', that m' has as immediate successor a copy n' of a node n of G iff (m,n) is an edge of G. Then exit with $G'=G_r$.
3. If, in G_r, a node m' has no immediate successor n', while in G m has an immediate successor n, then:

(a) Suppose G_r has one or more, say k, occurrences of a node n' such that, in G, n is an immediate successor of m. Let $\Omega, \Omega_1, \ldots, \Omega_k$ be arbitrary walks, in G_r, to the node m' and the k nodes n' respectively. (Figure 1(a).) If there exists j such that $\sigma(\Omega\lambda_n)=\sigma(\Omega_j)$, then G_{r+1} is obtained by adding the edge (m',n') shown dotted in Figure 1(a).

(b) Otherwise, G_{r+1} is obtained from G_r by adding a new node n' labelled λ_n, together with an edge from m' to this new copy of n.

4. Set $r \leftarrow r+1$ and go to 2.

Lemma 1

Let p' be a given copy, in G_r, of a node p of G, and suppose W_1 and W_2 are walks to p'. Then $\sigma(W_1)=\sigma(W_2)$.

Proof:

Induction on r. For $r=0$ the result is immediate. Assume the result for all nodes of G_r, and consider G_{r+1}.

Case (a) G_{r+1} is obtained from G_r by 3(a). If all walks in G_{r+1} to p' exist also in G_r, the result follows by induction hypothesis. Otherwise, any walk W to p' in G_{r+1} which does not exist in G_r must "traverse" the new edge (m',n'), in which case there must be a walk V_1 from n' to p' which exists in both G_r and G_{r+1}. If $V_1=\lambda_n,\lambda_q,\ldots,\lambda_p$ let $V=\lambda_q,\ldots,\lambda_p$. Now, by construction, there also exists a walk Ω_j to n' which exists in both G_r and G_{r+1}. Hence $\Omega_j V$ is a walk to p' in both G_r and G_{r+1}. Then we can write

$W=U\lambda_n W_1\lambda_n W_2\lambda_n\ldots W_i\lambda_n V$

where U, W_1, W_2, ..., W_i are (not necessarily distinct) walks in both G_r and G_{r+1}. (See Figure 1(b). Each walk W_k has as initial node an immediate successor of n'.) Now by the induction hypothesis, $\sigma(U)=\sigma(\Omega)$ (See Figure 1(a)). Hence, by property π, $\sigma(W) = \sigma(\Omega\lambda_n W_1\lambda_n\ldots W_i\lambda_n V)$. By construction, $\sigma(\Omega\lambda_n)=\sigma(\Omega_j)$ so by property P, $\sigma(W)=\sigma(\Omega_j W_1\lambda_n\ldots W_i\lambda_n V)$. But $\Omega_j W_1$ is a walk in G_r, so by the induction hypothesis,

$\sigma(\Omega_j W_1)=\sigma(\Omega)$ whence, by property π, $\sigma(W)=\sigma(\Omega\lambda_n W_2\lambda_n \ldots W_i\lambda_n V)$, etc. Finally, we obtain $\sigma(W)=\sigma(\Omega\lambda_n V)=\sigma(\Omega_j V)$. But W was an arbitrary walk to p' in G_{r+1} which does not occur in G_r, so by the induction hypothesis, all walks to p' define the same state.

Case (b) G_{r+1} is obtained from G_r by 3(b). Then if p' is other than the node n' added to obtain G_{r+1}, the lemma holds by induction hypothesis, since no new walks to p' have been created. Otherwise, p' is n'. But then any two walks W_1 and W_2 to p' are of the form $W_1=U_1\lambda_n$ and $W_2=U_2\lambda_n$ where U_1 and U_2 are walks in G_r to m', so that $\sigma(U_1)=\sigma(U_2)$ by the induction hypothesis. By property π, $\sigma(W_1)=\sigma(W_2)$.

Lemma 2 (Termination)

Suppose, in G_r, there exist two distinct copies p' of a node p of G. If W_1 is a walk to the first copy and W_2 a walk to the second, then $\sigma(W_1)\neq\sigma(W_2)$.

Proof:

Obvious. By Lemma 1, all walks to a given node p' of G_{r+1} define the same state. By construction, if G_{r+1} is obtained from G_r by adding a new copy p', then at least one walk to this new copy in G_{r+1} must define a different state than that associated with any other copy p'.

Since there are only finitely many states possible, Lemma 2 guarantees termination of the algorithm. Since, in G, all nodes are accessible from n_0, the exit condition 2. of the algorithm assures us that the function h, defined by h(n')=n for all nodes n of G, is a homomorphism.

Theorem

The algorithm terminates with a graph G' such that G is a homomorphic image of G'. For any pair of walks W_1, W_2 to the same node n' of G', we have $\sigma(W_1)=\sigma(W_2)$. In G', the number of copies n' of a node n of G is at most $|\text{Range }(\sigma)|$.

2. Self-Modifying Schemata

In this section, we present a simple model of program self-modification. Essentially, the model augments Ianov program schemata [3,7] with operators which, upon execution, alter designated instructions of the schema. In addition, we provide for label variables which permit dynamically changing GOTO's.

Using the results of the previous section, we prove that any such self-modifying Ianov schema is equivalent to an ordinary schema.

Syntax

Our model of a self-modifying programming language has the following syntax:

```
<test variable> ::=p|q|r...
<operator>::=A|B|C...
<fixed label>::=L0|L1|L2...
<variable label>::=LV0|LV1|LV2...
<label>::=<fixed label>| <variable label>
<operation>::=<fixed label>do <operator>, goto <label>
<test>::=<fixed label>if <test variable> goto <label> else goto <label>
<label modifier>::=<variable label>← <fixed label>
<operator modifier>::=<fixed label>← <operator>| <fixed label>← <label modifier>|
                       <fixed label>← <operator modifier>
<instruction modifier>::=<fixed label>do <label modifier>, goto<label>| <fixed
                        label>do <operator modifier>, goto <label>
<instruction>::= <operation>| <test>| <instruction modifier>
```

A self-modifying program schema is any finite sequence of instructions, each instruction of which is labelled by a unique fixed label.

If an <instruction> is an <operation> or <instruction modifier> we shall call it a DO instruction. α is called the _scope_ of the DO instruction <fixed label> _do_α, _goto_ <label>. p is also called the scope of the test instruction <fixed label> _if_ p _goto_ <label> _else goto_ <label>. We shall often refer to an instruction with, say, <fixed label> =L, as instruction L.

Interpretation

As usual, the operators are uninterpreted. The test variables are uninterpreted Boolean variables. The role of fixed labels is clear. Variable labels are meant to bind fixed variables, and are initially unbound. An attempt to execute _goto_ <label> is undefined if label is an unbound variable label, or is bound to a fixed label with no associated instruction, or if it is a fixed label with no associated instruction. Otherwise, control is transferred to that instruction specified by <label> if <label> is fixed, or by its current binding if <label> is variable. With this condition in mind, if an instruction is an operation or a test it is interpreted as for Ianov schemata. If an instruction I is an instruction modifier there are two possibilities:

A. I is of the form

<fixed label> _do_ <label modifier>, _goto_ <label>

Suppose <label modifier> is LV$\leftarrow$L. Then upon execution of I, LV is bound to L. In addition, if <label> is LV, then the next instruction to be executed is L.

B. I is of the form

<fixed label> _do_ <operator modifier>, _goto_ <label>

Then <operator modifier> is L$\leftarrow\alpha$ where L is a fixed label and α is an operator, label modifier, or operator modifier. If L has no associated instruction in the program, or if L labels a test instruction, the result of executing I is undefined. Otherwise, L labels an instruction of the form

L <u>do</u> β, <u>goto</u> <label>

where β is an operator, label modifier, or operator modifier.

After execution of I, this instruction is replaced, in the program, by

L <u>do</u> α, <u>goto</u> <label>

and control is transferred to <label> of instruction I.

These informal remarks are now made precise.

We formally specify what it means to execute a self-modifying program P by defining three sequences:

(a) an <u>admissible label sequence</u> $L_{i_1}, L_{i_2}, \ldots, L_{i_r}$ of fixed labels of P

(b) an <u>execution sequence</u> $\tilde{L}_{i_1}, \tilde{L}_{i_2}, \ldots, \tilde{L}_{i_{r-1}}$

(c) a <u>state sequence</u> $S_0, S_1, \ldots, S_{r-1}$. If P has m variable labels LV1, LV2,...,LVm, and n DO instructions labelled L1,L2,...,Ln then each S_i is an m+n component <u>state</u> $(B_1,B_2,\ldots,B_m,C_1,C_2,\ldots,C_n)$.

These sequences must satisfy the following conditions:

1. L_{i_1} labels the first instruction of P. $S_0 = (\emptyset,\ldots,\emptyset,\alpha_1,\ldots,\alpha_n)$ where, for $1 \leq i \leq r$, α_i is the scope of the DO instruction L_i.

2. Suppose, for $1 \leq j \leq r-1$, that instruction L_{i_j} is

   L_{i_j} <u>do</u> α, <u>goto</u> <label>

   and $S_{j-1} = (B_1,\ldots,B_m,C_1,\ldots,C_n)$.

<u>Case (i)</u> C_{i_j} is an operator, say C_{i_j}=A. Then $S_j=S_{j-1}$ and $\tilde{L}_{i_j}$=A. If <label> is fixed, then $L_{i_{j+1}}$ =<label>. If <label> is variable, say <label> = LVp, then $B_p \neq \emptyset$, and $L_{i_{j+1}}=B_p$.

<u>Case (ii)</u> C_{i_j} is an instruction modifier, say C_{i_j}=Lk$\leftarrow\beta$. Then Lk is a DO instruction, $S_j=(B_1,\ldots,B_m,C_1,\ldots,C_{k-1},\beta,C_{k+1},\ldots,C_n)$, and $\tilde{L}_{i_j}$=Lk$\leftarrow\beta$. $L_{i_{j+1}}$ is determined as in Case (i).

<u>Case (iii)</u> C_{i_j} is a label modifier, say C_{i_j}=LVk←L. Then $S_j=(B_1,\ldots,B_{k-1},L,B_{k+1},\ldots,B_m,C_1,\ldots,C_n)$, and $\tilde{L}_{i_j}$=LVk←L. If <label> is fixed, $L_{i_{j+1}}$ = <label>. If <label> is variable, say <label>=LVp, then if p=k, $L_{i_{j+1}}$=L, while if p≠k, then $B_p \neq \emptyset$ and $L_{i_{j+1}}=B_p$.

3. Suppose, for $1 \leq j \leq r-1$, that instruction L_{i_j} is

L_{i_j} <u>if</u> p <u>goto</u> $\langle\text{label}\rangle_1$ <u>else goto</u> $\langle\text{label}\rangle_2$

and $S_{j-1}=(B_1,\ldots,B_m,C_1,\ldots,C_n)$. Then $S_j=S_{j-1}$ and one of the following two conditions holds.

(i) $L_{i_{j+1}}$ is determined from $\langle\text{label}\rangle_1$ as in 2. Case (i),and $\tilde{L}_{i_j}$=p.

(ii) $L_{i_{j+1}}$ is determined from $\langle\text{label}\rangle_2$ asin 2.Case (i), and $\tilde{L}_{i_j}=\bar{p}$.

An admissible label sequence specifies those instructions encountered in a legitimate execution of the program, where by "legitimate" we mean that branches on variable labels are taken according to their current bindings. A state indicates these current bindings (as $B_1,\ldots,B_m$) together with the current scopes of the DO instructions (as $C_1,\ldots,C_n$). $B_i=\emptyset$ indicates that LVi is currently unbound. An execution sequence is simply a history of the operators, test exits, and modifiers encountered in an execution of the program.

Given an execution sequence $\tilde{L}_{i_1},\ldots,\tilde{L}_{i_{r-1}}$, an <u>operator-test sequence</u> is obtained by deleting any term $\tilde{L}_{i_j}$ which is an operator modifier or label modifier. This sequence yields a history of the "real computation", namely those operators and test exits, in the order in which they are effected, during an execution of the program.

Two self-modifying programs are <u>strongly equivalent</u> if they have the same operator-test sequences. Notice that if two Ianov schemata are equivalent ([7]) they need not be strongly equivalent. However, strong equivalence implies equivalence, which is sufficient for our purposes.

Example 1

```
L0 do LV←L1, goto L3
L3 do A, goto L4
L4 if p goto L2 else goto LV
L2 do L1←C, goto L5
L1 do L3←LV←L2, goto L5
L5 do B, goto L6
L6 if q goto L3 else goto L7
L7 do H, goto L7
```

Two admissible label sequences are:

L0, L3, L4, L1, L5, L6, L3, L4, L2, L5, L6 and

L0, L3, L4, L2, L5, L6, L3, L4, L1, L5, L6 with corresponding operator-test sequences $A, \overline{p}, B, q, \overline{p}, B$ and

$A, \overline{p}, B, q, A, \overline{p}, C$

Representation as a Directed Graph

In order to utilize the results of Section 1, we represent a self-modifying program as a directed graph, and then define an appropriate notion of state on this graph. We then decompose this graph according to the algorithm of Section 1. Finally, the resulting graph is altered in a natural way to yield the flowchart of a Ianov schema strongly equivalent to the original program.

First, represent a program P by a directed graph G. G has a node, labelled Li, for each instruction Li of P. For each occurrence of goto Lj in P, say within instruction Li, G has an edge from Li to Lj. For each occurrence of a variable label LV in P, determine all of the possible bindings of LV. There can be only finitely many such bindings, say L1,...,Lk. For each occurrence of goto LV in P, say within an instruction Li, G has edges from Li to each of L1,...,Lk. (An attempt to execute <u>goto</u> LV will be undefined if LV is currently unbound, or if LV is bound to a fixed label with no associated instruction. We could accommodate this possibility by introducing an "undefined" node, joining to it the node Li. This approach introduces an inessential complexity to the analysis, which we choose to avoid by considering only well-formed programs for which LV is properly bound whenever <u>goto</u> LV is to be executed. Clearly, there is a procedure for testing a given program for well-formedness.) Finally, G has as distinguished initial node that node L0 which labels the first instruction of P. With no loss in generality, assume that all nodes of G are accessible from L0. Figure 2(a) is the graph G for the program of Example 1.

For each walk $W=L_{i_1},\ldots,L_{i_r}$ from L0 in G, define a state $\sigma(W)$ as follows:

1. If W is an admissible label sequence, then it has associated with it a state sequence $S_o,S_1,\ldots,S_{r-1}$. Define $\sigma(W)=S_{r-1}$.
2. If W is inadmissible, define $\sigma(W)=\Omega$.

It follows that there are only finitely many states, and σ satisfies property π of Section 1.

Now apply to G the algorithm of Section 1 to obtain a graph G' such that G is a homomorphic image of G'. In G', all walks to a given node define the same state so that we can associate a state with each node. Figure 2(b) is the graph G' for the program of Example 1. The dotted box Ω denotes a subgraph all of whose nodes have state Ω. States are indicated alongside nodes. If no state is indicated beside a node, its state is assumed to be that of its predecessor. For clarity, we have used an "abbreviated" state vector (B_1,C_1,C_2) where B_1 denotes the binding of LV, and C_1,C_2 denote the scopes of L1,L3 respectively.

Now, remove all nodes of G' whose associated state is Ω, together with those edges directed to them yielding a graph G''. By the construction of G, and the fact that G is a homomorphic image of G', every walk in G'' from its start node is an admissible label sequence of P, and conversely, every admissible label sequence is such a walk in G''. Now, replace each node label Li of G'' by the component C_i of its state, if Li is a DO instruction, or by the scope of instruction Li if Li is a test instruction. Furthermore, since in G'' all walks to a given node define the same state, we can safely delete all nodes labelled by an instruction or label modifier. The resulting graph (after appropriately adding T or F to the branches of a test) is the flowchart of a Ianov schema. This schema is strongly equivalent to the self-modifying program P. Figure 2(c) is the Ianov flowchart for the program of Example 1.

It follows that all of the problems (e.g. the halting, divergence and equivalence problems) which are decidable for Ianov schemata remain decidable for self-modifying schemata.

3. Remarks

Our model of self-modification does not permit test instructions to be modified, so that program flow of control is alterable only through label modification. Also, in our model, the size of "core storage" remains fixed - no instruction "locations" are created or destroyed. There are a number of ways in which the model could be generalized, without altering the basic result of the previous section. For example, it is not difficult to see that if we allow any, or all, of the following types of self-modification, we do not transcend the class of Ianov schemata.

1. Replace any instruction (test, operation, or modifier) by any given instruction.
2. Replace the instruction Li by the instruction Lj.
3. Create new instructions where these are specified in the program, e.g. <u>create</u> "L3 <u>do</u> A, <u>goto</u> LV2".
4. Erase instructions.
5. Allow fixed labels to be replaced by other fixed labels, or by variable labels.

or any of a number of different variations on this theme. Essentially, our model and any generalization of it as above, models self-modifying programs whose behaviour can be simulated by setting and testing finite switches. For example, the program of Example 1 can be simulated by the flowchart of Figure 2(d) which represents a Ianov schema augmented by suitable facilities for setting and testing switches. Using techniques very like those of Section 2, one can prove that any such augmented Ianov schema is equivalent to an ordinary schema, which is the basis for our claim that generalizations like 1-5 above do not change the basic result of Section 2.

On the other hand, if we permit a program to grow in an unbounded fashion, we can easily exceed the limits of Ianov schemata. For example, suppose we introduce three new instruction modifications.

1. copy L to LV. If LV is currently bound to Li, then upon execution, the body of the instruction labelled L replaces that labelled Li.
2. point L to LV. If LV is currently bound to Li, and if L labels do α, goto <label>, then after execution, L labels do α, goto Li.
3. LV ← gen. Upon execution, LV is bound to a new fixed label, distinct from any which has thus far occurred.

Notice that 1. and 2. together with our language of Section 2 will yield equivalent Ianov schemata. However, 3. permits dynamic core allocation for program space, and as the following example shows, takes us outside the class of Ianov schemata.

```
L0 if q goto L3 else goto L1
L1 do A, goto L2
L2 do H, goto L2
L3 do LV ← gen, goto L4
L4 copy L1 to LV, goto L5
L5 point L1 to LV, goto L0
```

If we interpret H as the halt operator, and define the trace of a program to be the set of operator-test sequences terminating with H, then the trace of this program is $\{q^{n-1}\bar{q}\,A^nH \mid n\geq 1\}$ which is not regular, and hence cannot be the trace of any Ianov schema.

The consideration of such growing programs leads to strong analogies with theories of growing automata [1]. Questions involving minimal complete sets of self-modification primitives, conditions for self reproduction, and the synthesis of self-modifying programs seem to us worthy of future investigation.

Acknowledgement

This work was done under National Research Council of Canada grant A7642. The author wishes to thank Richard S. Bird for a number of revealing conversations.

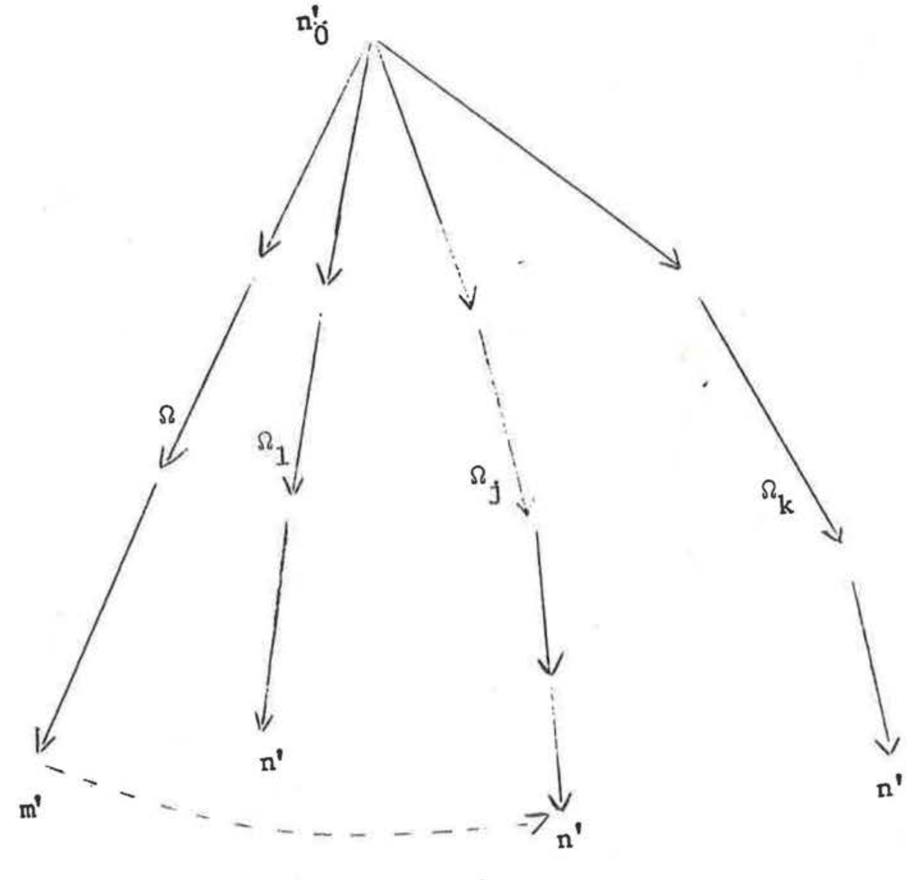


Figure 1(a)

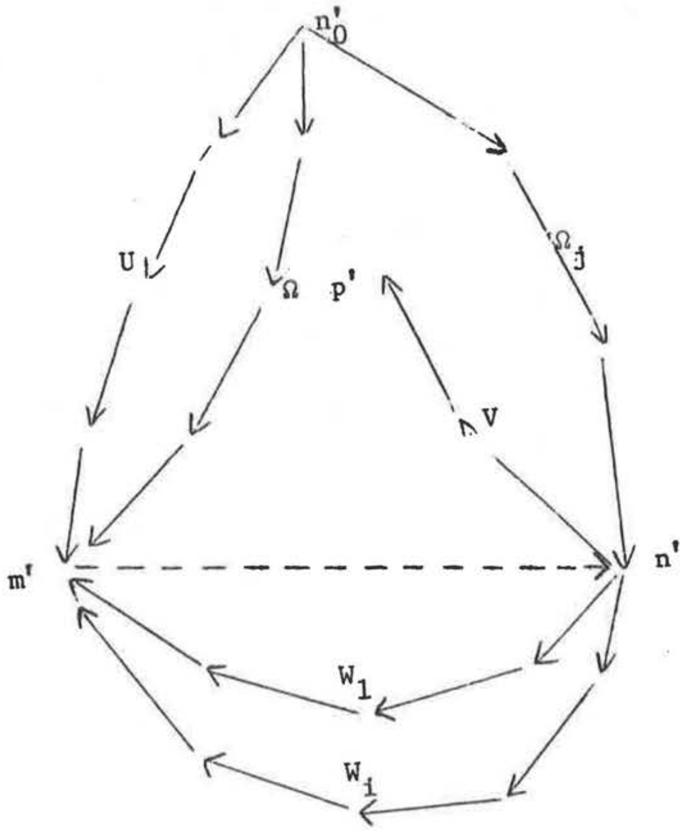


Figure 1(b)

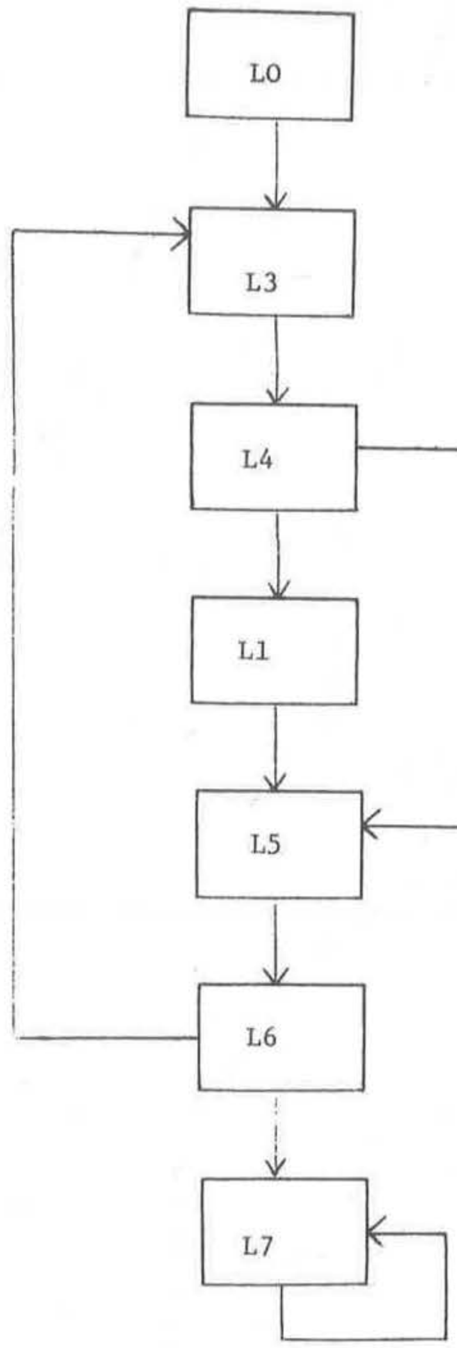
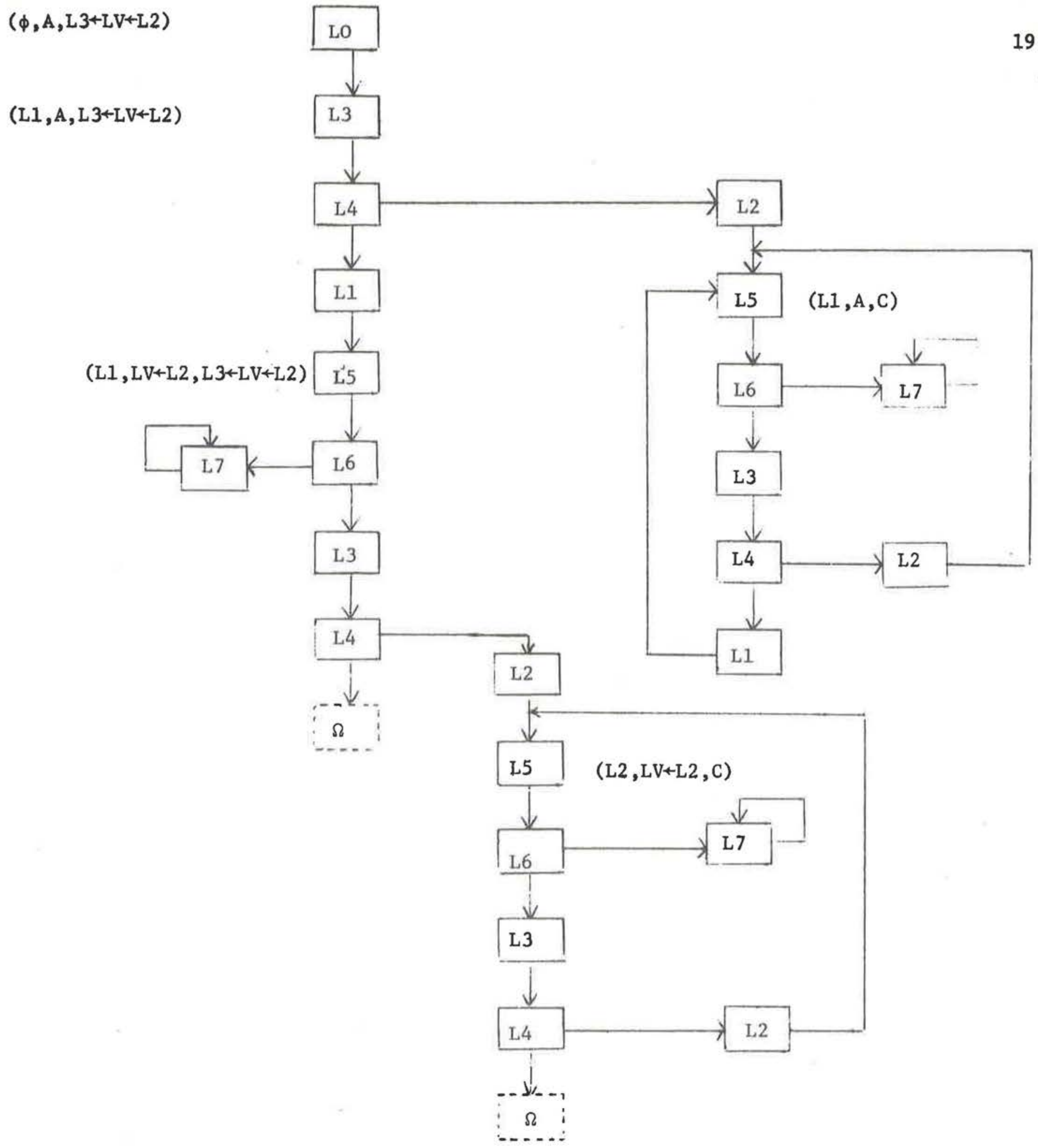


Figure 2(a)


(ϕ,A,L3←LV←L2)
L0
(L1,A,L3←LV←L2)
L3
L4
L2
L1
L5
(L1,A,C)
(L1,LV←L2,L3←LV←L2)
L5
L6
L7
L7
L6
L3
L3
L4
L2
L4
L1
L2
Ω
L5
(L2,LV←L2,C)
L6
L7
L3
L4
L2
Ω

<u>Figure 2(b)</u>

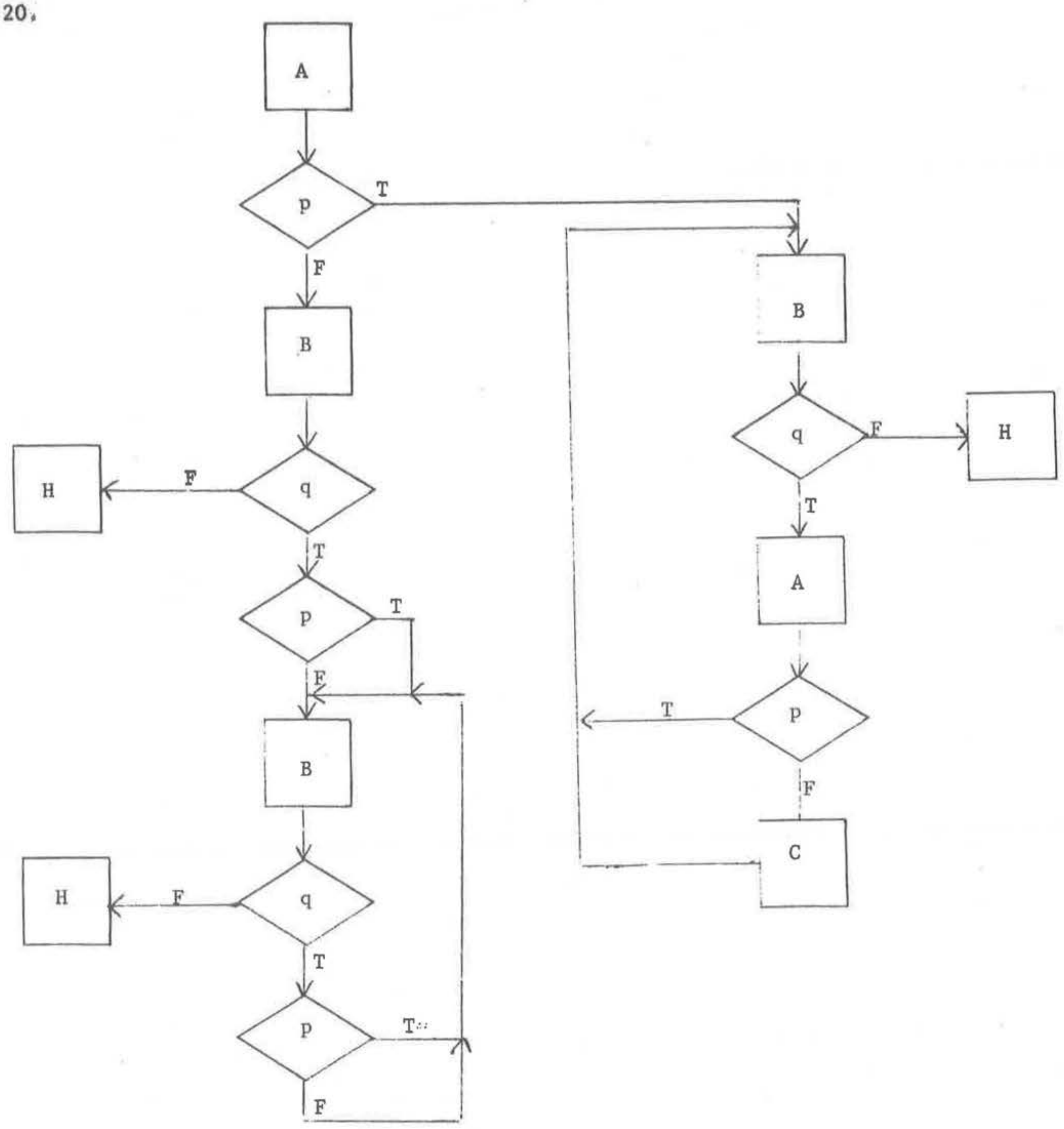


Figure 2(c)

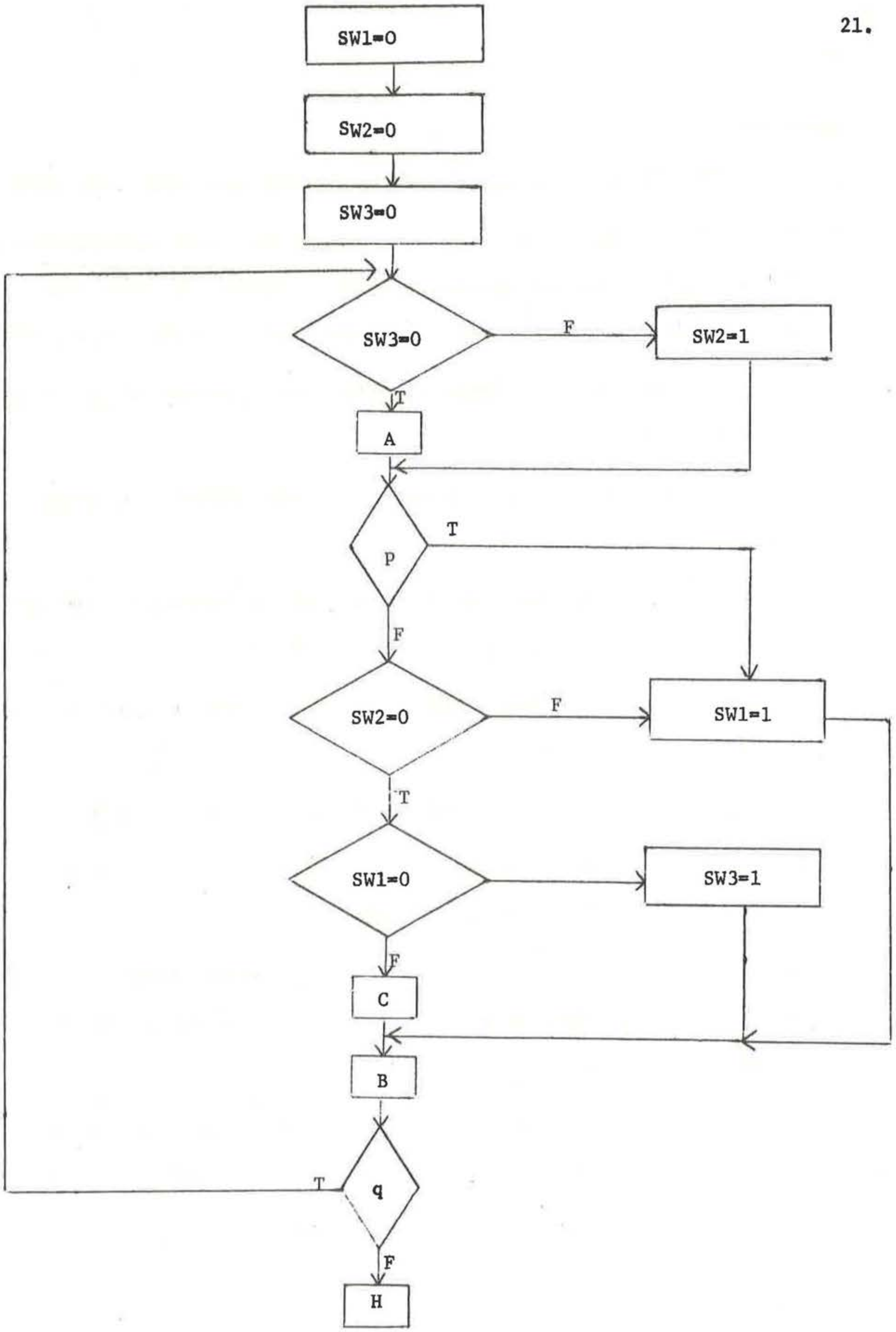

SW1=0
SW2=0
SW3=0
SW3=0
F
SW2=1
T
A
P
T
F
SW2=0
F
SW1=1
T
SW1=0
SW3=1
F
C
B
T
q
F
H

Figure 2(d)

References

1. E.F. Codd, "Cellular Automata", Academic Press, New York, N.Y. 1968.

2. H.H. Goldstine and J. von Neumann, Planning and coding problems for an electronic computing instrument, Part II, Vol.1, in "John von Neumann Collected Works", Vol.V, Pergamon Press, New York, N.Y., 1963.

3. I.I. Ianov, The logical schemes of algorithms, Problems of Cybernetics (1958), 75-125.

4. R.M. Karp and R.E. Miller, Parallel program schemata, J. Comput. System Sci. (1969), 147-195.

5. D.C. Luckham, D.M.R. Park, and M.S. Paterson, On formalised computer programs, J. Comput. System Sci. (1970), 220-249.

6. Z. Manna, and R.J. Waldinger, Toward automatic program synthesis, C. ACM (1971), 151-165.

7. J.D. Rutledge, On Ianov's program schemata, J.ACM (1964), 1-9.

8. H.R. Strong, Jr., Translating recursion equations into flow charts, J. Comput. System Sci. (1971), 254-285.

9. R.J. Waldinger, and R.C.T. Lee, PROW: A step toward automatic program writing, Proc. International Joint Conf. on Artificial Intelligence, Washington, D.C., 1969.

10. T. Winograd, Procedures as a Representation for Data in a Computer Program for Understanding Natural Language, M.I.T. Project MAC TR-84, February, 1971.