COMPUTATION USING *S*-PROGRAMS

A Thesis Submitted

to

LAKEHEAD UNIVERSITY

In Partial Fulfillment of the Requirement

for the

Degree

of

MASTER OF SCIENCE ©

by

Yandan Wen

1987

ISBN 0-315-44804-0

CONTENTS

ACKNOWLEDGEMENTS

I am deeply indebted to my supervisor Dr. C. F. Kent for his guidance and encouragement in the preparation of this thesis.

ABSTRACT

The concepts of $\mathcal{S}$ and $\mathcal{S}_n$-programs are given by Davis, Weyuker, 1983. Several parts of the complexity theory are carried out directly for $\mathcal{S}$ and $\mathcal{S}_n$-programs. The concepts of non-deterministic and deterministic computation from $\mathcal{S}$-programs are defined, and deterministic simulation of non-deterministic computation is proved. A universal $\mathcal{S}$-program for general (non-deterministic) computation is shown to require only one duplicate line label. Complexity results are given for these and other simulations, e.g. Turing Machine by $\mathcal{S}$-programs and the reverse. Cook,s Theorem for $\mathcal{S}_n$-programs is proved in full.

INTRODUCTION

In reference [2] Chapter 2 the authors use as their first model of abstract computation a very basic programming language, ***S***. Their first definition uses a set of registers, denoted X, V, Z, etc, with possible subscripts, a set of line labels, L, M, etc, possibly subscripted, and executable lines of only three types.

$V \leftarrow V+1$

$V \leftarrow V-1$ {leaves a null register null}

IF $V \neq 0$ GOTO L

In this definition the concept of an alphabet is avoided or, equivalently, the alphabet can be thought of as a 1-symbol "tally" language. Other programs, called $\mathcal{S}_n$-programs, are introduced which use "non-trivial" alphabets, A, and replace the first two rules above by "$V \leftarrow aV$", for $a \in A$, and "$V \leftarrow V^-$" (remove the rightmost symbol in V, if there is one), and the last by "IF V ENDS a GOTO L".

A certain portion of the usual theory of abstract computation is developed using these ***S*** symbols and, in due course, the theory is shown to be equivalent to the more usual formalization, the Turing Machine. After this equivalence is established, reference [2] becomes more conventional and develops much of its material using Turing Machines(TM's). In particular all results related to non-determinism are developed in the TM context. In this thesis, we develop a larger part of the basic theory, including non-determinism, in the ***S***-program context, directly.

In one small departure from the usual presentation (see[7]), we do not distinguish deterministic and non-deterministic ***S***-programs. There is only one type of ***S***-program, and it does both jobs depending upon whether the computation (execution) rules are deterministic or non-deterministic. The same could be done for TM's. We do not use indexings, or numberings of ***S***-programs in our work, but if that were necessary, a possible economy would result from having only one class, indexed once.

A further small departure occurs in our use of the "non-operable" instruction, $V \leftarrow V$, as a basic instruction. In reference [2,pp.23], $V \leftarrow V$, is a small subprogram, and its introduction therefore introduces several lines, and would complicate our statement of several complexity relations. The use of $V \leftarrow V$ is inessential. The instruction $V_0 \leftarrow V_0 - 1$, (or $V_0 \leftarrow V_0^-$), for a register which was conventionally always left empty, would accomplish the same purpose, i.e. it would pass control to the succeeding line of the ***S***-program($\boldsymbol{S}_n$-program).

In Chapter 1 we define the basic concepts of (general) ***S***-programs, indeterminacy of an ***S***-program, state the execution rules, and show how to replace a general ***S***-program by one in which each line label occurs the same number of times (constant indeterminacy). The chapter concludes with a formal proof that, from the viewpoint of set acceptance, a program, **P**, and its constant indeterminacy companion, **P'**, are equivalent.

Chapter 2 is devoted to developing the usual result about simulating non-deterministic computation by deterministic computation, but here for ***S***-programs directly. The usual exponential increase in the time required is shown to apply.

Chapter 3 develops the equivalence between general ***S***-programs and general(non-deterministic) TM's. This is a direct equivalence and avoids the usual chain of equivalences, from non-deterministic TM to deterministic TM to (deterministic) ***S***-program to ***S***-program. Complexity results, in terms of time requirements, are given, but a limitation of the first type of ***S***-program emerges. This restriction is a consequence of the previously mentioned fact that ordinary ***S***-programs use tally languages, for which the length and the value of words, x, coincide (or differ by at most 1).

Chapter 4 begins by showing how to replace constant-indeterminacy ***S***-programs, of indeterminacy α, by programs of constant indeterminacy 2. This result is used to exhibit a universal (non-deterministic) ***S***-program, which differ from the universal (deterministic) ***S***-program of reference [2,pp.58] by the insertion of only one duplicate line label, (one additional line). A second construction is given which does not depend upon the constant-indeterminacy result, but is more complex.

Finally, in Chapter 5, we return to a classic complexity result, Cook's Theorem, formulated here for $\boldsymbol{S}_n$-programs. Because of the lack of distinction between value of x and length of x for tally languages, and consequently for reference [2]'s first model of an ***S***-program, we change here to $\boldsymbol{S}_n$-programs. In terms of complexity, ***S***-program are equivalent to $\boldsymbol{S}_1$-programs. Cook's Theorem is given the usual sort of proof, *by exhibiting an algorithm for constructing a* conjunctive normal form, Ψ_x, for each ***S***-program, **P**, and input x. However, we do go one step farther than most presentations by showing that the CNF, Ψ_x, is satisfied if <u>and only if</u>, P accepts x.

CHAPTER 1

Programming Language *S*

In this Chapter, we describe the **Programming Language *S*** and the **syntax** of *S*-program. Our definitions follow closely these in reference [2,pp.15-17], with minor differences.

1.1 *S*-Programs

Like any other program languages, programming language ***S*** has

a. *Input variables* :

$X_1, X_2, X_3, \ldots$

b. *Local variables* :

$Z_1, Z_2, Z_3, \ldots$

c. *Output variable* :

Y

d. *Label names* :

$A_1, B_1, C_1, A_2, B_2, \ldots$

e. One of the following *statements*(instructions) in the table 1.1

Instruction	Interpretation
$V \leftarrow V+1$	Increase by 1 the value of the variable V. Variables are the names for registers holding values during a computation from the program. At the beginning of each computation all variables have initial value 0.
$V \leftarrow V-1$	If the value of V is 0, leave it unchanged; otherwise decrease by 1 the value of V.
$V \leftarrow V$	A non-operable line, control passes to next line
IF $V \neq 0$ GOTO L	If the value of V is not zero, perform the instruction with label L next; Otherwise proceed to the next instruction in the list.

where V may be any variable and L may be any label.

Table 1.1

f. *A Program* is an ordered list of instructions (finite).

g. *The length* of the program is the length of the list of those instructions, i.e. the number of lines.

h. *The empty program* contains no instruction, its length is 0.

i. *A state of a program* **P** is a list of equations of the form V=m, where V is a variable and m is a number, including exactly one equation for each variable that occurs in **P**.

j. *A label block* is a maximal set of consecutive lines of the program, all bearing the same label.

1.2. Syntax of *S*-programs

Suppose we have a program **P** of length n. Let σ be a state of **P** and let V be a variable that occurs in σ. The *value of V* at σ is then the (unique) number q such that the equation V=q is one of the equations making up σ. In order to say what happens "next", we also need to know which instruction of **P** is about to be executed. We therefore define a *snapshot* or *instantaneous description* of a program **P** of length n to be a pair (i,σ) where $1 \leq i \leq n+1$, and σ is a state of **P**. (Intuitively the number i indicates that it is the ith instruction which is about to be executed; i=n+1 corresponds to a "stop" instruction).

If $s=(i,\sigma)$ is a snapshot of **P** and V is a variable of **P**, then the *value of* V at s just means the value of V included in the state σ.

We shall handle the deterministic versus non-deterministic issue not by considering programs of the two types, but by considering only one type of program and representing deterministic computation, **rule**-II, as a special case of non-deterministic (ordinary) computation, **rule**-I. Non-determinism in computations is handled by use of multiple occurrences of the same line label, say L, so that in response to a "GOTO L"* the computation may choose non-deterministically any line labelled L just like non-deterministic Turing machine, which might have, for certain combinations of state and scanned symbol, more than one possible chice of behavior, defined in [4,pp.204]. In deterministic computation of [2,pp.25] a "GOTO L" command always chooses the first line of label L in the program.

1.2.1 Non-Deterministic Execution Rule-I

For a given *S*-program **P**, If (i,σ) is a non-terminal snapshot of *P*, we define the *successor* of (i,σ) to be the snapshot (j,τ) following the rules below.

* The unconditional GO TO L will be used below and represents the small subprogram $V \leftarrow V + 1$, IF $V \neq 0$ GO TO L, using some variable V which does not appear elsewhere in the program.

Rule-I

Line changing flowchart

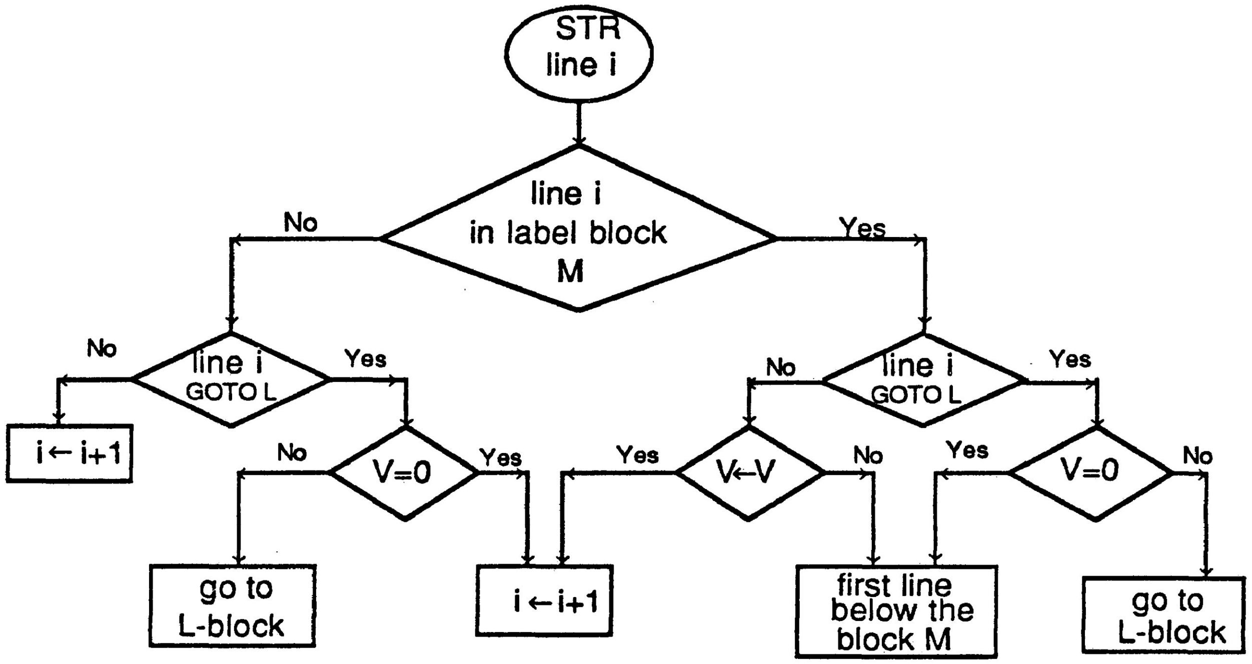

Fig 1.1

I-A. The ith line is not in any label block, i.e. is not a labelled line, σ, and contains the equation V=m.

(a.1). The ith line is not the form "IF V≠0 GOTO L".

i=i+1;

τ has three subcases:

i) The *i*th line is "V←V+1", τ is obtained from σ by replacing the equation V=m by V=m+1.

ii) The *i*th line is "V←V-1", τ is obtained from σ by replacing the equation V=m by V=m-1 if m≠0; if m=0, $\tau=\sigma$.

iii) The *i*th line is "V←V", $\tau=\sigma$.

(a.2). The i^{th} line is the form "IF $V \neq 0$ GOTO L".

$\tau = \sigma$;

j has two subcases:

i) If $V=0$, $j=i+1$.

ii) If $V \neq 0$, j is the number of any line in any L-block.

I-B. The i^{th} line is in the label block, B_M, of label M, σ contains the equation $V=m$.

(b.1). The i^{th} line is not the form "IF $V \neq 0$ GOTO L".

(b.1.1). The i^{th} line is "$V \leftarrow V$".

$\tau = \sigma$;

$j=i+1$.

(b.1.2). The i^{th} line is not "$V \leftarrow V$".

j is the number of the first line below B_M.

τ has two subcases:

i) The i^{th} line is "$V \leftarrow V+1$", τ is obtained from σ by replacing the equation $V=m$ by $V=m+1$.

ii) The i^{th} line "$V \leftarrow V-1$", τ is obtained from σ by replacing the equation $V=m$ by $V=m-1$, if $m \neq 0$; if $m=0$, $\tau = \sigma$.

(b.2). The i^{th} line is the form "IF $V \neq 0$ GOTO L".

$\tau = \sigma$;

j has two subcases:

i) If $V=0$, j is the number of the first line below B_M.

ii) If $V \neq 0$, j is the number of any line in any L-block.

1.2.2 Deterministic Execution Rule-II

Rule-II

Line changing flowchart

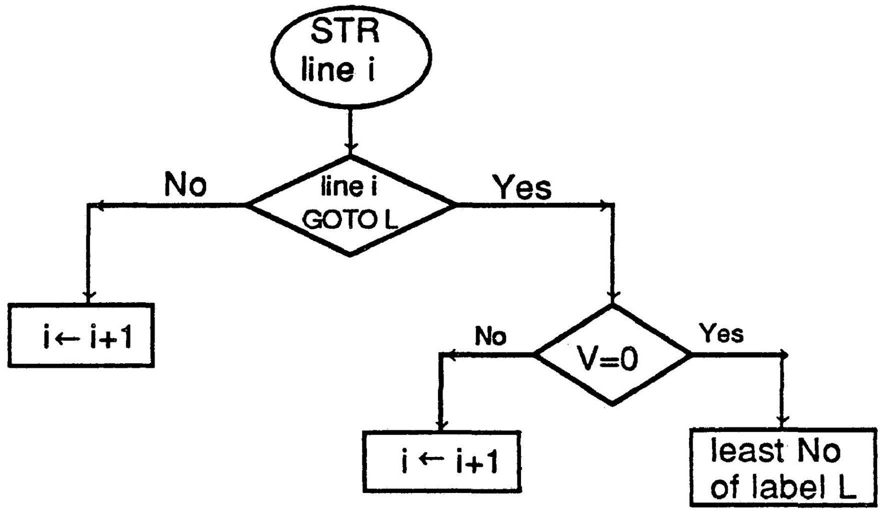

Fig 1.2

II-A The i^{th} line is not the form "IF V≠0 GOTO L".

i=i+1;

τ has three subcases:

i) The i^{th} line is "V←V+1", τ is obtained from σ by replacing the equation V=m by V=m+1.

ii) The i^{th} line is "V←V-1", τ is obtained from σ by replacing the equation V=m by V=m-1 if m≠0; if m=0, τ=σ.

iii) The i^{th} line is "V←V", τ=σ.

II-B The i^{th} line is the form "IF V≠0 GOTO L".

τ=σ;

j has two subcases:

i) If V=0, j=i+1.

ii) If V≠0, j is the number of the first line in the first L-block.

DEFINITION 1.1 *A computation*, C, of a program **P** of length n is a sequence $s_1, s_2, \ldots, s_k$ of snapshots of **P** such that s_{i+1} is the successor of s_i for $i=1,2,\ldots,k-1$. $s_1=(1,\sigma_1)$ which is initial snapshot, $s_k=(n+1,\sigma_k)$ which is terminal snapshot.

DEFINITION 1.2 Any computation C, which is being executed by **Rule-I**, is called a **non-deterministic computation**, for convenience, written **NDC** to emphasize this case.

DEFINITION 1.3 Any computation C under **Rule-II** is called a **deterministic computation**, for convenience, written **DC** to emphasize this case.

As stated earlier, we do not classify ***S***-programs as "deterministic" or "non-deterministic", but have only one set of ***S***-programs. It is the computations from these programs which are called deterministic, or non-deterministic. Our general term "computation" thus stands for what other authors call a non-deterministic computation (program, machine, etc.).

Frequently it happens that restricted classes of programs suffice for certain types of computations, we consider several results of this nature in this paper, but begin with a very simple example, which should not be confused with the more important results of the next section.

Example **1.1**

```
        [L]        X←X+1
                   IF X≠0 GOTO L
        [L]        X←X-1
```

Pro.1.1

This program contains two instructions having the same label. According to the definition of **DC**, its snapshot, in effect, interprets a branch instruction as always referring to the first statement. So the program is equivalent to the following:

```
        [L]        X←X+1
                   IF X≠0 GOTO L
                   X←X-1
```

Pro. 1.2

We use **DCSP** to represent the program which is executed by **Rule**-II.

Lemma 1.1 Given a general *S*-program, **P**, there is a companion program, **P***, in which no line label occurs more than once, and so that the set of deterministic computations from **P** is the same as the set of all computations from **P***.

Proof If line i of **P** has label L, but is not the top line of label L, then control in a deterministic computation can pass to line i only if the line immediately above i is executed. Since this happens whether or not the label L is present in line i, we can erase all prefix labels L in lines which are not the top one with label L. In this new program, **P***, all computations become **Rule-I** computations. Therefore, for program **P***, the set of all computations is the same as the set of deterministic computations of **P**.

###

1.3. *S*-programs of Indeterminacy δ

In a general *S*-program, there may be several different occurrences of a label. One label that occurs may not have to occur the same number of times as another label. We refer to the number of occurrences of label L in the label block as the multiplicity of L, and the maximum multiplicity of any label in **P** as the indeterminacy of **P**, say δ.

Example **1.2**

```
        X←X-1
  [L]   X←X
  [L]   X←X+1
  [M]   Z←Z+1
  [L]   IF Z≠0 GOTO M
  [N]   Z←Z-1
  [L]   IF X≠0 GOTO N
  [M]   IF Z≠0 GOTO L
        Y←Y+1
```

Pro. 1.3

In **Pro.1.3**, the multiplicity of L in first L-label block is 2, the multiplicity of M in first M-label block is 1, and so on. The indeterminacy of this program, δ, is 2.

In our proof, in the next chapter, of the standard result on simulation of non-deterministic computation by deterministic, we will require an intermediate result, like Theorem1.1(below), to show that general *S*-program of indeterminacy, δ, can be replaced by program in which each labelled line has indeterminacy δ+1. Thus, we next turn to the procedure for this replacement.

1.3.1 Algorithm for Indeterminacy-AFI

Let **P** be a general ***S***-program

Step 1. Determine the indeterminacy δ, and let $\alpha=\delta+1$.

Step 2. For a fixed label L.

i). Put L-blocks in order as $B_{L_1}, B_{L_2}, \ldots, B_{L_m}$ (suppose there are m L-blocks in **P**).

ii). In B_{L_i}, replace all labels L by L_i.

iii). (a) The multiplicity of L_i, δ_i, will be made equal to α.

(a.1) $i\neq m$, add statement "GOTO L_{i+1}" with label L_i below the last line of the L_i-block. Call this new label block B'_{L_i}.

(a.2) $i=m$, add statement "GOTO L_1" with label L_m below the last line of the L_m-block. Call this new label block B'_{L_m}.

(b) The multiplicity of L_i, may not yet be equal to α. Insert statements "$V\leftarrow V$" with label L_i between the last line of the old L_i-block and the new last lines added in (a) until the multiplicity of L_i is equal to α. Call this new label block B'_{L_i}.

Step 3. Repeat step 2 until all L-blocks are treated.

Step 4. Inside any line which contains form "GOTO L", replace by "GOTO L_1"; Note that no line which lacks an internal or a prefix label is changed.

Step 5. Repeat step2, 3 and 4 until all labels have been treated.

Now we use **NDCSP**, **P'**, to represent the program which has been changed by the Algorithm-**AFI** and is executed by **Rule-I**.

For use in the chapter 4 later, we give the simulation steps of **P'** simulating a one step GOTO statement of **P**.

Corollary 1.1 An **NDCSP**, **P'**, takes $\leq \alpha$ steps to simulate a one step GOTO statement of a general *S*-program, **P**.

Proof For a statement "GOTO L" in **P**, there is the corresponding line "GOTO L_1" in **P'**. When a **P**-computation executes "GOTO L", the next step will go to one of lines labelled L in **P**. By **AFI** step-2, the **P'**-computation, in simulating this step may go though all L-blocks, i.e. α steps, in the worst case. So **P'** may take α steps to simulate a one step GOTO statement of **P**.

###

In Ex.**1.2**, δ is 2 for label L; according to the above procedure after step 2, it looks like:

```
         X←X-1
[L1]     X←X
[L1]     X←X+1
[L1]     GOTO L2

[M]      Z←Z+1

[L2]     IF Z≠0 GOTO M

[N]      Z←Z-1
```

```
[L_3]   IF X≠0 GOTO N

[M]     IF Z≠0 GOTO L
        Y←Y+1
```

Pro. 1.4

After step 3 and step4, the program looks like:

```
        X←X-1
[L_1]   X←X
[L_1]   X←X+1
[L_1]   GOTO L_2

[M]     Z←Z+1

[L_2]   IF Z≠0 GOTO M
[L_2]   X←X
[L_2]   GOTO L_3

[N]     Z←Z-1

[L_3]   IF X≠0 GOTO N
[L_3]   X←X
[L_3]   GOTO L_1

[M]     IF Z≠0 GOTO L_1
        Y←Y+1
```

Pro. 1.5

After step 5, we have:

```
        X←X-1
[L_1]   X←X
[L_1]   X←X+1
[L_1]   GOTO L_2

[M_1]   Z←Z+1
[M_1]   Z←Z
[M_1]   GOTO M_2
```

```
[$L_2$]    IF $Z \neq 0$ GOTO $M_1$
[$L_2$]    $X \leftarrow X$
[$L_2$]    GOTO $L_3$

[$N_1$]    $Z \leftarrow Z-1$
[$N_1$]    $Z \leftarrow Z$
[$N_1$]    GOTO $N_1$

[$L_3$]    IF $X \neq 0$ GOTO $N_1$
[$L_3$]    $X \leftarrow X$
[$L_3$]    GOTO $L_1$

[$M_2$]    IF $Z \neq 0$ GOTO $L_1$
[$M_2$]    $Z \leftarrow Z$
[$M_2$]    GOTO $M_1$
           $Y \leftarrow Y+1$
```

Pro. 1.6

1.3.2 NDCSP Accepts the Same Sets as General (Non-Deterministic) Computation of an *S*-Program

In this section we show that the replacement of an *S*-program, **P** of length n, by another, **P'**, of constant label multiplicity, as in the last section, does not affect the set accepted by general computation.

DEFINITION 1.4 Let f be the one-to-one order preserving map of the lines of the original program **P** (of length n onto the lines of **P'** of length N) that are not added in step-2 iii) of **AFI**. Call this the correspondence map, and for i, $1 \le i \le n$, f(i) is called the corresponding line for **P**, and f(n+1)=N+1.

Note that the difference between line i, of **P**, and line f(i) of **P'** are superficial. Line f(i) differ from line i, at most, in change of the prefix label, and change of the internally mentioned label, if line i is a GOTO line.

Lemma 1.2 Function f which is defined as above is an injection from **P** to **P'**.

Proof For the line i in **P**.

(a) line i is not in any block.

(a.1) line i is not of the form "GOTO L".
There is exactly one corresponding line f(i), which has the same form as i , by **AFI**-step 4.

(a.2) line i is of the from "GOTO L".
There is exactly one corresponding line f(i), which is changed to "GOTO L_1", by **AFI**-step 4.

(b) line i is in M-block B_M.

(b.1) line i is not of the form "GOTO L".
There is exactly one corresponding line f(i), which has the same form as i, in B'_{M_j} by **AFI**-step-2 (suppose

this M-block is the jth occurrence M-block in **P**).

(b.2) line i is of the from "GOTO L".

There unique corresponding line f(i) is the form that comes from (a.2) and (b.1) in this case. Thus, for all $i, 1 \le i \le n$, in **P**, there is exactly one corresponding line f(i) in **P'**. Therefore f is an injection from **P** to **P'**.

###

For both lemmas below we assume that **P** has no "$V \leftarrow V$" instruction. If so, then omitting such lines from **P** does not affect the inputs accepted. **P**(x) denotes that x is the input of program **P**.

Lemma 1.3 If there is a k-step computation of **P**(x) leading to a snapshot $s=(i,\sigma)$, then there is a computation of **P'**(x) leading to a snapshot $c=(f(i),\sigma)$.

Proof By induction on k.

Basis k=1.

The only 1-step computation of **P**(x) leads to the snapshot $s_1=(1,\sigma_1)$, where σ_1 is the initial state. By **AFI**, the initial snapshot of **P'**(x), c_1, has the same state σ_1, and starts at first line, i.e. $c_1=(1,\sigma_1)$. Hence in this case the assertion is correct.

Assume, for induction.

For any k-step computation C of **P**(x) leading to the snapshot $s_k=(i_k,\sigma_k)$, there is a computation C' of **P'**(x) leading to the snapshot $c_{k_p}=(f(i_k),\sigma_k)$.

Consider the (k+1)th-step.

Suppose there is a k+1-step computation of **P**(x) leading to the snapshot $s_{k+1}=(i_{k+1},\sigma_{k+1})$. We use the induction assumption on the part of that computation through step k, and we have three cases:

case 1) Line i_k is not of the form "GOTO L", $i_k \neq n+1$.

In this case state σ_k, in both computations, passes to the same state σ_{k+1}, and we need only check that the control is correctly passed.

a) line i_k is not in a label block, then $i_{k+1}=i_k+1$. In **P**'(x), control passes from $f(i_k)$ to the next line, i.e. $f(i_k+1)=f(i_k)+1$, so the corresponding snapshot of s_{k+1} is: $c_{(k+1)_p}=(f(i_k)+1,\sigma_{k+1}) = (f(i_k+1),\sigma_{k+1}) = (f(i_{k+1}),\sigma_{k+1})$.

Thus in this case, there is a computation of **P**'(x) leading to snapshot $c_{(k+1)_p}=(f(i_{k+1}),\sigma_{k+1})$.

b) line i_k is in a label block, say B_M.

control passes from i_k to the top line i_{k+1} below B_M in **P** (since line i_k is not $V \leftarrow V$); In **P**'(x) control passes from $f(i_k)$ to the top line, below B'_{M_j} (suppose this M-block is the jth occurrence M-block in **P**), and this line is $f(i_{k+1})$, so the corresponding snapshot of s_{k+1} is again:

$$c_{(k+1)_p}=(f(i_{k+1}),\sigma_{k+1}).$$

Thus, in this case, there is a computation of **P**'(x) leading to snapshot $c_{(k+1)_p}=(f(i_{k+1}),\sigma_{k+1})$.

case 2) Line i_k is of the form "GOTO L", $i_k \neq n+1$.

In this case no change is made in the state, in either computation, so $\sigma_k = \sigma_{k+1}$.

Control passes from line i_k to some line i_{k+1} in one of the L-blocks in **P**. We must see that there is a **P'** computation with correct transfer of control from snapshot $(f(i_k), \sigma_k)$ to $(f(i_{k+1}), \sigma_k)$. Since the executable command of line i_k is "GOTO L", the command of $f(i_k)$, according to algorithm **AFI** is "GOTO L_1". The correct computation for **P'** in this case selects the last line of block B'_{L_1}, last line of $B'_{L_2}, \ldots$, last line of $B'_{L_{q-1}}$, and last line of B'_{L_q}. Thus there is a computation **P'**(x) which eventually arrives at snapshot $(f(i_{k+1}), \sigma_k)$, as required.

case 3) $i_k = n+1$.

Then i_{k+1} stays on line n+1, state σ_{k+1} is unchanged. Since $f(i_k)$ is the corresponding line, then, by definition $f(i_k) = N+1$ (suppose program **P'** is of length N), thus the corresponding snapshot of s_{k+1} is:

$$c_{(k+1)_p} = (f(i_{k+1}), \sigma_k) = (N+1, \sigma_k).$$

This complete the induction step, so the result of the lemma follows.

End of proof.

###

Lemma 1.4 If there is a k-step computation of **P**'(x) leading to a snapshot $c=(f(j),\sigma)$, where f(j) is a corresponding line of **P**, then there is a computation of **P**(x) leading to a snapshot $s=(j,\sigma)$.

Proof By complete induction on k.

Basis k=1.
The same argument as in the basis case of Lemma 1.3 applies.

Inductive step
Assume for all h<k, if there is a h-step computation of **P**'(x) leading to snapshot $c_h=(f(i_h),\sigma_h)$, where $f(i_h)$ is a corresponding line of **P**, then there is a computation of **P**(x) leading to snapshot $s_{p_h}=(i_h,\sigma_h)$.

Now consider a k-step computation, $c_1,c_2,\ldots c_k$, of **P**'(x) leading to snapshot $c_k=(f(i_k),\sigma_k)$, where $f(i_k)$ is a corresponding line of **P**.

Choose the largest $h_0<k$, so that the h_0-step part of the k-step computation gives snapshot $c_{h_0}=(f(i_{h_0}),\sigma_{h_0})$, for a corresponding line $f(i_{h_0})$. By assumption there is a computation of **P**(x) leading to snapshot $s_{p_{h_0}}=(i_{h_0},\sigma_{h_0})$.

We diagram the present assumptions for greater understanding.

P-computation	**P'**-computation
$s_1 = (1,\sigma_1)$	$c_1 = (1,\sigma_1)$
$\vdots$	$\vdots$
$s_{p_{h_0}} = (i_{h_0},\sigma_{h_0})$	$c_{h_0} = (f(i_{h_0}),\sigma_{h_0})$
$s_{p_{h_0+1}} = (i_k,\sigma_{h_0+1})$	$c_{h_0+1} = (\quad ,\sigma_{h_0+1})$
	$\left.\vdots\right\}$ results of executing non-corresponding lines
	$c_k = (g,\sigma_k)$

use g to denote the first corresponding line number executed by **P'** after step h_0.

First we note that σ_{h_0+1}, the result of executing lines i_{h_0}, or equivalently $f(i_{h_0})$, on the state s_{h_0}, is the same as σ_k, since in the **P'**-computation the intervening steps do not change the state. Thus, to prove the lemma it is sufficient to show that $i_k=f^{-1}(g)$.

We consider cases on the joint form of the line i_{h_0} and its correspondent $f(i_{h_0})$.

a) lines i_{h_0} and $f(i_{h_0})$ are not GOTO lines.

a.i) both are labelled lines.

In this case **P** control passes to the top line below the label block in which i_{h_0} lies, and **P'** control to the top line below the label block in which $f(i_{h_0})$ lies. But, this is a corresponding pair of lines and $f^{-1}(g)=i_k$.

a.ii) neither line i_{h_0} nor $f(i_{h_0})$ is labelled.

In this case, control in both programs passes to the next line, i.e. $i_k=i_{h_0}+1$, $g=f(i_{h_0})+1$ and $f^{-1}(g)=i_k$.

b) lines i_{h_0} and $f(i_{h_0})$ are GOTO lines.

In this case line i_{h_0} is of the form "GOTO L" and line $f(i_{h_0})$ of the form "GOTO L_1". Now, line g is the first corresponding line of the $f(i_{h_0})$. Thus line g must lie in same L_i block, since the actions of the non-corresponding lines in steps h_0+1 through k-1 cannot transfer control outside of the (set of) L_i blocks. Since g is a corresponding line in an L_i block, there is a line i_k in some L-block of **P** which contains the corresponding instruction, so that $f^{-1}(g)=i_k$.

This completed the induction step and, hence, the lemma is proved.

###

From lemma 1.3 and lemma 1.4 , we can get a theorem as following:

Theorem1.1 Given a general *S*-program, **P**, there is another, **P**', in which all line labels occur with the same multiplicity, such that **P** accepts X if and only if **P'** accepts X.

Proof The lemmas above show that the same snapshots can be attained by **P** and **P'**, then, for given X, there is a halting computation of **P**(x) if and only if there is a halting computation of **P'**(x).

###

CHAPTER 2

Simulation Of General Computation By Deterministic Computation

In this Chapter, we prove, for *S*-programs directly, that if a set X can be accepted by an *S*-program, **P**, then there is another *S*-program, **P***, which accepts X and every computation from **P***, is deterministic(Lemma 4.6.1 of [4] proves that a non-deterministic Turing machine can be simulated by a deterministic one). In this sense general computation is no more powerful than deterministic computation except, of course, that it seems to require an exponential increase in time for the deterministic computation. The usual device used in simulating a non-deterministic computation by a deterministic one is to use a subprogram to generate instructions which remove choices from the main program in executing GOTO statements with multiple destinations. We refer to such instruction as "clock sequences", definition 2.1, below. The subprogram generates a sequence, C, of numbers $c_0, c_1, \ldots, c_m$; $0 \le c_i < \alpha$, where α is the indeterminacy of **P**, and **P*** executes a computation simulating m steps of some **P**-computation and consuming the clock sequence as it reads its values to direct the choice of GOTO destinations.

2.1 Clock Sequence for Simulation

DEFINITION 2.1 Any given non-negative integer c, can be represented by base $\alpha, \sum_{i=0}^{m} d_i \alpha^i$, $0 \le d_i \le \alpha - 1$. Call the sequence $\{d_m, d_{m-1}, \ldots, d_0\}$ a **Clock Sequence** for integer c.

The individual values d_i, $0 \le i \le m$, will be used to instruct the simulator which of the α possible line labels involved in a "GOTO L" should be executed. We call each d_i a <u>clock sequence value</u>.

Theorem 1.1 tells us that for general *S*-program computation from **P**, it suffices to consider only *S*-programs, **P'**, of constant indeterminacy. Thus, in order to simulate **P** by a **DCSP**, **P***, we need only consider a **NDCSP**, **P'**, in which all the labels in a label block occur α times, and simulate the computations of **P'** by to a **DCSP**, **P***.

Theorem 2.1 If a **NDCSP**, **P'**, accepts a set X, then there is a **DCSP**, **P***, which also accepts X.

Proof Suppose the constant indeterminacy of the given **NDCSP**, **P'**, is α.

First we make an intermediate change in **P'**.

c1. In each L-block, replace the α occurrences of L by $L_0, L_1, \ldots, L_{\alpha-1}$, in order.

c2. Insert the unlabelled line "GOTO L_α" in the L-block immediately below each L_j-line, $0 \le j \le \alpha-2$. Here we use the unconditional "GOTO L_α" as a basic command, as an abbreviation for "IF $V \ne 0$ GOTO L_α" for same register, V, which is initialized non-zero.

c3. Insert "$V \leftarrow V$" with label L_α below line $L_{\alpha-1}$.

After this change, no label is used more than once in program **P'** and there is no L-line to support the "GOTO L" statements of **P'**(see pro2.2). Thus, execution of **P'** would result in termination in response to each GOTO statement, however our attention is not on the execution of this temporary program, for which we retain the name **P'**.

Now we turn to creation the simulator, **DCSP**, **P*** from **P'** of length ℓ.

The flowchart for turning **P'** into **P*** follows.

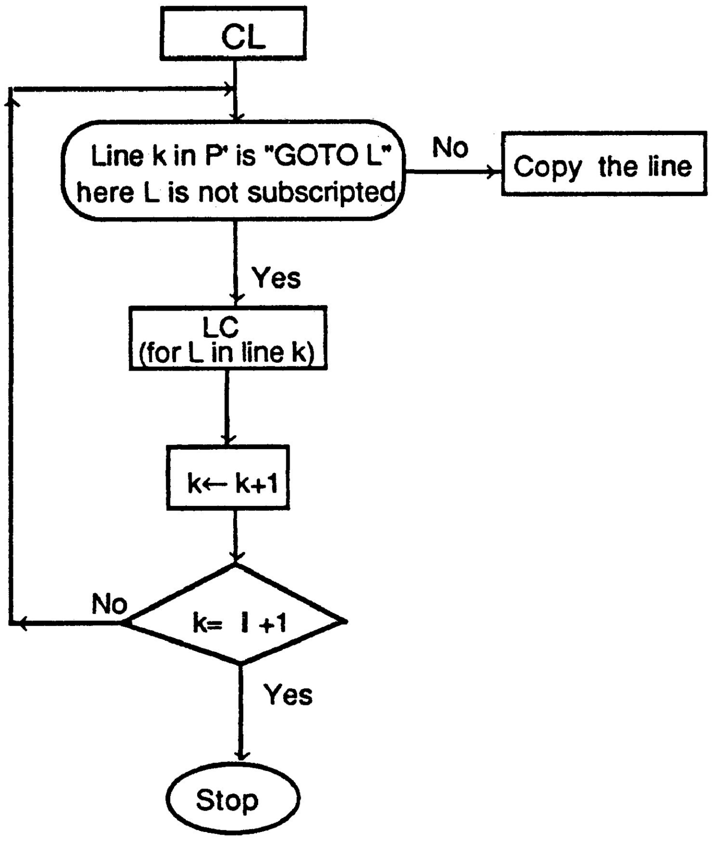

Fig 2.1

I. **P*** contains the following lines to create the clock sequence, call CL which is:

```
CL      $C \leftarrow C+1$        {C is the clock sequence, it is
                                  initially 0 }
        $Z_1 \leftarrow C$
```

II. If the line is not of the form "GOTO L" , here L is not subscripted, then the corresponding form in **P*** is the same as the one before in **P'**.

III. If the line is of the form "GOTO L" but not "GOTO L_α", then the corresponding form in **P*** is changed to "GOTO LC" which transfers control to the subprogram LC, below:

```
LC      $Z_4 \leftarrow 0$
        IF $Z_1 \neq 0$ GOTO RML     {RML will compute the remainder
                                     and the quotient of $Z_1$ modulo $\alpha$}
        GOTO CL                      {the current clock sequence has
                                     been consumed. Create another
                                     clock sequence.}
```

IV Read next line, if it is line $\ell+1$, stop the simulation; Otherwise, go back to the step II.

Note that if the program **P'** had h different labels, there are h kinds of Change-subprograms.

By definition 2.1 $rm(Z_1,\alpha)$ is the clock sequence value and $qt(Z_1,\alpha)$ is the new(reduced) clock sequence. We use registers Z_2 for $rm(Z_1,\alpha)$ and Z_4 for $qt(Z_1,\alpha)$, in subprogram RML. Now $rm(Z_1,\alpha)=Z_1-\alpha*[Z_1/\alpha]$ and $qt(Z_1,\alpha)=[Z_1/\alpha]$, but to avoid analysing the complexity of primitive recursive functions computed by *S*-programs, we compute these two quantities by the following straightforward program.

```
[RML]        $Z_1 \leftarrow Z_1 - 1$                 {this part of the program reduces
             $Z_2 \leftarrow Z_2 + 1$                 $Z_1$ by $\alpha$, If a 0 arises before the
             IF $Z_1 \neq 0$ GOTO $RL_2$          last line of this part, we exit
             GOTO FL                   with the present $Z_4$ and $Z_2$.}
 [$RL_2$]      $Z_1 \leftarrow Z_1 - 1$                 {decrement clock}
             $Z_2 \leftarrow Z_2 + 1$                 {increment remainder}
             IF $Z_1 \neq 0$ GOTO $RL_3$
             GOTO FL
                 .
                 .
                 .                     {there are $3(\alpha-1)$ lines until
[$RL_{\alpha-1}$]    $Z_1 \leftarrow Z_1 - 1$                 $RL_{\alpha-1}$}
             $Z_2 \leftarrow Z_2 + 1$
             IF $Z_1 \neq 0$ GOTO $RL_\alpha$
              GOTO FL
[$RL_\alpha$]       $Z_1 \leftarrow Z_1 - 1$
             $Z_2 \leftarrow 0$                     {reset remainder value to 0}
             $Z_4 \leftarrow Z_4 + 1$                 {increment quotient}
             IF $Z_1 \neq 0$ GOTO RML           {start another reduction of $Z_1$ by
             GOTO FL                   $\alpha$}
[FL]         $Z_1 \leftarrow Z_4$
             IF $Z_2 \neq 0$ GOTO $L'_1$          {this part of the program utilises
             GOTO $L_0$                    the clock sequence value in
[$L'_1$]        $Z_2 \leftarrow Z_2 - 1$                 register $Z_2$ to direct control to a
             IF $Z_2 \neq 0$ GOTO $L'_2$          particular line $L_0, L_1, \ldots, L_{\alpha-1}$, the
             GOTO $L_1$                    new labels for the $\alpha$-tuple of lines
              ...                      labelled L inthe program P'.}
[$L'_{\alpha-2}$]     $Z_2 \leftarrow Z_2 - 1$
             IF $Z_2 \neq 0$ GOTO $L'_{\alpha-1}$
             GOTO $L_{\alpha-2}$
[$L'_{\alpha-1}$]     GOTO $L_{\alpha-1}$                 {end RML}
```

There also are h kinds of RML-subprograms, since each label block, L, requires a different subprogram.

Therefore, if there is **P**'-computation which accepts the set X, then there is a clock sequence which generates a **P***-computation such that X is accepted.

###

Let us take an example.
Suppose **P**' is following, and α=3.

```
       V←V              .           V←V+1
       V←V+1            .     M     V←V+1
       GOTO L           .     M     V←V
  L    IF V≠0 GOTO M    .     M     V←V-1
  L    V←V-1            .           Y←V
  L    V←V+1            .
```

Pro 2.1

Then the temporary **P**' is :

```
                          .    $L_3$    V←V
       V←V                .             V←V+1
       V←V+1              .    $M_0$    V←V+1
       GOTO L             .             GOTO $M_3$
 $L_0$ IF V≠0 GOTO M      .    $M_1$    V←V
       GOTO $L_3$         .             GOTO $M_3$
 $L_1$ V←V-1              .    $M_2$    V←V-1
       GOTO $L_3$         .             GOTO $M_3$
 $L_2$ V←V+1              .    $M_3$    V←V
       GOTO $L_3$                       Y←V
```

Pro 2.2

So the **P*** is :

```
CL       C←C+1
         Z_1←C
         V←V
         V←V+1
         GOTO LC
L_0      IF V≠0 GOTO MC
         GOTO L_3
L_1      V←V-1
         GOTO L_3
L_2      V←V+1
         GOTO L_3
L_3      V←V
         V←V+1
M_0      V←V+1
         GOTO M_3
M_1      V←V
         GOTO M_3
M_2      V←V-1
         GOTO M_3
M_3      V←V
         Y←V
LC       Z_4←0
         IF Z_1≠0 GOTO RML
         GOTO CL
MC       Z_4←0
         IF Z_1≠0 GOTO RMM
         GOTO CL
[RML]    Z_1← Z_1-1
         Z_2←Z_2+1
         IF Z_1≠ 0 GOTO RL_2
         GOTO FL
[RL_2]   Z_1← Z_1-1
         Z_2←Z_2+1
```

```
           IF $Z_1 \neq 0$ GOTO $RL_3$
           GOTO FL
[$RL_3$]   $Z_1 \leftarrow Z_1 - 1$
           $Z_2 \leftarrow 0$
           $Z_4 \leftarrow Z_4 + 1$
           IF $Z_1 \neq 0$ GOTO RML
           GOTO FL
[FL]       $Z_1 \leftarrow Z_4$
           IF $Z_2 \neq 0$ GOTO $L'_1$
           GOTO $L_0$
[$L'_1$]   $Z_2 \leftarrow Z_2 - 1$
           IF $Z_2 \neq 0$ GOTO $L'_2$
           GOTO $L_1$
[$L'_2$]   GOTO $L_2$                    { end of RML }
[RMM]      $Z_1 \leftarrow Z - 1$
           $Z_2 \leftarrow Z_2 + 1$
           IF $Z_1 \neq 0$ GOTO $RM_2$
           GOTO FM
[$RM_2$]   $Z_1 \leftarrow Z_1 - 1$
           $Z_2 \leftarrow Z_2 + 1$
           IF $Z_1 \neq 0$ GOTO $RM_3$
           GOTO FM
[$RM_3$]   $Z_1 \leftarrow Z_1 - 1$
           $Z_2 \leftarrow 0$
           $Z_4 \leftarrow Z_4 + 1$
           IF $Z_1 \neq 0$ GOTO RMM
           GOTO FM
[FM]       $Z_1 \leftarrow Z_4$
           IF $Z_2 \neq 0$ GOTO $M'_1$
           GOTO $M_0$
[$M'_1$]   $Z_2 \leftarrow Z_2 - 1$
           IF $Z_2 \neq 0$ GOTO $M'_2$
```

```
                GOTO $M_1$
[$M'_2$]        GOTO $M_2$              {end of RMM}
```

Pro 2.3

Theorem 2.2 Given a set, X, accepted by an *S*-program, **P**, there is another *S*-program, **P***, which also accepts X, and each computation of **P*** is deterministic.

Proof As commented above, it suffices to show that **P*** accepts the same set as the version of **P'** which has constant indeterminacy. For each X, if there is a computation of **P'** which terminates on input X, there will be a value of the clock sequence for which **P*** also terminates on X. Since, eventually, arbitrarily large values of the clock sequence are generated by **P***, unless termination occurs, then **P*** accepts X. Conversely, if there is a **P*** computation accepting X, then **P'** will have an accepting computation based on the sequence of choices dictated by the appropriate clock sequence of **P***. Since no line label is repeated in **P***, each **P*** computation is deterministic.

###

2.2 Time Estimation

We consider the number of steps required by **P***(x) to fully simulate t steps of **P'**(x).

Lemma 2.1 A single call to the subprogram LC(Change L), where the values of Z_1 is z_1, may require $P_{TL}(z_1)=5z_1+2\alpha+2$ computation steps of **P***.

proof Let us go back to look at the subprograms:

There are $1+z_1$ steps in CL, $3z_1+[z_1/\alpha]+2\alpha+1$ steps in LC in the worst case, and $(4z_1+[z_1/\alpha]+2\alpha+2) \leq (5z_1+2\alpha+2)$. Thus the total number of steps in the worst case is at most: $P_{TL}(z_1)=5z_1+2\alpha+2$.

Here α is a fixed number, $P_{TL}(z_1)$ is a linear polynomial.

###

Theorem 2.3 Given an **NDCSP, P'**, there is a **DCSP, P***, which will simulate a t-step computation of **P'** in $O(\alpha^{2t})$ steps.

proof To simulate t "steps" of **NDCSP, P'**, the **DCSP, P*** must successively calculate for all clock sequences of length 1,2,...,t. Thus the value of Clock sequence C must run from 1 to $C(t)=(\alpha-1)\alpha^{t-1}+...+(\alpha-1)\alpha+(\alpha-1)$ (Unless, of course, **P*** halts for one such value of C).

Now, to simulate <u>1 step</u> of **P'** which is not of the form "GOTO L" requires <u>1 step</u> of **P***.

Suppose the step i to be simulated is "GOTO L" in **P'** then in **P*** it must be changed to "GOTO LC", and requires $P_{TL}(C(i))$ (C(i) is present clock sequence value) steps of **P*** by the lemma 2.1.

Thus, to simulate t steps of **P'**, the worst simulation case would involve simulating t successive GOTO statements. **P*** must cycle through all of the

Clock strings of α-length 1, then all of these of α-length 2,...,all of α-length t. Since for each t there are α^t such strings, the worst case is:

$$P_{TL}(1)\alpha + P_{TL}(2)\alpha^2 + \ldots + P_{TL}(C(t-1))\alpha^{t-1} + P_{TL}(C(t))\alpha^t \ .$$

Since $P_{TL}(C(t))\alpha^t = (5C(t)+2\alpha+2)\alpha^t = (5(\alpha^t-1)+2\alpha+2)\alpha^t$.

Therefore $T = O(\alpha^{2t})$.

###

CHAPTER 3

T.M.Accepts The Same Class Of Sets As NDCSP

In this chapter, we are going to prove that a general non-deterministic computation *S*-program, **P**, can be simulated by a deterministic Turing machine, **M**, and conversely. Note that the present formulation (including the use of a two-way infinite tape) follows [6]. Actually Turing's original development employed quintuples rather than quadruples. As we know from Chapter 2, it is necessary only to consider a deterministic computation *S*-program, **P***, because, a general *S*-program **P** can be simulated by a program **P***, which admits only deterministic computation.

3.1 The Flowchart of T.M. Simulator for DCSP

Since one **P***-computation step is specified by its snapshot, which contains the line number to be executed and the content of each register, it is convenient to consider two-way infinite multiple-tape Turing machines, and then to appeal to the standard result for simulating multiple-tapes by a single tape. This simulation is refered in [2,pp.116] in which the authors mention that the contents of a two-way infinite k-tapes T.M. and the position of the tapehead on each can be represented as a single tape with 2k tracks. Using this representation, the proof of Lemma 4.5.2 of [4], in fact, shows how to simulate any computation by a two-way infinite T.M. using only one tape.

We use $V_3, V_4, \ldots, V_{n+2}$ for, $x_1, x_2, \ldots, x_n$, the input variables of **P***, V_{n+3}, $V_{n+4}, \ldots,$ V_{n+p+2} for, z_1, $z_2, \ldots,$ z_p, the local variables of **P***, and V_{n+p+3} for, Y, the output variable of **P***. although it is possible to consider any ***S***-program language, we simplify the discussion to a "Tally-language" in which $V_j=0$, is represented by 1, and $V_j=m$ is represented by $1^{|V_j|+1}$=11...1 (m+1 times).

To construct the simulating Turing machine, we adopt the convention that the input string is placed on the first tape, tape-1; the number of executed line in **P*** is placed on the second tape, tape-2; the variable $V_j (3 \leq j \leq n+p+3)$ is initialized blank on the tape-j. At the end of a computation a n+p+3-tape **T.M.** is to leave its output on its first tape; the contents of the other tapes are ignored.

Before proceeding to the T.M. code, we describe the idea of the construction for the simulating T.M. There are n+p+3 tapes, of which tape-1 is reserved for input and output, only. Tape-2 records the number of the next ***S***-program line to be executed. Tape-3 through n+2 are reserved to receive the n inputs $V_3, V_4, \ldots, V_{n+2}$, which are the initial contents of the n input registers. Tape-(n+3) through n+p+2 are initialized with value 0, and later become the locations of values for the variable registers. Execution of ***S***-program instructions "$V \leftarrow V+1$" and "$V \leftarrow V-1$" can be accomplished by modifying one register tape, and incrementing tape-2 by 1, to record change of control. When the executed line is a GOTO, none of the input or local register tapes is altered, but tape-2 is erased and rewritten with the number of a line bearing the correct label. In this process, states of the TM. are used to remember the line numbers to be written on tape-2. The instruction "$V \leftarrow V$" results in incrementing tape-2 only. A flowchart which relates the various procedures follows. Note that the n initial inputs are all recorded on tape-1 initially as a concatenation.

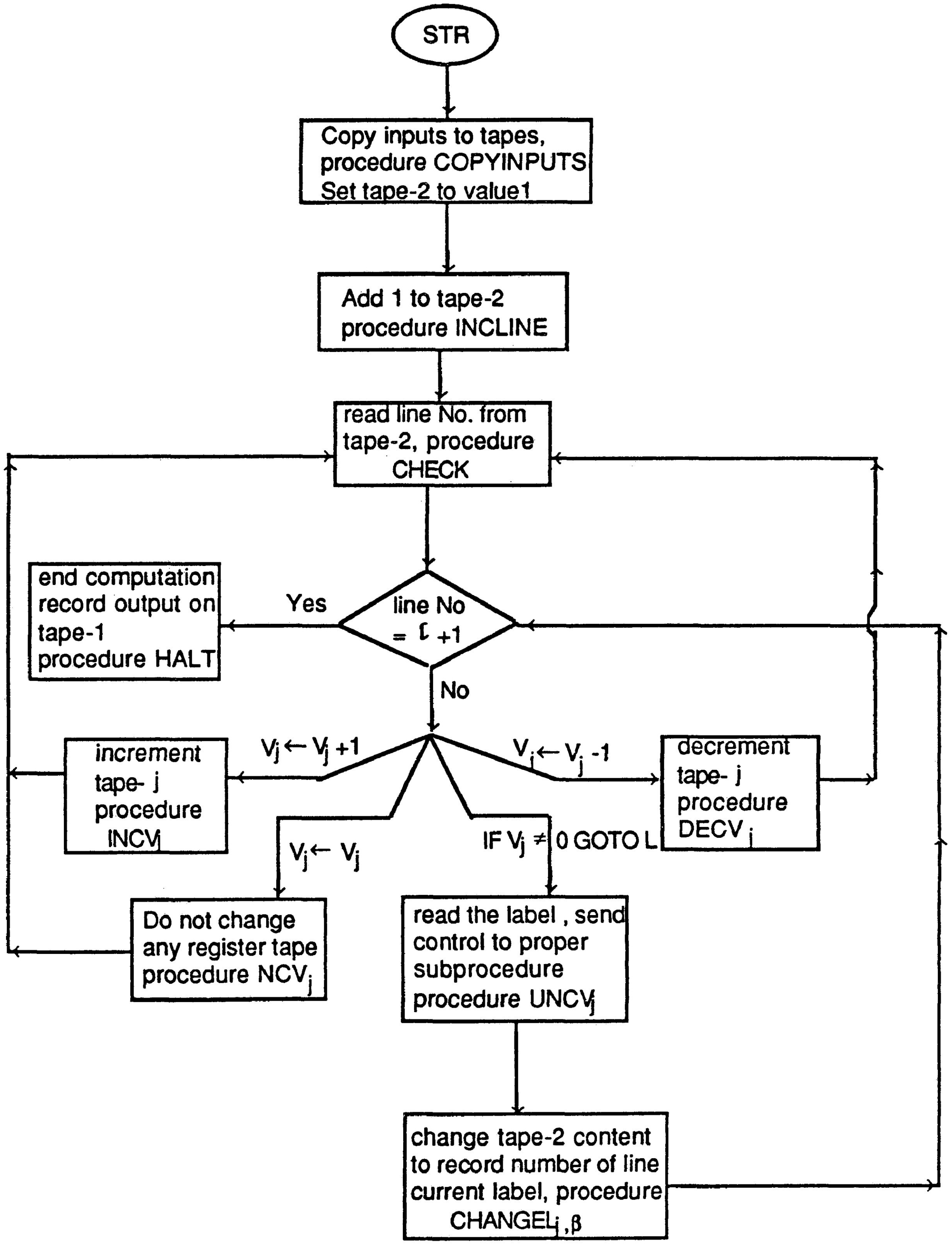
STR
Copy inputs to tapes, procedure COPYINPUTS Set tape-2 to value1
Add 1 to tape-2 procedure INCLINE
read line No. from tape-2, procedure CHECK
line No = ℓ +1
Yes
end computation record output on tape-1 procedure HALT
No
$V_j \leftarrow V_j + 1$
increment tape- j procedure INCVj
$V_j \leftarrow V_j - 1$
decrement tape- j procedure DECVj
$V_j \leftarrow V_j$
Do not change any register tape procedure NCVj
IF $V_j \neq 0$ GOTO L
read the label, send control to proper subprocedure procedure UNCVj
change tape-2 content to record number of line current label, procedure CHANGELj,β

Fig 3.1

3.2 Procedures of Simulation

First we initialize each tape as following:

tape-1	$\underline{B}\ 1^{\lvert V_3\rvert+1}\ B\ 1^{\lvert V_4\rvert+1}\ B \ldots B\ 1^{\lvert V_{2+n}\rvert+1}\ B^{@}$	{the input string is placed on the first tape}
tape-2	$B\ 1\ \underline{B}$	{number of executed line is place on the second tape}
tape-3	$\underline{B}$	{input variabletape}
⋮	⋮	
tape-(n+2)	$\underline{B}$	{input variable tape }
tape-(n+3)	$B\ 1\ \underline{B}$	{local variable tape }
	⋮	
tape-(n+p+2)	$B\ 1\ \underline{B}$	{local variable tape }
tape-α	$B\ 1\ \underline{B}$	{output variable tape}

Here α is equal to n+p+3, the underline "-" represents the head position on the each tape of the T.M..

In the following procedures, we use an α-tuple notation with superscripts, say $[B^{(1)}\ B^{(2)}\ \ldots\ B^{(\alpha)}]$ (α times B) to represent the α-tuple of symbols which occur at the present head positions of the α-tape T.M.. For example, $[B^{(1)}\ 1^{(2)}\ B^{(3)} \ldots B^{(\alpha)}]$ denotes that the head points to blank on the first tape, the head points at 1 on the second tape, and the head points to blank on the other tapes. The second α-tuple in each T.M. instruction may contain, besides alphabet symbols, also the symbols R and L which are used in the standard T.M. sence to indicate moves of one square right, or left, on the indicated tape. Our T.M. instructions will be of the quadruple form, using the symbols just defined and states as needed. Since states are used by the T.M. to remember what to do, we invent and use MNEMONIC symbols for states.

@ where: $V_3=X_1, V_4=X_2, \ldots, V_{n+2}=X_n$.

(1) **COPYINPUTS** - copy the inputs $V_3, V_4, \ldots, V_{n+2}$ to tape-3 until tape-n+2 (n times) in order. We use C_{iq} ($0 \le i \le n$, $0 \le q \le 2$) to represent states.

$$C_{10}[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{R}^{(1)}\mathbf{B}^{(2)}\mathbf{R}^{(3)}\mathbf{B}^{(4)}\ldots\mathbf{B}^{(\alpha)}]\,C_{11}$$

{start at blank for each tape, move the heads on tape-1 and tape-3 to the right, the rest are not to change.}

$$C_{11}[1^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\mathbf{B}^{(4)}\ldots\mathbf{B}^{(\alpha)}]\,[1^{(1)}\mathbf{B}^{(2)}1^{(3)}\mathbf{B}^{(4)}\ldots\mathbf{B}^{(\alpha)}]\,C_{12}$$

{read 1 on tape-1, write 1 on tape-3 }

$$C_{11}[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]\,C_{20}$$

{if the head on tape-1 points to a blank, go to state C_{20} which works on tape-1 and tape-4 }

$$C_{12}\;[1^{(1)}\mathbf{B}^{(2)}1^{(3)}\mathbf{B}^{(4)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{R}^{(1)}\mathbf{B}^{(2)}\mathbf{R}^{(3)}\mathbf{B}^{(4)}\ldots\mathbf{B}^{(\alpha)}]\,C_{11}$$

{if both heads on tape-1 and tape-3 point at 1, then move the heads on both tapes to the right, go back to state C_{11}}

. . .

. . .

{Then for the states C_{20}, C_{21}, C_{22} and C_{30}, the idea of operation is similar to C_{10}, C_{11}, C_{12} and C_{20}, so C_{30}, ..., C_{n0}, C_{n1}, C_{n2} and C_{n+10}}

$$C_{n0}[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(n+2)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{R}^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(n+1)}\mathbf{R}^{(n+2)}\mathbf{B}^{(n+3)}\ldots\mathbf{B}^{(\alpha)}]\,C_{n1}$$

$$C_{n1}[1^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(n+2)}\ldots\mathbf{B}^{(\alpha)}]\,[1^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(n+1)}1^{(n+2)}\mathbf{B}^{(n+3)}\ldots\mathbf{B}^{(\alpha)}]\,C_{n2}$$

$$C_{n1}[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(n+2)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(n+2)}\ldots\mathbf{B}^{(\alpha)}]\,L_0$$

{state L_0 is in the next procedure}

$C_{n2}\ [1^{(1)}B^{(2)}1^{(3)}B^{(4)}\ldots B^{(\alpha)}]\ [R^{(1)}B^{(2)}R^{(3)}B^{(4)}\ldots B^{(\alpha)}]\ C_{n1}$

{After we finish working on the tape-1 and tape-(n+2), the inputs will be copied from tape-1 to tape-3 to tape-(n+2)}

After procedure **COPYINPUTS**, the T.M. looks like:

tape-1 $B\ 1^{|V_3|+1}\ B\ 1^{|V_4|+1}\ B \ldots B\ 1^{|V_{2+n}|+1}\ \underline{B}$

tape-2 $B\ 1\ \underline{B}$

tape-3 $B\ 1^{|V_3|+1}\ \underline{B}$

tape-4 $B\ 1^{|V_4|+1}\ \underline{B}$

.
.
.

tape-(n+2) $B\ 1^{|V_{n+2}|+1}\ \underline{B}$

tape-(n+3) $B\ 1\ \underline{B}$ {local variable tape }

.
.
.

tape-(n+p+2) $B\ 1\ \underline{B}$ {local variable tape }

tape-α $B\ 1\ \underline{B}$ {output variable tape}

The next procedure will be started at state L_0.

(2) **INCLINE** - increase the number of the executed line by 1.

$L_0\ [B^{(1)}B^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ [B^{(1)}L^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ L_1$

{move the head on tape-2 to left}

$L_1\ [B^{(1)}1^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ [B^{(1)}L^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ L_1$

{read 1 on tape-2 until the first blank was found}

$L_1\ [B^{(1)}B^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ [B^{(1)}1^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ L_2$

{add 1 to the left end of tape-2

$$L_2[\mathbf{B}^{(1)}1^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}][\mathbf{B}^{(1)}\mathbf{L}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]R_0$$

{move the head on tape-2 to left, go to state R_0 which is in the procedure CHECK}

After execution of this procedure, the head on tape-2 is on the blank just to the left side of the string of 1's. In this case the T.M. looks like:

tape-1 $\mathbf{B}\ 1^{|V_3|+1}\ \mathbf{B}\ 1^{|V_4|+1}\ \mathbf{B}\ \ldots\ \mathbf{B}\ 1^{|V_{2+n}|+1}\ \underline{\mathbf{B}}$

tape-2 $\underline{\mathbf{B}}\ 1\ 1\ \mathbf{B}$

tape-3 $\mathbf{B}\ 1^{|V_3|+1}\ \underline{\mathbf{B}}$

tape-4 $\mathbf{B}\ 1^{|V_4|+1}\ \underline{\mathbf{B}}$

.
.
.

tape-(n+2) $\mathbf{B}\ 1^{|V_{n+2}|+1}\ \underline{\mathbf{B}}$

tape-(n+3) $\mathbf{B}\ 1\ \underline{\mathbf{B}}$ {local variable tape }

.
.
.

tape-(n+p+2) $\mathbf{B}\ 1\ \underline{\mathbf{B}}$ {local variable tape }

tape-α $\mathbf{B}\,1\,\underline{\mathbf{B}}$ {output variable tape}

The next procedure will be started in state R_0.

(3) **CHECK** - reads the line number on the tape-2. If it is ℓ+1, then the machine halts, here ℓ represents the length of the $\mathcal{S}$-program **P***.

$R_0\ [\mathbf{B}^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(\alpha)}]\ [\mathbf{B}^{(1)}\mathbf{R}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]\ R_0$ {move the head on tape-2 to right}

$R_0\ [\mathbf{B}^{(1)}1^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]\ [\mathbf{B}^{(1)}\mathbf{R}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]\,R_{01}$

{the first 1 is reached}

R_{01} $[\mathbf{B}^{(1)}1^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]$ $[\mathbf{B}^{(1)}\mathbf{R}^{(2)}\ldots\mathbf{B}^{(\alpha)}]$ R_1 {the second 1 is reached}

R_1 $[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(\alpha)}]$ $[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(\alpha)}]$ $F_0^{[1]}$ {in state R_1, if a block is read on tape-2 , then go to state $F_0^{[1]}$ which finds the instruction in line 1.}

R_1 $[\mathbf{B}^{(1)}1^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]$ $[\mathbf{B}^{(1)}\mathbf{R}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]$ R_2 {in state R_1, if a 1 is read on tape-2, then go to the state R_2, continue to read tape-2}

. .
. .
. .

R_k $[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(\alpha)}]$ $[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(\alpha)}]$ $F_0^{[k]}$ { in state R_k, if a blank is read then go to the state $F_0^{[k]}$ which corresponds to the kth line in P^*}

R_k $[\mathbf{B}^{(1)}1^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]$ $[\mathbf{B}^{(1)}\mathbf{R}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]$ R_{k+1} { $1 \le k \le \ell$} {in state R_k, if a 1 is read then go to state R_{k+1} and continue to read tape-2.}

. .
. .
. .

R_ℓ $[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(\alpha)}]$ $[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\ldots\mathbf{B}^{(\alpha)}]$ $F_0^{[\ell]}$

R_ℓ $[\mathbf{B}^{(1)}1^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]$ $[\mathbf{B}^{(1)}\mathbf{R}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]$ $R_{\ell+1}$

$R_{\ell+1}$ $[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]$ $[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]$ H_0 {the program execution halts, so go to state H_0 which is in procedure HALT}

$R_{\ell+1}$ $[\mathbf{B}^{(1)}1^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]$ $[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]$ H_0

After this operation, the head on tape-2 moves to the blank just to the right side of the string of 1's.

Now find the instruction on the kth line of program P^*. If the variable in the kth line is V_j, then the T.M works on the tape-j, here $3 \le j \le \alpha$, $1 \le k \le \ell$.

If the instruction is $V_j \leftarrow V_j+1$, we have subprocedure called $INCV_j$.

(4.1) **INCV$_j$**-for tape-j

$$F_0^{[k]}\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}1^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,F_{1,j}^{[k]}$$

{since the head on the tape-j, is immediately to the right of the last 1 and add 1 on tape-j }

$$F_{1,j}^{[k]}\,[\mathbf{B}^{(1)}\ldots 1^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}\mathbf{R}^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,L_0$$

{move the head on the tape-j to the right one square, go to the state L_0 which is in procedure INCLINE}

If the instruction in kth line is $V_j \leftarrow V_j - 1$, then we use subprocedure called DECV$_j$. The procedure checks whether $V_j \neq 0$, and takes action on the result of the check.

(4.2)**DECV$_j$** -for tape-j

$$F_0^{[k]}\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}\mathbf{L}^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,F_{1,j}^{[k]}$$

{move the head on the tape-j to the left one square }

$$F_{1,j}^{[k]}\,[\mathbf{B}^{(1)}\ldots 1^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}\mathbf{L}^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,F_{2,j}^{[k]}$$

{continue left past the rightmost 1 on tape-j}

$$F_{2,j}^{[k]}\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}\mathbf{R}^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,F_{3,j}^{[k]}$$

{ $V_i=0$, move the head one square to the right }

$$F_{2,j}^{[k]}\,[\mathbf{B}^{(1)}\ldots 1^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}\mathbf{R}^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,F_{4,j}^{[k]}$$

{there are two 1's, $V_i \neq 0$, move head one square to right}

$$F_{3,j}^{[k]}\,[\mathbf{B}^{(1)}\ldots 1^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}\mathbf{R}^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,L_0$$

{move the head to the right on the tape-j, then go to the state L_0, leaves tape-j unchanged}

$$F_{4,j}^{[k]}\,[\mathbf{B}^{(1)}\ldots 1^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}\mathbf{B}^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,L_0$$

{change rightmost 1 to blank on tape-j, then go to state L_0}

If the instruction is $V_j \leftarrow V_j$, we have subprocedure called NCV_j.

(4.3) **NCV_j** - for tape-j

$$F_0^{[k]}\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}\mathbf{B}^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,L_0$$

{leaves tape-j unchanged}

If the instruction in kth line is "IF $V_j \neq 0$ GOTO L", then we use a subprocedure called $UNCV_j$. The procedure first checks whether $V_j \neq 0$, and then takes appropriate action.

(4.4) **$UNCV_j$** -for tape-j

$$F_0^{[k]}\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}\mathbf{L}^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,F_{1,j}^{[k]}$$

{head on the tape-j moves to the left one square }

$$F_{1,j}^{[k]}\,[\mathbf{B}^{(1)}\ldots 1^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}\mathbf{L}^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,F_{2,j}^{[k]}$$

{continue left past right most 1}

$$F_{2,j}^{[k]}\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}\mathbf{R}^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,F_{3,j}^{[k]}$$

{$V_j=0$, move head on tape-j to right one square}

$$F_{2,j}^{[k]}\,[\mathbf{B}^{(1)}\ldots 1^{(j)}\ldots\mathbf{B}^{(\alpha)}]\,[\mathbf{B}^{(1)}\ldots\mathbf{B}^{(j-1)}\mathbf{R}^{(j)}\mathbf{B}^{(j+1)}\ldots\mathbf{B}^{(\alpha)}]\,F_{4,j}^{[k]}$$

{$V_j \neq 0$, the head on tape-j goes back to right one square}

$F^{[k]}_{3,j}[B^{(1)} \ldots 1^{(j)} \ldots B^{(\alpha)}]\,[B^{(1)} \ldots B^{(j-1)} R^{(j)} B^{(j+1)} \ldots B^{(\alpha)}]\,L_0$

{$V_i=0$, go to the state L_0 leave tape-j unchanged}

$F^{[k]}_{4,j}[B^{(1)} \ldots 1^{(j)} \ldots B^{(\alpha)}]\,[B^{(1)} \ldots B^{(j-1)} R^{(j)} B^{(j+1)} \ldots B^{(\alpha)}]\,N_{0,\beta}$

{$V_i \neq 0$, go to state N_0 which is in the procedure CHANGEL$_{j,\beta}$, β is the number of the line with label L, $1 \leq \beta \leq 1$ = the(constant) multiplicity of line labels in the *S*-program}

(5) **CHANGEL$_{j,\beta}$**- for the instruction of "IF $V_j \neq 0$ GOTO L", If there are m different labels in **P**, then there will be m versions of this procedure, each with its own unique value of β.

$N_{0,\beta}[B^{(1)} B^{(2)} B^{(3)} \ldots B^{(\alpha)}]\,[B^{(1)} L^{(2)} B^{(3)} \ldots B^{(\alpha)}]\,N_{1,\beta}$

{move the head on tape-2 one square left }

$N_{1,\beta}[B^{(1)} 1^{(2)} B^{(3)} \ldots B^{(\alpha)}]\,[B^{(1)} B^{(2)} B^{(3)} \ldots B^{(\alpha)}]\,N_{2,\beta}$

{erase one 1 on tape-2}

$N_{2,\beta}[B^{(1)} B^{(2)} B^{(3)} \ldots B^{(\alpha)}]\,[B^{(1)} L^{(2)} B^{(3)} \ldots B^{(\alpha)}]\,N_{1,\beta}$

{continuing move head on tape-2 to left}

$N_{1,\beta}[B^{(1)} B^{(2)} B^{(3)} \ldots B^{(\alpha)}]\,[B^{(1)} B^{(2)} B^{(3)} \ldots B^{(\alpha)}]\,N^{[L]}_{\beta,0}$

{finish by erasing all tallys, go to the state $N_{\beta,0}{}^{[L]}$, move the head to left in order to print $\beta+1$ tallys on tape-2}

$N^{[L]}_{\beta,0}\ [B^{(1)} B^{(2)} \ldots B^{(\alpha)}]\,[B^{(1)} 1^{(2)} B^{(3)} \ldots B^{(\alpha)}]\,N^{[L]}_{\beta,1}$

{print $\beta+1$ tally on tape-2 for the label L, there will be m kinds of this instruction if the program

P* has m difference labels}

$N^{[L]}_{\beta,1}\ [B^{(1)}1^{(2)}\ldots B^{(\alpha)}]\ [B^{(1)}L^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ N^{[L]}_{\beta,2}$

{after writing 1 in one square of tape-2, move the head to the left again}

$N^{[L]}_{\beta,2}\ [B^{(1)}B^{(2)}\ldots B^{(\alpha)}]\ [B^{(1)}1^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ N^{[L]}_{\beta,3}$

$N^{[L]}_{\beta,3}\ [B^{(1)}1^{(2)}\ldots B^{(\alpha)}]\ [B^{(1)}L^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ N^{[L]}_{\beta,4}$

. .
. .
. .

$N^{[L]}_{\beta,2\beta}\ [B^{(1)}B^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ [B^{(1)}1^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ N^{[L]}_{\beta,2\beta+1}$

$N^{[L]}_{\beta,2\beta+1}\ [B^{(1)}1^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ [B^{(1)}L^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ N^{[L]}_{\beta,2(\beta+1)}$

$N^{[L]}_{\beta,2\beta+2}\ [B^{(1)}B^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ [B^{(1)}B^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ R_0$

{state R_0 is in the procedure $CHECK_{j,\beta}$}

After this operation, the head on tape-2 is on the end left of the string

(6) **HALT**- if the program halts, copy the last tape, tape-α, to the tape-1, there are two steps:

(6.1)-erase the tape-1

$H_0\ [B^{(1)}B^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ [L^{(1)}B^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ H_1$

{move the head on tape-1 from right to left one square}

$H_1\ [1^{(1)}B^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ [B^{(1)}B^{(2)}B^{(3)}\ldots B^{(\alpha)}]\ H_0$

{erase 1 on tape-1 of the right end.}

$H_1[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}][\mathbf{L}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]H_2$

{reach another blank, move the head on tape-1 to left again. After this it erased one input register}

$H_2[1^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}][\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]H_1$

{start to erase another input value}

$H_2[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}][\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]H_3$

{until the first double blanks reached}

(6.2) COPY- copy tape-α to tape-1

$H_3[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}][\mathbf{R}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{L}^{(\alpha)}]H_4$

{move the head on tape-1 to right, and the head on tape-α to left}

$H_4[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots 1^{(\alpha)}][1^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots 1^{(\alpha)}]H_5$

{copy 1 from tape-α to tape-1}

$H_4[\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}][\mathbf{B}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{B}^{(\alpha)}]H_6$

{finish copy }

$H_5[1^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots 1^{(\alpha)}][\mathbf{R}^{(1)}\mathbf{B}^{(2)}\mathbf{B}^{(3)}\ldots\mathbf{L}^{(\alpha)}]H_4$

{continually copy}

Here, the state H_6 represents a "halt state", and does not appear in any other instructions.

From the procedures above we immediately get the following:

Theorem 3.1 Given a **DCSP**, **P***, with n inputs registers, there is a deterministic Turing Machine, **M**, which accepts the same set of n-tuples as **P***.

We now look at the number of steps taken in each procedure above.

a). Initialization configuration

T.M., **M**, takes one step to initialize the inputs register on tape-1.

b). In COPYINPUTS

There are $3|x_i|+1$ steps from state C_{10} to C_{20}, thus **M** takes totally $3\sum_{i=1}^{n}|x_i|+n$ steps in this case.

c). In INCLINE

There are $(3+L)$ step in the worst case.

d). In CHECK

There are 3 steps from state R_0 to R_1, and there are $(L+1)$ steps from state R_1 to H_0 in the worst case.

Thus $\text{step(CHECK)}=L+4$ in the worst case.

e). It is very easy to find out:

$\text{step}(\text{INCV}_j)=2$

$\text{step}(\text{DECV}_j)=4$

$\text{step}(\text{NCV}_j)=1$

$\text{step}(\text{UNCV}_j)=4$

f). In $\text{CHANGEL}_{j,\beta}$

There are $2L+2$ steps from state $N_{0,\beta}$ to $N_{\beta,0}^{[L]}$ in the worst case, and there are $2(L+1)+1$ steps from state $N_{\beta,0}^{[L]}$ to R_0 in worst case.

Thus $\text{step}(\text{CHANGEL}_{j,\beta})=4L+5$ in the worst case.

g). In HALT

There are $2\sum_{i=1}^{n}|x_i|+n+2$ steps from state H_0 to H_3 in which all inputs are erased.

And there are $2|Y|+2$ steps from state H_3 to H_6 in which output is copied to tape-1.

Thus $\text{step(HALT)}=2\sum_{i=1}^{n}|x_i|+|Y|+n+4$.

Corollary 3.1 For a t-step deterministic computation *S*-program **P***, there is a T.M., **M**, which takes O(t) steps to simulate **P***.

Proof First **M** would intialize the inputs and copy those inputs to the individual input register tape, that requires:

$$1+3\sum_{i=1}^{n}|x_i|+n$$

And the worst simulation case would involve simulating t successive GOTO statements and require:

$$t(\text{step}(\text{UNCV}_j)+\text{step}(\text{CHANGEL}_{j,\beta})).$$

If **P*** halts at t step, then at this case **M** takes:

$$2\sum_{i=1}^{n}|x_i|+|Y|+n+4 \text{ steps}$$

Thus, simulating a t-step computation of **P*** in the worst case, **M** would require:

$$t(\text{step}(\text{UNCV}_j)+\text{step}(\text{CHANGEL}_{j,\beta}))+3\sum_{i=1}^{n}|x_i|+n+1+2\sum_{i=1}^{n}|x_i|+|Y|+n+4.$$

$$= t(4+4\ell+5)+5\sum_{i=1}^{n}|x_i|+|Y|+2n+5$$

$$= O(t).$$ ###

Corollary 3.2 For t-steps of a non-deterministic computation *S*-program, **P**, there is a T.M., **M**, which takes $O(\alpha^{2t})$ steps to simulate **P**, here α is indeterminacy of **P**.

Proof From Lemma 1.3, we know that a t-step **P**-computation can be a simulated by (t+c)-step **P'**-computation, here c is some constant, and each label block has the same indeterminacy α in **P'**.

And from Theorem 2.3 a (t+c)-steps of a **P'**-computation can be simulated by $O(\alpha^{2(t+c)})$-steps, which is $O(\alpha^{2t})$-step **P***-computation.

Finally from Corollary 3.1 $O(\alpha^{2t})$-step **P***-computation can be simulated by a $O(O(\alpha^{2t}))$-step, which is $O(\alpha^{2t})$-step, **M**-computation of a Turing machine.

Therefore a t-step **P**-computation of ***S***-program can be simulated by a $O(\alpha^{2t})$-step **M**-computation of Turing machine.

###

3.3. A General *S*-program Accepts the Same Set as a T.M.

We now show how to find a general non-deterministic computation ***S***-program for any given a NDTM (Non-deterministic Turing machine[2]).

For simplicity we consider quintuple Turing machines instead of quadruples, because a quadruple Turing machine can be simulated by a quintuple Turing machine [2,pp.101]. There are two kinds of quintuples:

$$q_i \quad s_j \quad s_k \quad R \quad q_l$$

$$q_i \quad s_j \quad s_k \quad L \quad q_l$$

We want to construct a program **P** in the language ***S*** which simulates NDTM, **NM**.

Let a quintuple non-deterministic tally Turing machine be **NM** with states $q_1, q_2, \ldots, q_m$, and alphabet {0,1}, here 0 represents the blank, **B**.

P will simulate **NM** by using the numbers in base 2 to represent strings(on the NDTM tape). The tape configuration at a given stage in the computation by **NM** will be encoded by **P** using three numbers stored in the registers L, H, and R. The value of H will be the numerical value of symbol(0 or 1) currently being scanned by the **NM**'s head. The value of L will be a number which represents in base 2 a string of symbols({0,1}) which begins with the leftmost 1 to the current head position and ends at the square just left of the head. The value of R represents in a similar manner the string of symbols to the right of the head, ending with the rightmost 1 on the tape. Note that one or both of L and R may be 0.

For example, consider the tape configuration of **NM** :

```
...0001111011100...
           ↑
           q3
```

Here $H=1$;
$L=1*2^4+1*2^3+1*2^2+1*2^1+0*2^0=30$;
$R=1*2^1+1*2^0=3$;

The program **P** will consist of three parts:

BEGINNING
MIDDLE
END

BEGINNING - Suppose the initial tape of **NM** is :

$$\mathbf{B}\ x_1\ \mathbf{B}\ x_2\ \mathbf{B}\ \ldots\ \mathbf{B}\ x_n\ \mathbf{B}$$

where the numbers $x_1, x_2, \ldots, x_n$ are represented by Tally strings(see section 3.2). Thus the part **BEGINNING** has the initial value of L,H,R:

```
L ← 0
H ← 0
R ← $CONCAT_2^{(2n-1)}(x_1,0,x_2,0,\ldots,0,x_n)$
```

here for given strings $u_1,u_2,\ldots,u_n \in A^*$, $CONCAT_m^{(n)}(u_1,u_2,\ldots,u_n)$ is simply the string obtained by placing the string $u_1,u_2,\ldots,u_n$ one after the other.

MIDDLE - this part will simulate **NM** in a step-by-step "interpretative" manner.

Associate with each state q_i a label A_i and with each state-symbol pair (q_i,j) a label B_{ij} $(j=0,1)$. For each label A_i ,place the following pairs of lines, in order i=1,2,..,m (for definitness, since the order does not matter) at the beginning of the *S*-program **MIDDLE**.

```
[$A_i$]    IF H=0 GOTO $B_{i0}$
           IF H=1 GOTO $B_{i1}$
```

If **NM** contains the quintuple q_i j k R q_l (j,k=0,1), then we introduce the block of instructions

```
[$B_{ij}$]   H ← 0                          {k=0}
[$B_{ij}$]   H ← 1                          {k=1}
           L ← $CONCAT_2$(L,H)
           H ← $LTEND_2$(R)
           R ← $LTRUNC_2$(R)
           GOTO $A_l$
```

If **NM** contains the quintuple q_i j k L q_ℓ (j,k=0,1), then we introduce the block of instructions

```
[B_ij]    H ← 0                      {k=0}
[B_ij]    H ← 1                      {k=1}
          R ← CONCAT_2(H,R)
          H ← RTEND_2(L)
          L ← RTRUNC_2(L)
          GOTO A_ℓ
```

If there is no quintuple in **NM** beginning q_i t (t≠0,1), we introduce the block

```
[B_it]    GOTO END
```

Finally, the part **END** of *P* can be taken simply to be

```
          Z ← CONCAT_2^(3) (L,H,R)
          Y ← Z
```

where functions (see [2])

1. $RTEND_2(L)$ gives the binary code for the rightmost symbol of a given word W_L when L is the binary code for W_L;

2. $LTEND_2(L)$ gives the binary code for the leftmost symbol of a given word W_L when L is the binary code for W_L;

3. $RTRUNC_2(L)$ gives the binary code for the result of removing the rightmost symbol from a given nonempty word W_L when L is the binary code for W_L;

4. $LTRUNC_2(L)$ gives the binary code for the result of removing the leftmost symbol from a given nonempty word W_L when L is

the binary code for W_L;

We have now completed the description of the program **P** which simulates the NDTM, **NM**. This gives us the proof of the following theorem.

Theorem 3.2 Given any non-deterministic quintuple Turing machine, there is a general *S*-program which accepts the same set as this Turing machine.

Summarizing results from Chapter one to this section, we observe the consequence as shown in the following:

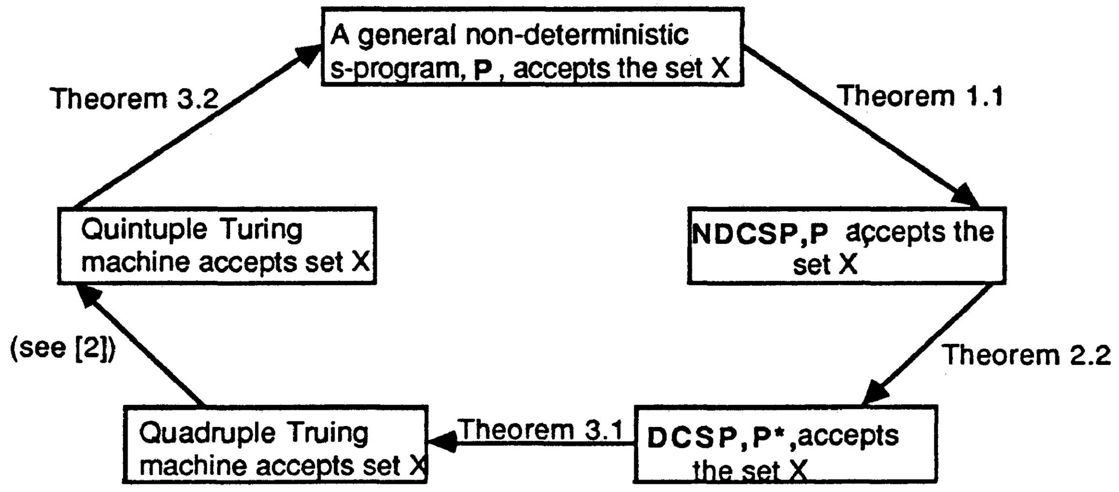
A general non-deterministic s-program, P, accepts the set X
Theorem 1.1
NDCSP,P accepts the set X
Theorem 2.2
DCSP,P*,accepts the set X
Theorem 3.1
Quadruple Truing machine accepts set X
(see [2])
Quintuple Turing machine accepts set X
Theorem 3.2

Fig 3.2

In attempting to compute the time requirement for an *S*-program to simulate a T.M. computation we encounter a limitation of the first form of *S*-programs discussed in reference [2], Consider the problem of simulating a single move of the T.M. using the program of Theorem 3.2. Of the several string manipulation operations, CONCAT, LTEND, RTEND, LTRUNC and RTRUNC, even the simplest, RTEND, involves evaluation of the remainder function, base 2, because we are working with numbers and must use numerical codes for the argument strings. As we saw in Chapter 2, the time required by an *S*-program to perform the remainder operation is linear in the value of its argument. Since the *S*-program must manipulate entire numerical register contents in simulating each T.M. step, and the time required is proportional to the values of these registers (here, in general, exponential in the lengths of the registers) the use of an *S*-program to simulate a T.M. would seem quite wasteful.

The limitation just discussed is imposed by the *S*-operations "$V \leftarrow V+1$" and "$V \leftarrow V-1$" which, in effect, force the *S*-program to work in a tally mode. In manipulating numerical codes for words of a T.M. language this is wasteful. Of course, in what we have done in Chapter 3, the argument of waste is not really applicable because, for a tally language the length of a word and its value coincide. However, this is not the correct way out of the difficulty.

The correct way to simulate even a tally language T.M. is to use Davis' idea of a symbol manipulating *S*-program, which we will do in Chapter 5. The operations "$V \leftarrow V+1$" and "$V \leftarrow V-1$" are then replaced by operations "$V \leftarrow aV$", which adds a new symbol to the left end of the string in V, and "$V \leftarrow V^{-}$", which deletes the rightmost symbol of the string in V. Using these operations it is easy to see that the string manipulation functions CONCAT, LTEND, RTEND, LTRUNC and RTRUNC (as distinct from their primitive recursive counterparts performed on numerical codes) can all be carried at in times linear in the lengths of the argument strings. For example, LTEND(x), is performed by the S_1-program.

```
[L]    Y←X
       X←X⁻
       IF X≠0 GOTO L
```

which requires $3(|x|-1)+2=3|x|-1$ steps to complete.

CHAPTER 4

Universal NDCSP

In the previous chapters, we discussed the special **NDCSP** in which each label occurs α times, and each label occurs only in one label block. We now can reduce the α to 2, to provide the most direct modification of the universal deterministic $\mathcal{S}$-program, of reference [2], to make it non-deterministic.

4.1 Algorithm for Binary Condition-AFBC

Suppose an **NDCSP, P'**, contains condition instruction "IF V≠0 GOTO L". If the condition is true in the "IF" statement, α of possibilities will exist, since there are α L statements. After executing one of L statements, the next statement to be executed may again be one statement, or one of α choices. The above condition is best illustrated by a α branching tree; the case α = 4 is shown below.

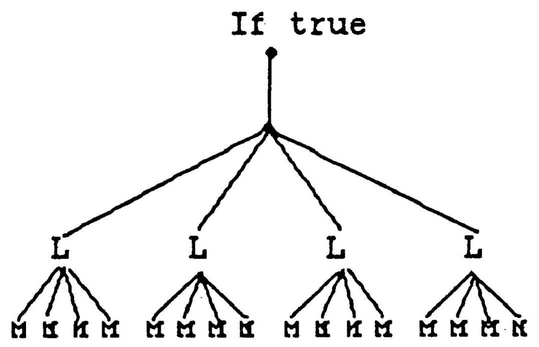

Fig 4.1

It is easy to reduce the value of α to 2, as shown in the next algorithm.

```
Step 0   Replace any P' line of the form
               IF (condition) GOTO L
         by    IF (condition) GOTO $A_1$

Step 1   Add these lines in blocks of 2.

               [$A_1$] GOTO $L_1$
               [$A_1$] GOTO $A_2$

               [$A_2$] GOTO $L_2$
               [$A_2$] GOTO $A_3$
                      .
                      .
                      .
               [$A_\alpha$] GOTO $L_\alpha$
               [$A_\alpha$] GOTO $L_\alpha$

Step 2   Replace the original L-block by

               [$L_1$]   ...          {first line of the L-block,
               [$L_1$]   ...          repeated twice}
                     GOTO M

               [$L_2$]   ...          {second line of the L-block,
               [$L_2$]   ...          repeated twice}
                     GOTO M
                      .
                      .
                      .
               [$L_\alpha$]   ...          {$\alpha$th line of the L-block,
               [$L_\alpha$]   ...          repeated twice}

               [M]   V←V
               [M]   V←V
```

In algorithms **AFI**(Chapter 1) and **AFBC** it is mainly the GOTO statements of each corresponding program which are changed. When we come to analyse the simulation steps, the GOTO statement usually brings out the worst case. Thus we consider how many steps would be taken for simulating a one step GOTO statement of **P'** by **P"**(lemma 4.1). Since there are α steps taken by **P'** to simulate a one step GOTO statement of **P**(corollary 1.1 Chapter 1), we have the result of Lemma 4.1. We will get the results of Corollary 4.1 and Corollary 4.2 in later sections of this chapter.

Lemma 4.1 The binary *S*-program, **P"**, takes $\alpha+3$ steps to simulate a one step GOTO statement of **P'** in the worst case.

Proof For a one step "GOTO L" statement of **P'**, by **AFBC**, steps 0, 1 and 2, **P"** takes $\alpha+3$ steps in the worst case.

###

Lemma 4.2 A binary *S*-program, **P"**, takes $2\alpha+3$ steps to simulate a one step GOTO statement of a general *S*-program, **P**.

proof The result immediately follows Corollary 1.1 and Lemma 4.1.

###

Theorem 4.1 Given an *S*-program, **P'**, of constant multiplicity, α, there is another, **P"**, of constant multiplicity 2, which accepts the same set as **P'** does.

Proof*(outline)* The technique for a formal proof of Theorem 4.1 is essentially the same as that for Lemma 1.3. We establish a (2-valued) correspondence relation, g, between lines of **P'** and lines of **P"**. A non-GOTO line corresponds to itself, and the kth line of on L-block corresponds to the pair of lines labelled L_k. One then shows, in the same manner as the proof of lemma 1.3, that there is a **P**'-computation leading to snapshot (i,σ) if, and only if, there is a **P**"-computation leading to $(g(i),\sigma)$.

4.2 Universal BNDCSP

In this and the following section we give two constructions of a universal *S*-program (which in our terms includes both non-deterministic and deterministic). The first construction uses the idea of a binary *S*-program developed in section 4.1, and varies little from the universal deterministic program of [2]. It is worth noting that both of our universal programs contain only one duplicated line label. Every general non-deterministic computation *S*-program can be simulated by **NDCSP** in which all similar labels occur in one block, and every **NDCSP** can be simulated by **BNDCSP** in which there are only two labels in the each block. Thus, in our first construction, we need only to give the universal program for every **BNCDSP**.

The non-deterministic computation universal program U_{UNSP1} is as follows. For definitions of the several primitive recursive functions used, and to follow the universal program itself, the reader may wish to consult [2,pp.58].

$$Z \leftarrow X_{n+1}+1$$

$$S \leftarrow \prod_{i=1}^{n} (P_{2i})^{X_1}$$

$$K \leftarrow 1$$

```
[C]    IF $K=Lt(Z)+1 \vee K=0$ GOTO F
       $U \leftarrow r((Z)_k)$
       $P \leftarrow P_{r(U)+1}$
       IF $\ell(U)=0$ GOTO N
       IF $\ell(U)=1$ GOTO A
       IF $\sim(P|S)$ GOTO N
       IF $\ell(U)=2$ GOTO M
       GOTO B
[B]    $K \leftarrow \min_{i \le Lt(Z)} [\ell((Z)_i)+2 = \ell(U)]$
[B]    $K \leftarrow \min_{i \le Lt(Z)} [\ell((Z)_i)+2 = \ell(U)]+1$
       GOTO C
[M]    $S \leftarrow [S/P]$
       GOTO N
[A]    $S \leftarrow S*P$
[N]    $K \leftarrow K+1$
       GOTO C
[F]    $Y \leftarrow (S)_1$
```

Pro.4.1

From **Pro 4.1** we have the Theorem as follows:

Theorem 4.2 There is a universal *S*-program(for both non-deterministic and deterministic computation) which contains only one repeated line label.

Of course the result is best possible, if we wish to permit non-deterministic computation. Since each non-deterministic computation can be simulated by a deterministic one, there is another universal *S*-program with no repeated labels, i.e. the one in reference [2].

As we discussed earlier in this chapter, we now compute how many steps a universal ***S***-program, U_{UNSP1}, takes to simulate a general ***S***-program, **P**, by using the "middleman" programs **P'** and **P"**.

Corollary 4.1 A universal ***S***-program, U_{UNSP1}, takes $2\alpha+16$ steps to simulate a one step GOTO statement of a general ***S***-program, **P**, by using program **P'** and **P"** if we count each primitive recursive line as "1-step" in U_{UNSP1}.

Proof In **Pro.4.1**, there are 13 steps to simulate a one step GOTO statement of **P"** in the worst case. And there are $2\alpha+3$ steps to simulate a one step GOTO statement of **P** by **P"** by Lemma 4.2. Thus a universal ***S***-program takes $2\alpha+16$ steps to simulate one step GOTO statement of **P**.

###

4.3. Universal *S*-program, Second Construction

By going to a little more trouble, it is possible to construct a universal ***S***-program which does not depend on any of the special reductions used in the last section.

In short, when the universal program encounters a GOTO to a labelled line, we can cause it to compute the number, μ, of occurrences of that label in the program being simulated, to seek the first(topmost) occurrence of that labelled line and (non-deterministically) either select that occurrence, or go into a loop which produces the second line of that label, etc.

For a "GOTO L" statement, we compute the following functions:

$$\mu = \sum_{i=1}^{Lt(Z)} \alpha(|\mathfrak{l}((Z)_i+2 - \mathfrak{l}(U)|)$$ {here α is a primitive recursive function in [2]}

$$LST(Z,j) = \min_{j<i\leq Lt(Z)} [\mathfrak{l}((Z)_i)+2 = \mathfrak{l}(U)]$$

μ gives the total number of occurrences of label L in the *S*-program, **P**. Note that α, which is mentioned in previous two sections, is greater than μ, i.e. if μ is the indeterminacy of program **P** then $\alpha=\mu+1$.

LST(Z,j) gives the least line number of the line labelled L between the jth line and the Lt(Z)th line.

The loop which does the non-deterministic selection of the GOTO destination is:

```
B    K ← LST(Z,I)          {I initialized 0, k now gives
     GOTO D                the least line number of the
                           line labelled L between the
                           Ith line and the last one }
D    GOTO C                {Either go back to C with the
D    μ ← μ-1               "present value" of K, and
     IF μ=0 GOTO C         execute that line, or compute
     I ← K+1               a new line number and go to B
     GOTO B                loop}
```

Placing these subprograms in the universal program yields a new universal program, U_{UNSP2} as follows, and a second proof of Theorem 4.2.

$$Z \leftarrow X_{n+1}+1$$

$$S \leftarrow \prod_{i=1}^{n} (P_{2i})^{X_i}$$

$$K \leftarrow 1$$

```
[C]   IF $K=Lt(Z)+1 \vee K=0$ GOTO F
      $U \leftarrow r((Z)_k)$
      $P \leftarrow P_{r(U)+1}$
      IF $\ell(U)=0$ GOTO N
      IF $\ell(U)=1$ GOTO A
      IF $\sim(P|S)$ GOTO N
      IF $\ell(U)=2$ GOTO M
      $\mu \leftarrow \sum_{i=1}^{Lt(Z)} \alpha(|\ell((Z)_i)+2-\ell(U)|)$
[B]   $K \leftarrow LST(Z,I)$
      GOTO D
[D]   GOTO C
[D]   $\mu \leftarrow \mu-1$
      IF $\mu=0$ GOTO C
      $I \leftarrow K+1$
      GOTO B
[M]   $S \leftarrow [S/P]$
      GOTO N
[A]   $S \leftarrow S*P$
[N]   $K \leftarrow K+1$
      GOTO C
[F]   $Y \leftarrow (S)_1$
```

Pro.4.2

Corollary 4.2 A universal *S*-program, U_{UNSP2}, takes $6\mu+11$ steps to simulate a one step GOTO statement of a general *S*-program, **P**, without using program **P'** and **P"** if we count each primitive recursive line as "1-step" in U_{UNSP2}.

Proof For a one step GOTO statement of **P**, in U_{UNSP2} the loop which does the non-deterministic selection of the GOTO destination in the worst case takes 6μ steps, Thus U_{UNSP2}

takes $6\mu+11$ steps to simulate a one step GOTO statement of **P** in the worst case.

###

CHAPTER 5

NP Completeness

5.1. Cook's Theorem for the Programming Language $\mathcal{S}_n$.

The programming language $\mathcal{S}_n$ is specifically designed for string calculation on an alphabet A of n symbols. It is supplied with the same input, output and local variables as $\mathcal{S}$, except that we now use them as having value in the set of all words on the alphabet A and we allow a unique null word as 0(empty word). The instruction types are as following (see [2,pp.77]).

Instruction	Interpretation
$V \leftarrow \sigma V$	For each symbol σ in the alphabet place the symbol σ to the left of the string which is the value of V.
$V \leftarrow V^-$	Delete the final symbol of the string which is the value of V. If the value of V is 0, leave it unchanged.
$V \leftarrow V$ @	a non-operable line, control passes to next line.
If v ENDS σ GOTO L	For each symbol σ in the alphabet A and each label L, If the value of the string in register V ends in the symbol σ, execute next some instruction labeled L; otherwise proceed to the next instruction.

Table 5.1

@ Not a basic instruction in [2]. We include it for convenience although its effect can be achieved under the rules of [2] by other means, e.g. $V \leftarrow V^-$ for some register V which is always void.

The following conventions apply to a set of lines which form an $\mathcal{S}_n$-program, P .

(1). The alphabet of the language $\mathcal{S}_n$ is $A=\{a_1,a_2,...,a_n\}$ and we choose another symbol a_0, to represent a "blank", to be used in a manner explained below. A* is, as usual, the set of words on A.

(2). The set of variables(register names) which occur in **P** is $\{V_1,V_2,...,V_r\}$.

(3). **P** is of length ℓ.

(4). **P** has the different label names: L_1 of multiplicity μ_1, L_2 of multiplicity μ_2,..., L_m of multiplicity μ_m.

In addition, we define a partition of the numbers of only the set of labelled lines, as follows:
$h \in H_{L_\beta}$ if and only if line h is labelled by L_β(β=1,2,...,m).

For use in calculating the atom count and clause count of standard CNF(conjunctive normal form), S, we give following lemma.

Lemma 5.1 Suppose $A_1,A_2,...,A_m$, each is a disjunction of literals[@], and the total literal count of A_i is Literal(A_i). $B_1,B_2,...,B_m$, each is a CNF, and the total clause count of B_i is Clause(B_i), the total atom count of B_i is Atom(B_i). Then we can calculate the atom count and the clause count of the CNF reduction of the formula τ:

@ The present development follows [5]

$$\tau = (A_1 \supset B_1) \wedge (A_2 \supset B_2) \wedge \ldots \wedge (A_m \supset B_m)$$

by the formulas

$$\text{Clause}(\tau) = \sum_{i=1}^{m} \text{Clause}(B_i);$$

$$\text{Atom}(\tau) = \sum_{i=1}^{m} (\text{Literal}(A_i)\text{Clause}(B_i) + \text{Atom}(B_i)).$$

Proof Note that in each $(A_i \supset B_i)$:

$$\text{Clause}(A_i \supset B_i) = \text{Clause}(B_i)$$

$$\text{Atom}(A_i \supset B_i) = \text{Literal}(A_i)\text{Clause}(B_i) + \text{Atom}(B_i).$$

Therefore

$$\text{Atom}(\tau) = \sum_{i=1}^{m} \text{Atom}(A_i \supset B_i)$$

$$= \sum_{i=1}^{m} (\text{Literal}(A_i)\text{Clause}(B_i) + \text{Atom}(B_i))$$

$$\text{Clause}(\tau) = \sum_{i=1}^{m} \text{Clause}(A_i \supset B_i) = \sum_{i=1}^{m} \text{Clause}(B_i).$$

###

5.2 Cook's Theorem, Necessity.

Cook's Theorem states that the acceptance problem for an **NP** set can be "encoded" by conjunctive normal form propositional formulas, and that the code can be computed in polynomial time(see[1]). Specifically, given an **NP** set, **S**, accepted by an $\mathcal{S}_n$-program, **P**, computing non-deterministically, there is for each $x \in A^*$ a polynomial-time computable CNF, Ψx, so that $x \in \mathbf{S}$ if and only if Ψx is satisfiable.

As the proofs of [2,pp.341] and [3] are usually done, we will not fully exhibit the polynomial time function which computes Ψx from x and **P**, but only provide a count of the number of atoms used in Ψx. The proof of Cook's Theorem is given into two parts. The first part shows the construction of Ψx, from which the necessity that Ψx is satisfiable if **P** accepts x follows, immediately. The second part, Theorem 5.2, provides an induction argument that, when Ψx is satisfied, **P** accepts x, which is the sufficiency portion of the proof.

Theorem 5.1 Given **P** and x, there is a CNF Ψx, of $O(P^3(|x|))$ atoms, which is satisfied if **P** accepts x in time $P(|x|)$.

Proof The atoms used in constructing Ψx are:

$\mathbf{R}_{k,j,s,p}$={At the step k, symbol a_s is in the position p of register V_j} ($1 \le k \le P(|x|)$, $1 \le j \le r$, $0 \le s \le n$, $1 \le p \le P(|x|)$).

$\mathbf{H}_{k,j,q}$= {At the step k, the length of the A* string on register V_j is q}($0 \le q \le P(|x|)$).

$\mathbf{L}_{k,h}$ = {At the step k, the h^th^ program line of **P** is executed}($1 \le h \le \mathcal{L}$).

Next, we encode the computation of **P** on k, in three part.

5.2.1 The Initialization

Let $x=\alpha_1\alpha_2\ldots\alpha_u$, where each $\alpha_i \in A$ and, to simplify notation, the subscripts do not denote the order of the symbols of A, and we may have $\alpha_i=\alpha_j$ even if $i \neq j$.

(1.1) "Register 1 is initialized with input x, and the symbol on each position of the register is unique at this step"

For each p, $1 \le p \le u$, each s, $0 \le s \le n$, $s \ne \alpha_p$.

$$\mathbf{R}_{1,1,\alpha_p,p}$$

$$\sim\mathbf{R}_{1,1,s,p}$$

The atom count is $(n+1)u$.

(1.2) "The rest of register 1 after position u is initialized blank(a_0), and not any other symbols."

For each p, $u+1 \le p \le P(|x|)$, each s, $1 \le s \le n$.

$$\mathbf{R}_{1,1,0,p}$$

$$\sim\mathbf{R}_{1,1,s,p}$$

The atom count is $(n+1)(P(|x|)-u)$.

(1.3) "Each remaining register is initialized blank all the way to position $P(|x|)$, and not any other symbols."

For each j, $2 \le j \le r$, and each p, $1 \le p \le P(|x|)$, each s, $1 \le s \le n$.

$$\mathbf{R}_{1,j,0,p}$$

$$\sim\mathbf{R}_{1,j,s,p}$$

The atom count is $(r-1)(n+1)P(|x|)$.

(1.4) "The length of the string on register 1 is u, and not any other length."

For each q, $0 \le q \le P(|x|), q \ne u$.

$$\mathbf{H}_{1,1,u}$$

$$\sim\mathbf{H}_{1,1,q}$$

The atom count is $P(|x|)+1$.

(1.5) "The length of register $j(\neq 1)$ is 0, not any other length."

For each j, $2 \leq j \leq r$, each q, $1 \leq q \leq P(|x|)$.

$$\mathbf{H}_{1,j,0}$$

$$\sim\mathbf{H}_{1,j,q}$$

The atom count is $(r-1)(P(|x|)+1)$.

(1.6) "Line 1 is being processed at step 1, and not any other line."

For each h, $2 \leq h \leq \ell+1$.

$$\mathbf{L}_{1,1}$$

$$\sim\mathbf{L}_{1,h}$$

The atom count is $\ell+1$.

Thus the initial part is the conjunction of CNF's given by (1.1)-(1.6) above all the clauses, called $\mathbf{Init}_x$. This expression is of length $O(P(|x|))$.

From this point onward, we present the CNF as a collection of clauses(disjunctions of atoms) which will be conjoined to form the final CNF. In each case we give the range of the subscripts, as in (1.3) above, so that for each selection of subscripts in the permitted range(s) there is an individual clause. I.E. in (1.3) there are $(r-1)(n+1)P(|x|)$ clauses and, since each is an atom in this case, the same number of atoms.

The remainder of the CNF will consist of one part, called CNF_k, for each computation step k.

In $P(|x|)$ computation steps, some legal computations may not halt while others may halt, i.e. reach line $\ell+1$. To write a CNF_k in a uniform way, we consider line $\ell+1$ as a "trapping state" which, once entered, causes each configuration to reproduce the one before. Using this device CNF_k "states" that the kth step of configuration results from the application of one of the four rules of **P**, or does nothing, if the trapping state has been reached.

In this Chapter the following terms are used:

state="Register length" + "Register Contents"
Configuration = State + Control

The clauses of CNF_k are prepared so that an unique truth value is assigned to each atom whose step subscript is k+1, called (k+1)-atoms, by the operation of step k, from the truth value already held by that and other atoms at the previous step. One way to view this is that we are defining a vector valued function $V(k)$, $0 \le k \le P(|x|)$, by recursion on k. For each k the value of $V(k)$ is a vector of truth values of length equal to the number of k-atoms. CNF_k is assembled in an obvious way from $V(k)$. We use the notations Atom(F) and Clause(F) for the number of atoms, clauses in the CNF, F.

We now discuss the construction of the CNF_k. We begin with a part of CNF_k which is designed to state that configuration are unique.

5.2.2 Unique Configuration, Q(k)

(2.1) "There is exactly one line to be executed"

For each h, $1 \leq h \leq \ell+1$, and each f, $1 \leq f \leq \ell+1$, $f \neq h$.

$$\sim \mathbf{L}_{k,h} \vee \sim \mathbf{L}_{k,f}$$

The atom count is $\ell(\ell+1)$.

(2.2) "The length of each register is unique."

For each j, $1 \leq j \leq r$, each q, $0 \leq q \leq P(|x|)$, each q', $0 \leq q' \leq P(|x|)$, $q' \neq q$.

$$\sim \mathbf{H}_{k,j,q} \vee \sim \mathbf{H}_{k,j,q'}$$

The atom count is $2rP(|x|)(P(|x|)+1)$.

(2.3) "There is exactly one symbol in position p for each register."

For each j, $1 \leq j \leq r$, each p, $1 \leq p \leq P(|x|)$, each s, $0 \leq s \leq n$, each s', $0 \leq s' \leq n$, $s \neq s$.

$$\sim \mathbf{R}_{k,j,s,p} \vee \sim \mathbf{R}_{k,j,s',p}$$

The atom count is $2r(n-1)nP(|x|)$.

Let the collection of all clauses (2.1)-(2.3) above be Q(k), which represents the unique configuration, at step k. Thus the total atom count of Q(k) is $O(P^2(|x|))$.

Next, we turn our attention to the part of CNF_k which insures correct change of state and of control. First, we define a formula CNF(h,k) for fixed line, h, of the program at step k. CNF(k,h) may be of several forms according to the type of line h being executed, thus we first give the flowchart to describe what is the special form for each case of CNF(k,h)

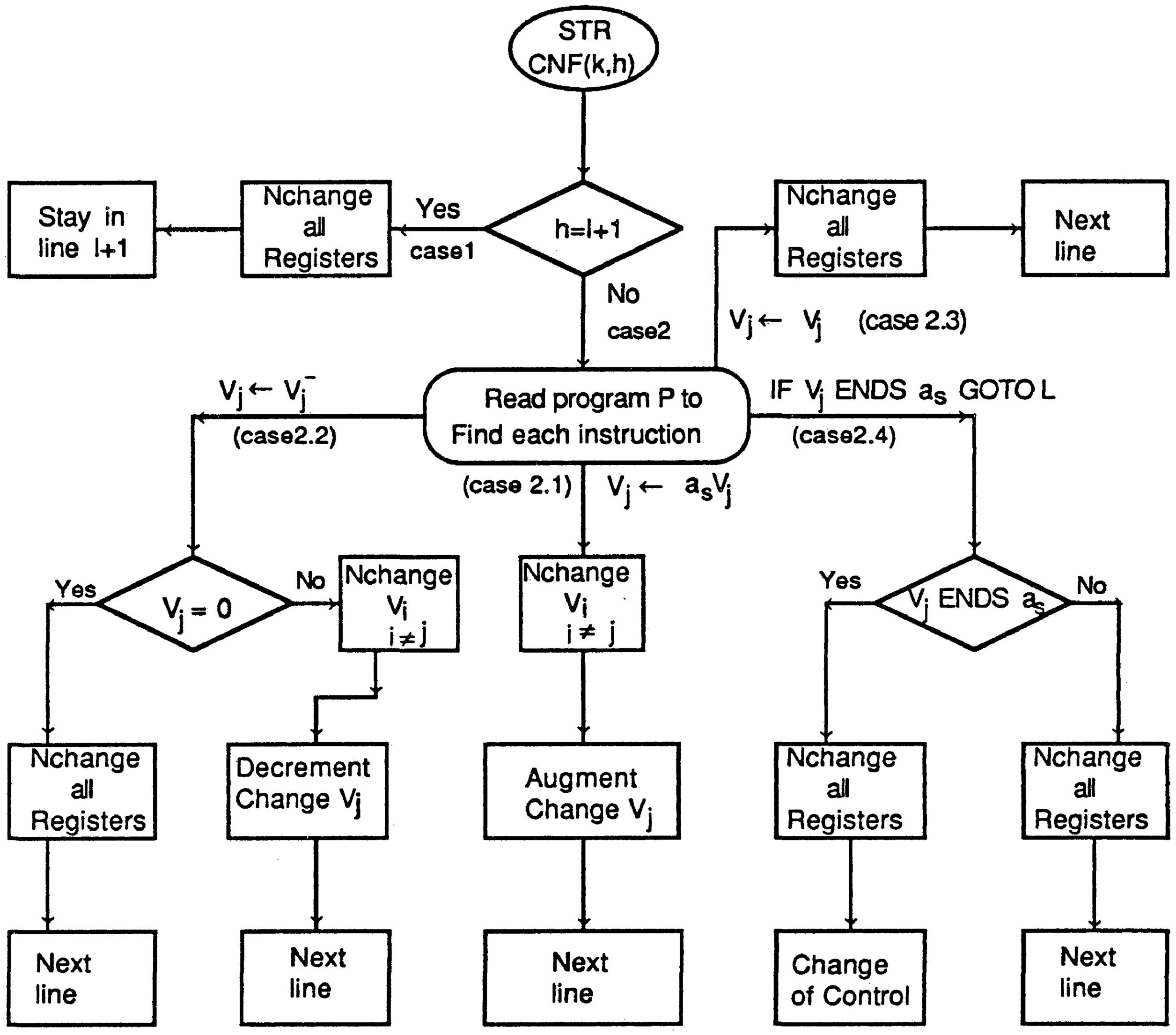

Fig 5.1

5.2.3 Change of State and of Control

There is at most one register, say j, whose content will be altered by any instruction, so the content of the rest of registers, i(≠j), in each position is not changed. Also the length of the non-blank string in register i is the same as before.

We consider the following clauses for fixed k and fixed h.

(I) Nchange in register i

a). "No change in the content of the register i."

For fixed i, for each s, $0 \leq s \leq n$, and each p, $1 \leq p \leq P(|x|)$.

$$\sim R_{k,i,s,p} \vee R_{k+1,i,s,p}$$

The atom count is $2(n+1)P(|x|)$.

The clause count is $(n+1)P(|x|)$.

b). "No change in the length of the register i."

For fixed i, for each q, $0 \leq q \leq P(|x|)$.

$$\sim H_{k,i,q} \vee H_{k+1,i,q}$$

The atom count is $2(P(|x|+1)$.

The clause count is $(P(|x|+1)$.

The CNF $NC_i(k,h)$ is the collection of all clauses in a) and b) aboves. And expresses the fact that there is no change in register i at step k, (if line h is executed[@]).

The atom count, $Atom(NC_i(k,h))$, is: $(2(n+1)P(|x|)+2(P(|x|+1))$.

The clause count, $Clause(NC_i(k,h))$, is: $((n+1)P(|x|)+(P(|x|+1))$.

(c) Let $NC_{\sim j}(k,h)$ be the collection of all clauses $NC_i(k,h)$, for $i \neq j$, $1 \leq i \leq r$. $NC_{\sim j}(k,h)$ expresses that there is no change in any register $i(\neq j)$ at step k, if line h is executed.

@ As yet, the line h does not play a rule in the definitions, but it will below.

The atom count, $Atom(NC_{\sim j}(k,h))$, is $(r-1)Atom(NC_i(k,h))$.

The clause count is, $Clause(NC_{\sim j}(k,h))$, is:

$(r-1)Clause(NC_i(k,h))$.

(d) Let

$$NCA(k,h) = \bigwedge_{i=1}^{r} NC_i(k,h)$$

NCA(k,h) states that there is no change in any register content when line h is executed at step k. It corresponds to [Nchange all Registers] in **fig 5.1.**

The atom count, $Atom(NCA(k,h))$, is $r(Atom(NC_i(k,h)))$.

The clause count is, $Clause(NCA(k,h))$, is

$$r(Clause(NC_i(k,h))).$$

(II) Change in register j

there are several possibilities for changing the content and the length of register j. Each case depends on the instruction of the $\mathbf{S}_n$ program **P** for the fixed k and fixed h.

Case 1 If at step k, line $\ell+1$ is being executed.

1.i) [Nchange all Registers]- "No change in all registers."

$$NCA(k,\ell+1)$$

1.ii) [Stay in line $\ell+1$] - "The computation stays in line $\ell+1$."

$$L_{k+1,\ell+1}$$

Let $NOP(k,\ell+1)$ denote the formula:

($\mathbf{L}_{k,\ell+1} \supset$ (collection of all clauses in case 1)

By Lemma 5.1, the atom count, Atom(NOP(k,ℓ+1)), is :

Atom(NOP(k,ℓ+1)=(Clause(NCA(k,ℓ+1))+1)+(Atom(NCA(k,ℓ+1))+1)

and the clause count, Clause(NOP(k,ℓ+1)), is :

Clause(NOP(k,ℓ+1)) = Clause(NCA(k,ℓ+1))+1

Case 2 If at step k, the line h($\neq\ell$+1) is being executed.

Case 2.1 If the instruction is "$V_j \leftarrow a_s V_j$" for fixed j.

2.1.i) [Nchange Register i] - "No change the content and the length of register i, for all i$\neq$j."

$$NC_{\sim j}(k,h)$$

2.1.ii) [Augment] - "The string length of register j is increased by 1."

For each q, $0 \leq q \leq P(|x|)-1$.

$$\sim\mathbf{H}_{k,j,q} \vee \mathbf{H}_{k+1,j,q+1}$$

The atom count is $2(P(|x|)+1)$.

The clause count is $(P(|x|)+1)$.

- "The content of the register j is changed by placing symbol a_s at position 1, and shifting all other symbols which were in the register j to the right by one position."

For fixed s and each s', $0 \leq s' \leq n$, each p, $1 \leq p \leq P(|x|)-1$.

$$\mathbf{R}_{k+1,j,s,1}$$

$$\sim\mathbf{R}_{k,j,s',1} \vee \mathbf{R}_{k+1,j,s',p+1}$$

The atom count is $(1+2(n+1)(P(|x|)-1))$.

The clause count is $(1+(n+1)(P(|x|)-1))$.

2.1.iii) [Next line] - "At the next step, the $(h+1)^{th}$ line will be processed."

$$\sim L_{k,h} \vee L_{k+1,h+1}$$

The atom count is 2.
The clause count is 1.

The CNF $A_{j,s}(k,h)$ is the collection of all clauses in case 2.1.

By Lemma 5.1, the atom count, $\mathrm{Atom}(A_{j,s}(k,h))$, is:

$$\mathrm{Atom}(A_{j,s}(k,h))=\mathrm{Atom}(NC_{\sim j}(k,h))+2(P(|x|)+1)$$
$$+1+2(n+1)(P(|x|)-1)+2$$

and the clause count, $\mathrm{Clause}(A_{j,s}(k,h))$, is :

$$\mathrm{Clause}(A_{j,s}(k,h))=\mathrm{Clause}(NC_{\sim j}(k,h))+(P(|x|)+1)+1$$
$$+(n+1)(P(|x|)-1)+1$$

Case 2.2 If the instruction is "$V_j \leftarrow V_j^-$" .

Case 2.2.1 V_j is empty.

2.2.1.i) [Nchange all Registers]-"No change in all registers."

$$NCA(k,h)$$

2.2.1.ii) [Next line] - "At the next step, the $(h+1)^{th}$ line will be processed."

$$\sim L_{k,h} \vee L_{k+1,h+1}$$

The atom count is 2.
The clause count is 1.

Let $ZN_j(k,h)$ denote the formula:

($\mathbf{H}_{k,j,0} \supset$ (collection of all clauses in case 2.2.1)

By Lemma 5.1, the atom count, Atom(ZN_j(k,h)), is :

Atom(ZN_j(k,h)) = (Clause(NCA(k,h))+1)+Atom(NCA(k,h)+2

and the clause count, Clause(ZN_j(k,h)), is:

Clause(ZN_j(k,h)) = Clause(NCA(k,h))+1

Case 2.2.2 V_j is not empty.

2.2.2.i) [Nchange register i] - "No change the content and the length of register i, for all i≠j。"

$NC_{\sim j}(k,h)$

2.2.2.ii) [Decrement] - "The string length of register j is decreased by 1".

For each q, $1 \le q \le P(|x|)$.

$$\sim\mathbf{H}_{k,j,q} \vee \mathbf{H}_{k+1,j,q-1}$$

The atom count is $2P(|x|)$.

The clause count is $P(|x|)$.

- "The symbol in end of the string is deleted, and the others are not changed".

For each q, $1 \le q \le P(|x|)$, each p, $1 \le p \le P(|x|)$, $p \ne q$, each s, $0 \le s \le n$.

$$\sim\mathbf{H}_{k,j,q} \vee \mathbf{R}_{k+1,j,o,q}$$

$$\sim\mathbf{H}_{k,j,q} \vee \sim\mathbf{R}_{k,j,s,p} \vee \mathbf{R}_{k+1,j,s,p}$$

The atom count is $2P(|x|)+3(n+1)P(|x|)(P(|x|)-1)$.

The clause count is $P(|x|)+(n+1)P(|x|)(P(|x|)-1)$.

2.2.2.iii) [Next line] - "At the next step, the (h+1)th line will be processed."

$$\sim\mathbf{L}_{k,h} \vee \mathbf{L}_{k+1,h+1}$$

The atom count is 2.
The clause count is 1.

Let $D_j(k,h)$ denote the formula:

$(\sim\mathbf{H}_{k,j,0} \supset$ (collection of all clauses in case 2.2.2)

By Lemma 5.1, the atom count, $\mathrm{Atom}(D_j(k,h))$, is:

$$\mathrm{Atom}(D_j(k,h))=(\mathrm{Clause}(ZN_j(k,h))+P(|x|)+P(|x|)+ (n+1)P(|x|)(P(|x|)-1)+1)+(\mathrm{Atom}(ZN_j(k,h)) +2P(|x|)+2P(|x|)+3(n+1)P(|x|)(P(|x|)-1)+2)$$

and the $\mathrm{Clause}(D_j(k,h)) = \mathrm{Clause}(ZN_j(k,h))+P(|x|)+P(|x|)+ (n+1)P(|x|)(P(|x|)-1)+1$

Case 2.3 The instruction is "$V_j \leftarrow V_j$".

This case is the same as case (2.2.1). that is:

$ZN_j(k,h)$.

Case 2.4 The instruction is "IF V_j ENDS a_s GOTO L_β".

Case 2.4.1 V_j does not end with symbol a_s.

2.4.1.i) [Nchange all Registers] - "No change all register."

$$NCA(k,h)$$

2.4.1.ii) [Change of Control] - "At the next step, the (h+1)th line will be processed."

$$\sim\mathbf{L}_{k,h} \vee \mathbf{L}_{k+1,h+1}$$

The atom count is 2.

The clause count is 1.

Let $NSE_{j,s}(k,h)$ denote the formula:

$$\bigwedge_{q=0}^{P(|x|)}[(\mathbf{H}_{k,j,q}\wedge\sim\mathbf{R}_{k,j,s,q})\supset(\text{collection of all clauses in case 2.4.1})].$$

By Lemma 5.1, the atom count of $NSE_{j,s}(k,h)$ is:

$$P(|x|)(2(\text{Clause}(NCA(k,h))+1)+\text{Atom}(NCA(k,h))+2).$$

and the clause count of $NSE_{j,s}(k,h)$ is:

$$P(|x|)(\text{Clause}(NCA(k,h))+1).$$

Case 2.4.2 V_j ends with symbol a_s.

2.4.2.i) [Nchange all Registers] - "No change in all register."

$$NCA(k,h)$$

2.3.2.ii) [Change of control] - "One of the lines with label L_β will be executed at next step."

$$\sim\mathbf{L}_{k,h}\vee\bigvee_{f\in H_\beta}\mathbf{L}_{k+1,f}$$

The atom count is $1+\mu_\beta$.

The clause count is 1.

Let $G_{j,s,\beta}(k,h)$ denote the formula:

$$\bigwedge_{q=0}^{P(|x|)}[(\mathbf{H}_{k,j,q}\wedge\mathbf{R}_{k,j,s,q})\supset(\text{collection of all clauses in case 2.4.2})].$$

By Lemma 5.1, the atom count of $G_{j,s,\beta}$ is:

$P(|x|)(2(\mathrm{Clause}(NCA(k,h))+1)+\mathrm{Atom}(NCA(k,h))+1+\mu_\beta)$.

and the clause count of $G_{j,s,\beta}$ is:

$$P(|x|)(\mathrm{Clause}(NCA(k,h))+1).$$

Now we can create the formula CNF(k,h) as follows for fixed k and fixed h.

$CNF(k,h) = NOP(k,h)$ {If line $\ell+1$ is to be executed }

$CNF(k,h) = A_{j,s}(k,h)$ {If line h reads "$V_j \leftarrow a_s V_j$"}

$CNF(k,h) = ZN_j(k,h) \wedge D_j(k,h)$ {If line h reads "$V_j \leftarrow V_j$", or "$V_j \leftarrow V_j^-$ "}

$CNF(k,h) = NES_{j,s}(k,h) \wedge G_{j,s,\beta}(k,h)$
{If line h reads "IF V_j ENDS a_s GOTO L_β",}

Although the particular choice of CNF(k,h) depends upon the instruction on line h of the program **P**, we see from the above atom count that, regardless of the line, the atom count of CNF(k,h) is $O(P^2(|x|)$.

At step k, only one line will be executed in **P**, so we define a single CNF, call CNF_k as follows.

$$CNF_0 = \mathbf{Init}_x \wedge Q(1)$$

$$CNF_1 = CNF(1,1) \wedge Q(2)$$

$$CNF_k = \bigwedge_{h=1}^{\ell+1} (\mathbf{L}_{k,h} \supset CNF(k,h)) \wedge Q(k+1) \qquad k \neq 0,1$$

Although these formulas are not yet in CNF, Lemma 5.1 shows that the atom count of CNF_k is $O(P^2(|x|))$.

Therefore, if there is an $\mathbf{S}_n$ program **P** which accepts the input x in time P(|x|), then the CNF $\boldsymbol{\Psi x}$ defined as follows will be satisfied.

$$\boldsymbol{\Psi x} = \bigwedge_{k=0}^{P(|x|)} CNF_k \wedge \mathbf{L}_{P(|x|),\, l+1}$$

Since the atom count of CNF_k is $O(P^2(|x|))$, thus the atom count of Ψx is $O(P^3(|x|))$, and we have proved Theorem 5.1.

5.3 Cook's Theorem, Sufficiency

Lemma 5.2. For each $0 \le T \le P(|x|)$, if the CNF

$$\bigwedge_{k=0}^{T} CNF_k$$

is satisfied, then there is a correct computation from program **P**, of length T, with input x.

For the proof of Lemma 5.2, as before we refer to the atoms $\mathbf{L}_{k,h}$, $\mathbf{H}_{k,j,q}$, and $\mathbf{R}_{k,j,s,p}$ with first subscript k, as <u>k-atoms</u>, and denote the collection of all k-atoms by $\mathbb{A}_k$ and the subset of those which are assigned the value TRUE by A_k. An examination of the construction of the formulas CNF_k shows that to satisfy CNF_k it is necessary to assign truth values to the set $\mathbb{A}_{k+1}$ of (k+1)-atoms. The process can be thought of

as proceeding according to the following table, from step 0 to k.

From the set of all step-1 atoms $\mathbb{A}_1$, A_1 assigned value TRUE;

From the set of all step-2 atoms $\mathbb{A}_2$, A_2 assigned value TRUE;

$\vdots \qquad \vdots$

From the set of all step-k atom $\mathbf{A}_T$, A_T assigned value TRUE;

From the set of all step-K+1 atoms $\mathbf{A}_{T+1}$, A_{T+1} assigned value TRUE.

The Chart of Atoms Assigned Value

Step	Atom Assigned value TRUE					
0	A_1					
1	A_1	A_2				
2	A_1	A_2	A_3			
$\vdots$	$\vdots$		$\vdots$	. . .		
T-1	A_1	A_2	A_3	. . .	A_T	
T	A_1	A_2	A_3	. . .	A_T	A_{T+1}

Fig 5,2

Proof By induction on T.

Basis T=1.

Satisfaction of the initialization configuration, C_0, shows immediately that there is a correct computation C_0 from **P**, of length 1.

For the induction step, the induction assumption is:

For $1 \le T \le P(|x|)$, if $\bigwedge_{k=0}^{T} CNF_k$ is satisfied, then there is a

correct computation, C_0, $C_{i-1} \vdash C_i$, i=1,2,...,T, of length T. (ending with line number, register lengths, and register contents determined by the TRUE atoms in CNF_k). Further, suppose that

$$\bigwedge_{k=0}^{T+1} CNF_k$$

is satisfied. Since we have

$$\bigwedge_{k=0}^{T+1} CNF_k = \bigwedge_{k=0}^{T} CNF_k \wedge CNF_{T+1}$$

then both $\bigwedge_{k=0}^{T} CNF_k$ and CNF_{T+1} are satisfied.

i) $\bigwedge_{k=0}^{T} CNF_k$ is satisfied;

ii) CNF_{T+1} is satisfied.

The induction step proceeds from the above two assumptions, and the induction assumption.

We have just noted that the truth values of all atoms whose step subscript is T+1 have been determined in satisfying CNF_T. The part Q(T+1) of CNF_T guarantees unique line number, register lengths and register contents. From (i) and (ii) and the induction assumption we conclude there is a correct computation of T steps, and that CNF_{T+1} is satisfied. We must see that truth value assigned the atoms of step subscript T+2 in satisfying CNF_{T+1} determine one more correct computation step,the $(T+1)^{th}$. The reasons are eventually obvious, from the way in which CNF_{T+1} was constructed.

Satisfaction of CNF_T requires that exactly one of the atoms $\mathbf{L}_{T+1,h}$ is TRUE. From CNF_{T+1} we see that for that value of h, denoted h', CNF(T+1,h') must be satisfied.

It remains to show that the truth values assigned in CNF(T+1,h') determine a correct $(T+1)^{th}$ step computation step.

Since we are arguing in general, we cannot commit to a fixed form for line h' of the program **P** but must proceed by cases on the four possible forms line h' may have. However, to avoid tedious repetition, we argue just one case, simply to show the form that the argument should take.

Suppose line h' is of the form "IF $V_{j'}$ ENDS $a_{s'}$ GOTO $L_{\beta'}$" and that the line numbers of **P** whose labels are $L_{\beta'}$ are $h_1, h_2, \ldots, h_t$, here $h_i \in H_{\beta'}$, $|H_{\beta'}| = \mu_{\beta'} = t$.

As above, exactly one atom $\mathbf{H}_{T+1,j',q'}$ has value TRUE and, by the induction assumption, this means that the true length of the register content in register j' is q' in the correct computation of length K. There are two corresponding possibilities for atom $\mathbf{R}_{T+1,j',s',q'}$.

(a) $\mathbf{R}_{T+1,j',s',q'}$ is FALSE. Then, from case (2.3.1.iii) we see that atom $\mathbf{L}_{T+2,h'+1}$ is TRUE, and that for each j,s,q, the "content" atoms $\mathbf{R}_{T+2,j,s,q}$ retain the same values as $\mathbf{R}_{T+1,j,s,q}$.

Thus, in case (a), since $\mathbf{R}_{T+1,j',s',q'}$ FALSE means that if $V_{j'}$ does not end with symbol $a_{s'}$, the (T+2)-atoms in CNF_{T+1} have the correct values to determine the next computation step.

(b) $\mathbf{R}_{T+1,j',s',q'}$ is TRUE. Then, from case (2.3.2.iii) we see that atom $\mathbf{L}_{T+2,f}$ is TRUE for exactly one of $f\in H_{\beta'}$, and that for each j,s,q, the "content" atom $\mathbf{R}_{T+2,j,s,q}$ retain the same values as $\mathbf{R}_{T+1,j,s,q}$.

Thus, in case (b), since $\mathbf{R}_{T+1,j',s',q'}$ TRUE means that if $V_{j'}$ ends with symbol $a_{s'}$, the (T+2)-atoms in CNF_{T+1} have the correct values to determine the next computation step. I.E. The line number, register length and register contents may be altered to agree with the truth values of the (T+2)-atoms and the result will be a correct computation step.

Completing the proof of the other cases in exactly the same manner leads to the completion of the induction step and thus the proof of the lemma.

###

Theorem 5.2 If $\Psi x = \bigwedge_{k=0}^{P(|x|)} CNF_k \wedge \mathbf{L}_{P(|x|),\ell+1}$ is satisfied, then there is a $\boldsymbol{S}_n$-program **P** which accepts x in time P(|x|).

Proof By Lemma 5.2, there is a correct computation C_0, $C_i \vdash C_{i+1}$, i=1,2,...,P(|x|)-1, of length P(|x|). Satisfaction of CNF_0 provide the truth values to establish C_0, the configuration corresponding to the initial line number(i.e. the first line), all register lengths and register contents with input x for **P**.

Lemma 5.2 shows that satisfaction of $\bigwedge_{k=0}^{P(|x|)} CNF_k$ causes a correct computation of length P(|x|).

Finally, since $\mathbf{L}_{P(|x|),\ell+1}$, is satisfied, computation of length $P(|x|)$ results in acceptance of x.

###

As discussed earlier, Theorem 5.1 and 5.2 together constitute Cook's Theorem except for the usual omission, to show that Ψx is computable in polynomial time in $|x|$. We have shown that the atom count in Ψx is $O(P^3(|x|))$ for each x belonging to a set X accepted in time $P(|x|)$. It remains to observe that Ψx is constructed in a uniform effective manner from the input x and the $\mathcal{S}_n$-program $\mathbf{P}$. Thus(by Church's Thesis) Ψx, or an appropriate code for Ψx, is computable by an $\mathcal{S}_n$-program, in a computation time proportional to its atom count, i.e. in $O(P^3(|x|))$.

REFERENCES

1. S. A. Cook, *The complexity of theorem proving procedures,* Proceedings of the Third ACM Symposium on Theory of Computing(1971),pp.151-158.

2. M. D. Davis and E. J. Weyuker, *Computability, Complexity, and Languages(Fundamentals of Theoretical Computer Science),* Academic press, 1983.

3. Herbert, Wilf, *Algorithms and Complexity,* Prentice-Hall, Englewood Cliffs, New Jersey, 1986, pp.197-201.

4. H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation,* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

5. E. Mendelson, *Introduction to Mathematical Logic,* Second edition, D. Van Nostrand Company, 1979, pp.28.

6. E. L. Post, *Recursive unsolvability of a problem of Thue,* The Journal of Symbolic Logic, Vol.11,1947, pp.1-11.

7 L. Stockmeyer, *Classifying the computational complexity of problems,* The Journal of Symbolic Logic, Vol.52, No 1, 1987, pp.1-43.