AN ABSTRACT OF THE THESIS OF

OLADELE FAJUYIGBE for the degree of MASTER OF SCIENCE in COMPUTER SCIENCES presented on September 1, 1978.

TITLE: THE SHORTEST STRINGS IN A CONTEXT-FREE LANGUAGE

ABSTRACT APPROVED: Redacted for Privacy

Associate Professor Curtis R. Cook.

In this thesis we show that the Mclean-Johnston Algorithm for finding the shortest terminal strings derivable from non-terminals in a context-free grammar G is of $O(N*|G|)$, where N is the number of variables (non-terminals) and $|G|$ is the size of the grammar G.

We also present a more efficient algorithm of $O(|G|)$ for the same task.

Our algorithm requires a very elaborate data structure for its implementation; however, in spite of its elaborate nature, the algorithm has the same space complexity as that of Mclean and Johnston, $O(|G|)$. And since our algorithm has a smaller time complexity, we believe that it is superior to the Algorithm of Mclean and Johnston.

THE SHORTEST STRINGS IN A CONTEXT-FREE LANGUAGE

by

OLADELE FAJUYIGBE

A THESIS

submitted to

OREGON STATE UNIVERSITY

In partial fulfillment of
the requirements for the
degree of

Master of Science

Completed September 1, 1978

Commencement June 1979

APPROVED:

Redacted for Privacy

Associate Professor of Computer Sciences
In charge of Major

Redacted for Privacy

Acting Chairman of Department of Computer Sciences

Redacted for Privacy

Dean of Graduate School

Date Thesis is Presented: September 1, 1978

Typed by Shanda L. Smith for Oladele Fajuyigbe.

DEDICATION:

From a new comer,

To all those who have worked relentlessly to unearth the mysteries of the black box - the computer. It is in understanding how it works that the fascination of the computer lies.

Also in undying memories of my grandmother - OSUHUNMIKE, my mother - JULIANAH and my sister - SABETI, all of whom were wonderful people.

To Professors Curtis R. Cook and Paul Cull, I owe a depth of gratitude for suggesting this area of research. Without the patient and stimulating supervision from both of them, this work would not have been possible in its present form.

I am also indebted to the University of Benin, Benin-City, Nigeria for granting me a one-year special training leave for the purposes of studying Computer Science at Oregon State University; and to Oregon State University for providing an excellent atmosphere for academic pursuit.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

THE SHORTEST STRINGS IN A CONTEXT-FREE LANGUAGE

I. INTRODUCTION

In [9] Michael J. Mclean and Daniel B. Johnston presented an algorithm for finding the shortest terminal strings derivable from each variable in context-free grammars (CFG).

There was no mention of time bounds in relation to the algorithms. This might be due to the lack of an acceptable notion of size in CFG. However, with the works of Ginsburg and Lynch [6] this gap has been bridged. So that it makes sense for one to attempt an investigation of the time bounds of the algorithm.

In this paper, we obtain the time complexity of the Mclean-Johnston algorithm. Furthermore, with the aid of an appropriate data structure, we give an algorithm which performs faster than the Mclean-Johnston, although both are of the same space complexity.

In chapter 2, we discuss basic facts about context-free grammars.

In chapter 3, we give a detailed description of the Mclean and Johnston Algorithm. We introduce the four measures of size in a CFG and obtain the time complexity of the Mclean-Johnston algorithm in relation to one of these.

Finally, in chapter 4, we introduce our new algorithm, discuss its data structure, space and time complexities. As a byproduct, the algorithm detects all useless variables, where a useless variable is one not involved in the derivation of a terminal string.

This work was motivated in part by the realization, as pointed out in [1], that out of the four classes of grammars in the Chomsky Hierarchy [8], the CFG are the most important in terms of application to programming languages and compiling. In converting a source program into an object program, the compiler goes through a series of processes. As shown in Figure 1, one of these processes is the syntax analysis. During this stage the string of tokens generated by the lexical analyser (the scanner) is examined to determine whether the string obeys certain structural conventions explicit in the syntactic definition of the programming language being used. A context-free grammar can be used to specify most of the syntactic structures of a programming language. This may explain the importance attached to context-free grammars and their various extensions in the theory of computation.

Various algorithms have been developed for parsing CFG; among these are Early's Algorithm [1], and the Cocke-Young-Kasami Algorithm [1]. The performances of these algorithms have been greatly improved by appropriate data structures. This brings us to our second reason for studying the present problem:

Since the effects of an appropriate data structure on the performances of an algorithm cannot be overemphasized, we are interested in finding out whether we could specify a data structure that will improve the time complexity of the Mclean-Johnston Algorithm.

We present in chapter 4 the results of our investigation.

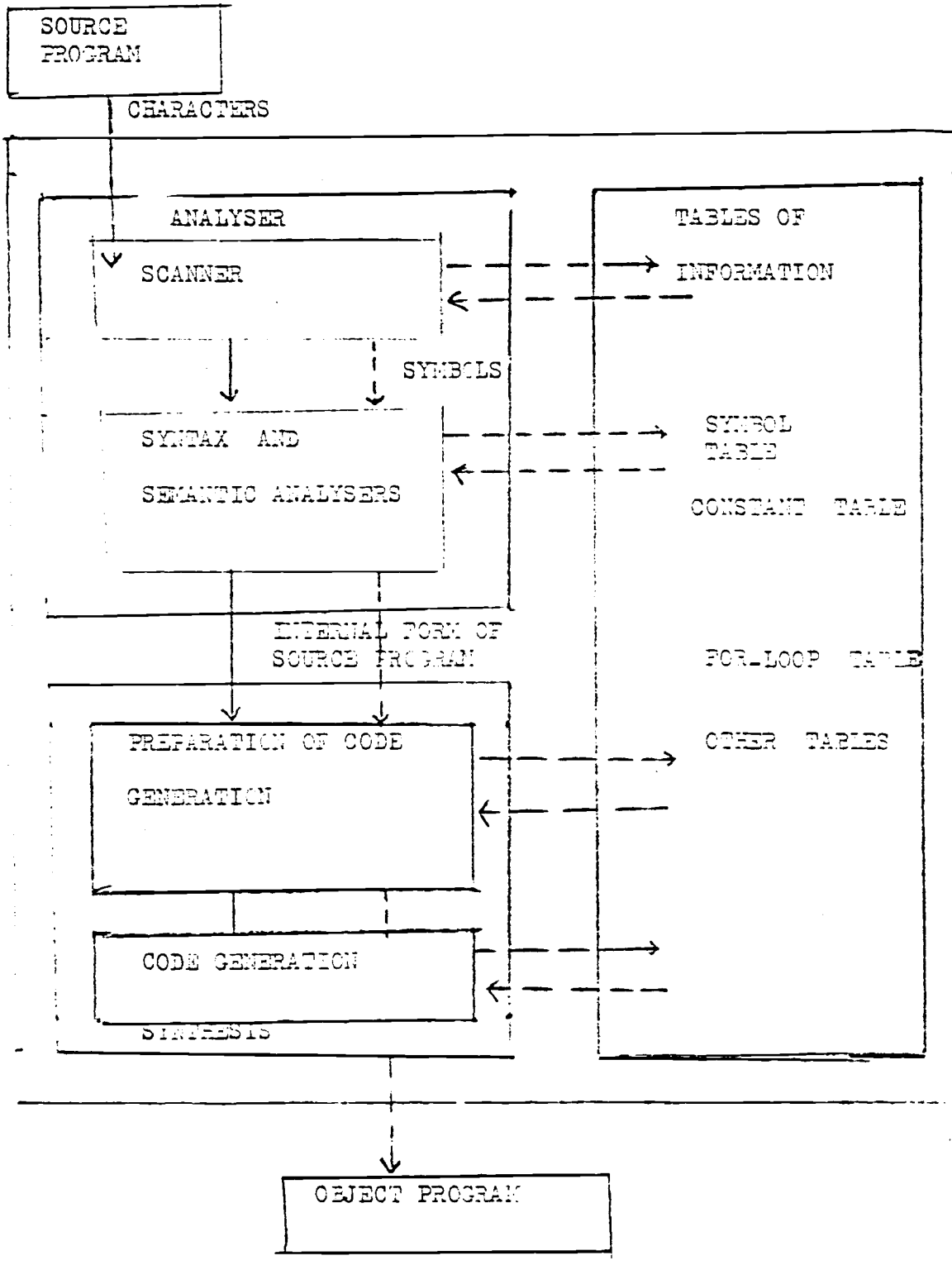


Fig.1. LOGICAL PARTS OF A COMPILER

(according to Gries [7]).

II. CONTEXT-FREE GRAMMARS AND LANGUAGES

In this chapter, we shall give a few definitions, examples, basic facts and simplification lemmas for context-free grammars and languages. We do not intent to go into too much detail, however, we shall give adequate references about proofs and further explanations.

First, we define an alphabet and a language:

An alphabet V is any finite set of symbols. The set of all finite strings over V is denoted by V*, the symbol ε is used to indicate the null string, i.e., the string that consists of no symbols, while V^+ denotes the set $V^*-\{\varepsilon\}$.

A language is any set of finite strings over an alphabet.

Definition: 2.1. A context-free grammar is a 4-tuple $G = (V_N, V_T, P, S)$, where

(2.1.1) 1. V_N is a finite set of symbols called variables (non-terminals).

2. V_T is a finite set of terminal symbols, disjoint from V_N. For convenience, we shall let $V = V_N \cup V_T$.

3. P is a finite subset of $V_N \times (V_N \cup V_T)^*$. An element $(A, \beta) \in P$ is called a production rule and written.

(2.1.2) $A ::= \beta$, $A \in V_N$ and $\beta \in V^*$.

4. S is a distinguished symbol in V_N called the start symbol.

The production rule (2.1.2) is said to be associated with A. Usually in describing a context-free grammar, all the rules associated with a given variable A are grouped together and written as

(2.1.3) $A ::= \beta_1 | \beta_2 | \ldots | \beta_n$, $\beta_j \in V_N$, $1 \leq j \leq m$, and | denotes 'or'.

Our definition of a CFG allows the string β to be a null string, although the usual definition [8] does not allow this.

We define the length $|w|$ of a string w in V^* as the number of symbols in w. $|\varepsilon| = 0$.

In a CFG $G = (V_N, V_T, P, S)$, the string w_1Aw_2, $w_1, w_2 \in V^*$, and $A \in V_N$ is said to directly generate (derive) the string $w_1 x w_2$, $x \in V^*$, if $A::=x$ is a production rule of G. In this case we simply write

(2.1.4) $$w_1Aw_2 \rightarrow w_1 x w_2$$

And this is similarly called a direct derivation in G.

More generally, a string w is said to generate w' if there exists a finite chain of strings $w_1, w_2, \ldots, w_n$ such that w_i is directly generated by w_{i-1}, $i = 2,3,\ldots,n$, $w=w_1$ and $w_n = w$. We specify this by writing

(2.1.5) $$w \overset{*}{\underset{G}{\rightarrow}} w'.$$

When there is no danger of confusion, the G in (2.1.5) is supressed.

Definition: 2.2. The language generated by G, denoted by $L(G)$, is defined as

$$L(G) = \{w | w \in V_T^*, \text{ and } S \overset{*}{\underset{G}{\rightarrow}} w\}.$$

The language L is said to be context-free if there is some CFG G such that $L = L(G)$.

A CFG is said to be ε-free, if P has no ε-production rules.

Since by [1], every context-free language CFL has an ε-free grammar which generates it, we may content ourselves with ε-free grammars.

Definition: 2.3. Let $G = (V_N, V_T, P, S)$ be a CFG. A tree is a derivation tree for G if:

1. Every node has a label, which is a symbol of V.
2. The label of the root is S.
3. If a node n has at least one descendant other than itself, and has label A, then A must be in V_N.
4. If nodes $n_1, n_2, \ldots, n_k$ are the direct descendants of node n, in order from the left, with labels $A_1, A_2, \ldots, A_k$, respectively, then

$$A:: = A_1A_2\ldots A_k$$

must be a production in P.

We refer to the tree as the derivation tree of the string represented from left to right by the labels of the pendant nodes.

If S:: = ε is a production rule in P, then a derivation tree for the string ε is

The descendant node is neither specified nor labelled.

Example:

$G = (V_N, V_T, P, S)$, where

$V_N = \{S, A, B\}$

$V_T = \{a, b, d\}$, and the production rules are

S:: = SA|AB

A:: = a|ABB

B:: = b|Bd

Then Figure 2 is a derivation tree of the string abd from S.

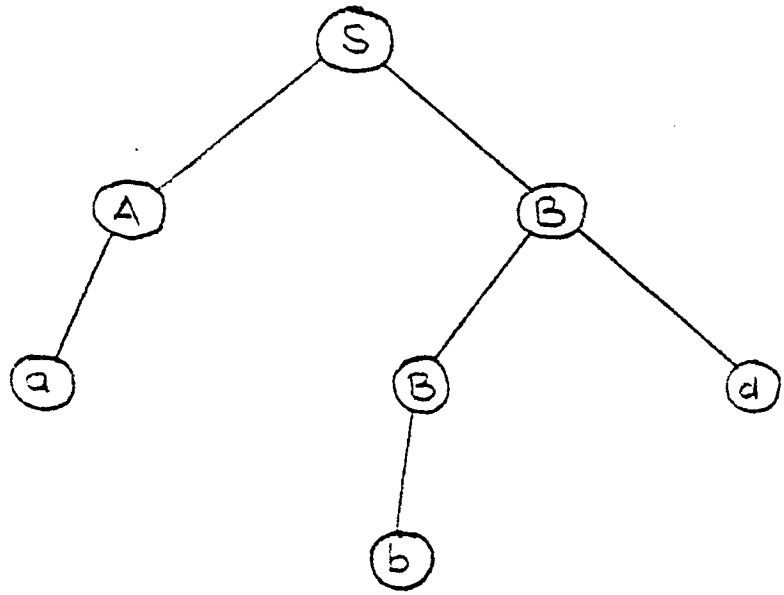


Figure 2. A derivation tree.

We are now going to discuss some theorems which relate directly to the derivation trees and the shortest terminal strings.

Our use of derivation tree will not always require the restriction that the root be S. Whenever we require the root to be any other symbol A, different from S, we shall refer to the derivation tree as an "A-derivation tree."

We give now a theorem proved in [8] page 21, Theorem 2.3:

Theorem 2.4. Let $G = (V_N, V_T, P, S)$ be a CGF. Then $S \overset{*}{\rightarrow} \alpha$, $\alpha \in V$, if and only if there is a derivation tree in G with a result α.

A corollary of this theorem is the following:

Corollary: 2.5. Let G be a CFG, then the language generated by B is empty if and only if the collection of derivation trees whose pendant nodes form a terminal string is empty.

Since the language generated by the grammar $G = (V_N, V_T, P, S)$ is empty if there are no shortest terminal strings derivable from the start symbol S, an algorithm on shortest terminal strings will help determine whether the language generated by a CFG is empty or not and also identify useless variables and production rules, and thus help in reducing the grammar.

Lemma 2.6. Let G be a CFG, then the shortest terminal string derivable from a variable A, if it exists, can be produced by applying only production rules which do not reintroduce the symbol A.

Proof: Consider an A-derivation tree of such a string, if there is another node n other than the root with the label A, then the subtree rooted at n has A as the root and to the pendant nodes one will have a string, no longer than the original string. Thus one has reduced the copies of A by one, and continuing in like manner, one will reach a situation in which there is only a node, the root, with the label A.

It is now clear that one can pair together a shortest terminal string derivable from a variable A and a production rule associated with A used in achieving it. This leads us to yet another definition:

Definition. 2.7. Let G be a CFG. A production rule

$$A::= x$$

will be referred to as a shortest rule of A if starting with this rule one can derive a shortest terminal string for A.

Now, the following theorem constitutes the whole basis of the Mclean-Johnston algorithm:

Theorem. 2.8. Let G be a CFG. Let A, A_1, A_2, ..., A_k be the non-pendant nodes of the derivation tree of a shortest terminal string w derivable from the variable A, then, for $1 \leq j \leq k$, the subtree with root A_j, is a derivation tree of a shortest terminal string w_j derivable from the variable A_j.

Proof: Assuming A_j is not the root of a derivation tree of a shortest terminal string for A_j, then replacing the "A_j-subtree" by a derivation tree of one of its shortest terminal strings will produce a shorter terminal string for A, contradicting the choice of the string w.

III. THE MCLEAN AND JOHNSTON ALGORITHM

In this chapter, we describe the Mclean-Johnston Algorithm [9] for finding the shortest terminal strings derivable from variables in a CFG.

Basically the algorithm sets out to determine whether a terminal string could be derived from a variable, and if so finds the length of the shortest string, and finally finds a shortest terminal string for a variable.

This is a three-in-one task. However, the algorithm achieves this in two parts. Part I of the algorithm: In this part a yes or no is returned to the question whether a terminal string can be derived from a given variable, also the shortest length of such a string is returned together with a production rule associated with the variable that could be used to derive the string.

Part II consists in using the production rule obtained in part one to construct the derivation tree of a shortest string, and therewith the string itself.

3.1. The Description of the Algorithm

For each variable A the algorithm finds the length of the known shortest terminal string derivable from it. This we call temporary shortest length of a terminal string derivable from A, and denote it by L(A).

Defining the temporary length of a production rule

$A:: = A_1 A_2 \ldots A_m$ as $\sum_{j=1}^{m} L(A_j)$, if $L(A_j)$ is defined for each j, $1 \leq j \leq m$,

the algorithm finds for each variable A a temporary shortest production rule P(A) whose temporary length, if defined, is equal to L(A).

It searches all production rules associated with A for a shorter production rule than P(A), if one exists, it updates P(A) and L(A) accordingly.

After searching sequentially the production rules associated with A it proceeds to the next variable.

If after a complete search of all the production rules no temporary shortest production rule is replaced, the algorithm stops.

The following partially coded form will help make the algorithm clear.

SS denotes the set of symbols known to have terminal strings derivable from them, N the number of variables.

```
L00:    SS ← V_T
L0:     I ← 0.
        If I ≠ N THEN GO TO L6.
L1:     I ← I + 1
L2:     USE SS TO UPDATE THE NEXT PRODUCTION RULE ASSOCIATED WITH I.
        IF RULE IS NOT SHORTER THAN P(I) THEN GO TO  L3.
        UPDATE P(I) AND L(I) ACCORDINGLY.
        SET CHANGE TO TRUE.
        PUT I IN SS.
L3:     IF ALL THE RULES ASSOCIATED WITH I ARE NOT YET EXHAUSTED THEN
        GO TO L2.
L4:     OF I ≠ N THEN GO TO L1.
```

L5: IF CHANGE THEN GO TO L0.

L6: STOP.

McLean and Johnston decided to initialize P(A) and L(A) to 0. With this type of initialization the algorithm required another Boolean variable E(A), to indicate whether or not a terminal string is known to exist for A; otherwise the definition of temporary shortest length will not hold; these are now defined for only variables for which E is True.

3.2 The Data Structure of the Algorithm

The variables are linearly ordered and listed as

$$A_1, A_2, \ldots, A_n$$

Similarly, all the production rules associated with a particular variable are grouped together into one block of rules:

(3.2.1) $A_i :: = \alpha_{i1} | \alpha_{i2} | \ldots | \alpha_{il_i}$, $1 \leq i \leq n$, where

(3.2.2) $\alpha_{ik} = B_1 B_2 \ldots B_{m_k}$, $1 \leq k \leq l_i$, $B_j \in V^*$, $1 \leq j \leq m_k$.

A quick look at (3.2.1) and (3.2.2) suggests a list representation for all the production rules, and this is defined as follows:

A one-dimensional array PP holds all the production rules, with 0 delimiting the rules. Each block (3.2.1) has all its production rules stored consecutively together. Another array D indicates the range within which the production rules associated with each variable could be found, i.e. PP(k), $D(i-1)+1 \leq k \leq D(i)$, contain the production rules associated with the variable numbered i.

And lastly, the variables are stored in a circular list LINK, i.e., LINK(i) = i+1, $1 \leq i \leq n-1$, and LINK(n) = 1,

Example: $G = (\{A_1, A_2\}, \{x_1, x_2\}, P, A_1)$

$A_1 :: = A_2x_1 | x_2$

$A_2 :: = A_1A_2 | x_1$

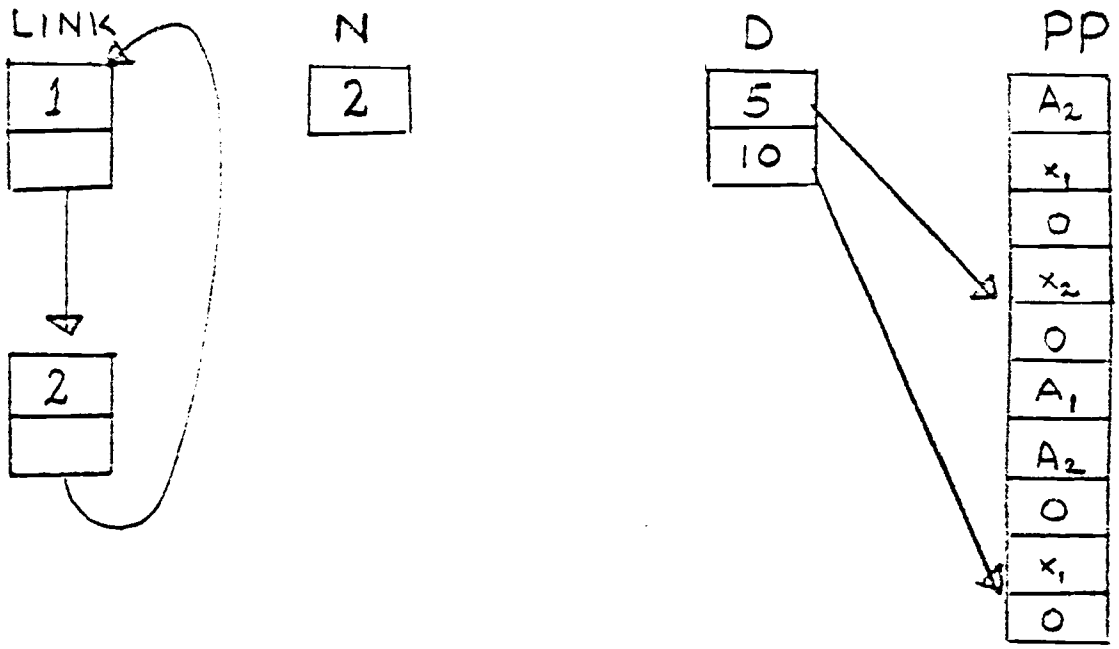


Figure 3.2. The Mclean-Johnston Data Structure

3.3 The Computer Program for the Mclean-Johnston Algorithm

The length function and the Boolean variable L and E respectively, of section 3.1 are extended to functions L^+ and E^+ over V as follows:

(3.3.1) $E^+(X) = \begin{cases} \text{True, if X is a terminal} \\ E(X), \text{ if X is in } V_N \end{cases}$

(3.3.2) $L^+(X) = \begin{cases} 1, \text{ if X is a terminal} \\ L(X), \text{ if X is in } V_N \end{cases}$

Using E^+ as a control function, one can now think of L as a function over V*, by setting

$$(3.3.3)\quad E^*(X) = \prod_{i=1}^{m} E^+(A_i), \text{ if } X = A_1A_x\ldots A_m,\ A_i \text{ in } V$$

and

$$L^*(X) = E^*(X) * \sum_{i=1}^{m} L^+(A_i)$$

where E*(X) is 1 or 0 depending on whether it is true or false. Then our temporary length of a production rule defined in 3.1 is the value of L* evaluated on its right hand side. The computer program is designed to minimize L*(A) over production rules associated with A.

To reduce the number of searches needed another function G(X) is introduced. It stores an estimated lower bound for the shortest length of a terminal string derivable from the variable A. So that as soon as G(A) = L(A), we know a shortest terminal string has been found for the variable A. G(X) is simply the value of L*(X) in (3.3.3) without the factor E*(X)! The minimum of this over production rules associated with A gives the value of G(A).

See Appendix I for the detailed computer program.

3.4 Measure of Size in a CFG

In this section, we shall introduce the four notions of size in a CFG. With this we shall then be able to determine the time complexity of the Mclean-Johnston Algorithm.

According to [2], the time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm. Therefore central to time complexity is the notion of size.

According to Ginsburg and Lynch [6], there are four notions of size in a CFG.

These are as follows:

1. S_G - the total number of occurrences of variables and terminals on both sides of all production rules in G.
2. V_G - the total number of occurrences of variables on both sides of all production rules in G.
3. P_G - the number of production rules of G.
4. N_G - the number of variables of G.

Assuming that each variable has at least a production rule associated with it, then the above measures of size satisfy the following inequalities:

(3.4.1) $$N_G \leq P_G \leq V_G \leq S_G$$

For our present study, the measures of size that seem appropriate are P_G, V_G, S_G, and we may need the following modifications:

i. S_R - the total number of occurrences of variables and terminals on the right hand side of all the production rules.

ii. V_R - the total number of occurrences of variables on the right hand side of all the production rules. And finally

iii. S_{RD} - defined as follows:

Let RD(K) = number of different variables appearing on the right hand side of the K^{th} production rule, adding 1 if the rule contains terminals, then

$$S_{RD} = \sum_k RD(K).$$

We have

$$N_G \leq P_G \leq S_{RD} \leq S_R \leq S_G.$$

(3.4.2) $N_G \leq P_G \leq V_R \leq S_R \leq S_G$

There is no ordering between V_G and S_{RD}.

3.5 Time Complexity of Part I of Mclean-Johnston Algorithm

Although the Mclean and Johnston Algorithm is found to be of $O(N*|G|)$, there are cases in which it performs very excellently, particularly if the production rules are cleverly arranged.

Theorem 3.5. The Mclean-Johnston algorithm is at most of $O(N*S_R)$.

See Appendix I for a full text of the algorithm.

Proof At line 21 of the algorithm a next variable is selected, also at line 26 a next production rule associated with the selected variable is selected and finally for the selected production rule, at line 41 the next symbol is selected.

Thus, for going through the algorithm from line 21 to 118, we examine each symbol on the right hand side of each production rule at least once. This is S_R number of examinations.

Assuming the delete on line 104 is not effected, then the algorithm will repeat lines 21 to 118 at most N times before stopping. This then gives $N*S_R$ total number of examinations.

Counting the number of comparisons, we have at lines 43 and 47 a comparison for each symbol on the left; at line 55, one for each variable on the left (counting repetitions); at lines 61 and 70 one comparison for each production rule. And at line 78 three for each

variable on the right. Thus, for going through lines 21 to 118 once we have

$$3*S_R + V_R + 2*N_G + 2*P_G$$

comparisons. And since the lines are repeated at most N times before the algorithms stops we have

$$N*(3*S_R + V_R + 2*N_G + 2P_G) \leq N*(8S_R).$$

And by the ordering earlier given, we have

$$O(N*S_R).$$

This order is true whether the grammar is in Chomsky, Greibach or in any other form, because the inequalities (3.4.1) are also fulfilled in all these cases.

Although the deletion on line 105 saves time it does not change the time complexity of the algorithm. This can easily be seen by assuming uniformity, that is, the total number of symbols (counting duplications) on the right of all production rules associated with each variable is the same.

If this is R', then by not deleting we shall have $O(N^2*R')$, while by deleting we have $O((N(N-1)/2)*R')$. Both of these are of $O(N*S_R)$, since $S_R = N*R'$ in this case.

Example: G = ({A,B,C,D}, {a}, P, A) with production rules

A:: = BB...B (m times)

B:: = CC...C (m times)

C:: = DD...D (m times)

D:: = aa...a (m times)

The length of the shortest terminal string derivable from A is m^4. This example is the worst case for the algorithm. Only one rule can

be deleted at each time the algorithm goes through lines 21 to 118.

However, if the above example were arranged in reverse order

D:: = aa...a (m times)

C:: = DD...D (m times)

B:: = CC...C (m times)

A:: = BB...B (m times)

then the algorithm will go through lines 21 to 118 only once.

IV. A NEW ALGORITHM

A thorough examination of the Mclean-Johnston Algorithm reveals several areas in which one can incorporate time saving devices. For example, at line 55 on encountering a variable X for which E(X) is false, one can discontinue processing the production rule and move to another production rule. This suggests using only variables for which E(X) is true for updating and processing any production rule.

This itself is not the ideal situation. The ideal situation we feel, is suggested by theorem 2.8, since only "shortest" production rules are involved in producing a shortest terminal string, then only such rules should be used in updating any production rule, not temporary shortest production rules!

An algorithm based on this principle will require updating (rewriting as terminals) each occurrence of a variable on the right hand side of a production rule only once and therefore will be at most of $O(S_R)$. Since to convert a production rule to a terminal string will require rewriting each variable in the rule at least once, this appears to be the best one can do.

Thus, such an $O(S_R)$ algorithm will be more efficient than the Mclean-Johnston algorithm which was of order $N*S_R$.

With a modified data structure, grouping together each occurrence of a variable on the right hand side of a production rule, and updating them together, we achieve an $O(S_{RD})$, where S_{RD} is as defined in chapter III.

Although our algorithm aims primarily at obtaining the shortest length of a terminal string derivable from a variable, it stores also

the production rule used for achieving this. Thus it accomplishes everything the Mclean-Johnston Algorithm does.

4.1 The Description of the New Algorithm

We associate with each production rule P defined by $A ::= A_1A_2 \ldots A_m$, A_k in V, a linear equation

(4.1.1) $A = \alpha + \sum_{j=1}^{N} c_jA_j$, A_j in V_N, and c_j the number of times A_j occurs in the production, $\alpha \in V_T$, a bundling together of all terminals in P.

Then if L(A) denotes the length of the shortest terminal string derivable from A, we define $L(P) = |\alpha| + \sum_{j=1}^{N} L(A_j)*c_j$, then

(4.1.2) $L(A) = \min_{P} (L(P))$, P a production rule associated with A.

Our Algorithm can now be described simply as follows:

(4.1.3) 0. Initialize the latest symbols with shortest terminal strings to V_T. k = 0.

1. Use the set of the latest symbols with shortest terminal strings to update the remaining production rules.

2. Determine the length of the next set of shortest terminal strings. Set this to LL.

3. For each variable A for which no shortest terminal string is known, determine L(A) according to (4.1.2). If L(A) = LL, then LL is the length of the shortest terminal string derivable from A. Accumulate such variables in the set S_k, as the set of the latest symbols with shortest terminal strings. Delete all their production rules. If set empty then STOP, else increase k by 1 and go to 1.

Example:

$A_1 :: = A_2a|b$

$A_2 :: = A_1A_2|a$

$A_3 :: = A_1A_1A_2|aA_2$

$k = 0$	$S_o = \{a,b\}$,	$A_1 \rightarrow b$,	$LL = 1$,	$L(A_1) = 1$,	$L(A_2) = 1$.
		$A_2 \rightarrow a$		$P(A_1) = 2$,	$P(A_2) = 4$.
$k = 1$	$S_1 = \{A_1,A_2\}$	$A_3 \rightarrow bba$	$LL = 2$,	$L(A_3) = 3$,	
		$A_3 \rightarrow aa$		$P(A_3) = 6$	
$k = 2$	$S_2 = \{A_3\}$				
$k = 3$	$S_3 = \emptyset$.				

To be able to establish the complexity of our algorithm we introduce a parameter k as a counter: each time we go through steps 1 to 3 we increase k by 1, and denote the set of latest symbols with shortest terminal strings by S_k. At initialization $k = 0$.

We denote the remaining set of variables by $\bar{S}_k$, and define it recursively as

$$\bar{S}_0 = V_N$$
$$\bar{S}_k = \bar{S}_{k-1} - S_k$$

Let $k = k_0$ whenever the algorithm stops.

Theorem 4.1: $k_0 \leq N + 1$.

Proof: Obvious. Since if for every k, $o \leq k \leq N$, $S_k \neq \emptyset$, then $|\bar{S}_{k+1}| \leq |\bar{S}_k| - 1$. Therefore $\bar{S}_N = \emptyset \rightarrow S_{N+1} = \emptyset$.

Theorem 4.2: For $A \in S_k$, denote $L(A)$ by L_k, then L_k is the length of a shortest terminal string derivable from A.

Proof: Again obvious, since $L_k \leq L_{k+j}$, $j \geq 0$.

From theorem 2.8 and lemma 2.6, we have the corallary;

Corollary 4.3. For each shortest terminal string, there is a derivation tree whose depth is at most N.

Theorem 4.4. If $A \in \bar{S}_{k_0}$, then no terminal string can be derived from A.

Proof: Let $\bar{T}_k$ denote the terminal strings generated during the k step. Since the algorithm stops when S_k is empty, it follows that T_{k_0} is empty.

Suppose A is in $\bar{S}_{k_0}$ then every production rule associated with A is of the form $A:: = w_1 A_1 w_2$, A_1 in $\bar{S}_{k_0}$. If this were not so, let $A:: = A_1 A_2 \ldots A_m$ be a production rule associated with A which violates the condition. Let r' be minimum r such that $A_1 A_2 \ldots A_m$ is in $(\bigcup_{n=1}^{r} S_k)^*$ Then $L(A) > L_{r'+1}$, otherwise A would be in $S_{r'+1}$.
And for $k > r'$ $L(A) > L_k$. In particular $L(A) > L_{k_0} - 1$ therefore S_{k_0} is not empty, since it must now contain at least A. Which is a contradiction of the definition of k_0.

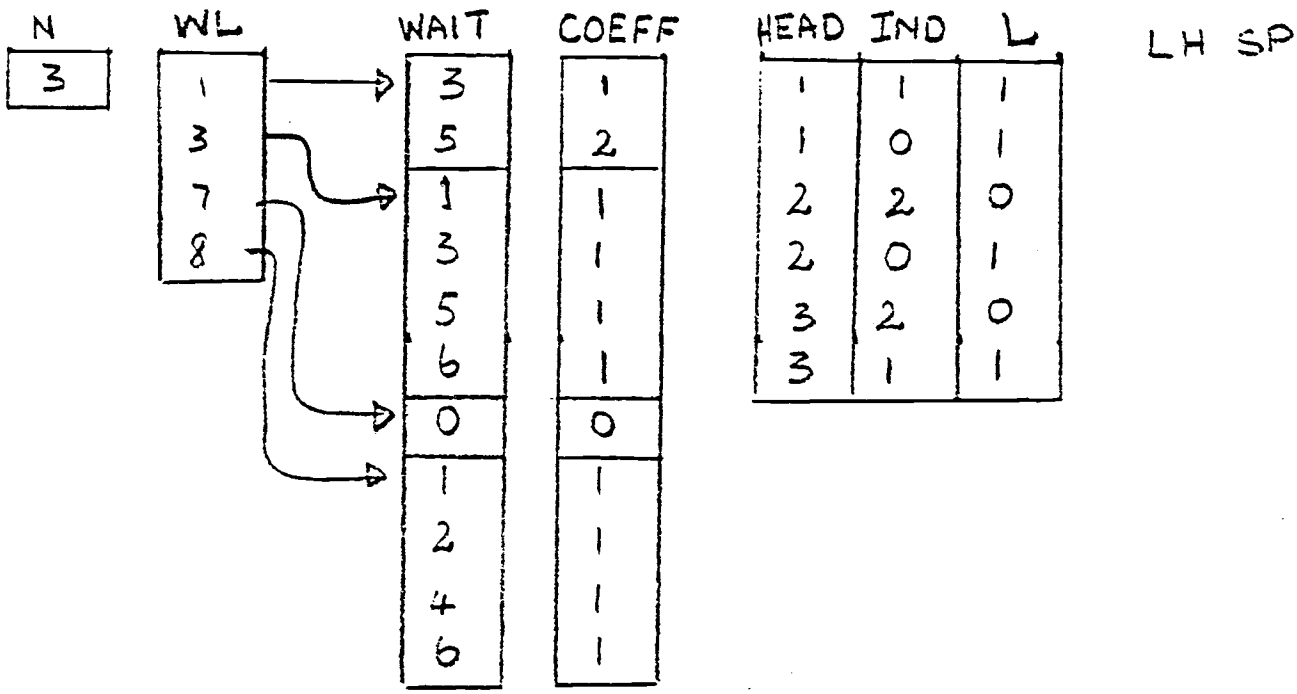
We have therefore shown that the algorithm works and it stops in at most N+1 steps. The additional step is introduced to enable us proof the correctness of the algorithm.

The present algorithm is more efficient than the Mclean-Johnston Algorithm, since it re-writes only those variables for which shortest terminal strings have been found.

Since re-writing all the occurrences of a variable in a production rule requires at most S_{RD} re-writings for the entire algorithm, it follows that the algorithm is at worst of $O(S_{RD})$.

4.2 The Data Structure for the New Algorithm

The data structure we are going to describe is suggested by the equation (4.1.1).



$A_1 :: = A_2a \mid b$

$A_2 :: = A_1A_2 \mid a$

$A_3 :: = A_1A_1A_2 \mid aA_2$

Figure 4.1. Data Structure for the New Algorithm

The ε-rule $A :: = \varepsilon$, is indicated by IND(P)=0 and L(P)=0.

Figure 4.1 contains all the main arrays in the data structure. The production rules in which a variable appears on the right hand side are stored in increasing order in the list WAIT, and entering in the last segment those production rules in which terminals are involved.

In the list COEFF are stored the number of times a variable appears in a production rule, in a position corresponding to the position of that rule in the variable's segment in WAIT.

The list WL indicates where the sequence of the production rules in which a particular variable is involved begins in WAIT. (As the name suggests WAIT refers to production rules waiting to be updated by the variable under whose segment they appear).

We use a few other arrays such as HEAD, IND, L, LH, and SP to indicate respectively, the left hand side of a production rule, the number of variables still in the production rule, the number of terminals in the production rule, the length of the shortest terminal string known for the variable and the production rule used in obtaining this.

The algorithm works briefly as follows:

Begin the k^{th} step: If IND(j) = 0, then the j^{th} rule has produced a terminal string, L(j) stores the length of that string while we compare to see whether it is the shortest terminal string for HEAD(j) if so, we store its length in LH(HEAD(j)) and the production rule, i.e. j in SP(HEAD(j)). If L(j) is of the shortest length so far for this step, we add HEAD(j) to the list of variables in S_k and set L_k = L(j).

If we are still interested in being able to prevent re-writing production rules associated with variables for which shortest terminal strings have been found, then we will still have to modify our data structure further:

We introduce another list WPHEAD to replace HEAD which stores the left hand side of a production rule, in ascending order, in which a variable is involved and for each entry in WPHEAD there is a POINTER

indicating where the sequence of production rules begin in WAIT. Thus, if WPHEAD (K) has no shortest terminal string we use the POINT(K) to access production rules associated with WPHEAD (K) in which the current variable is involved and then update.

For the example in Figure 4.1 we shall have the following layout:

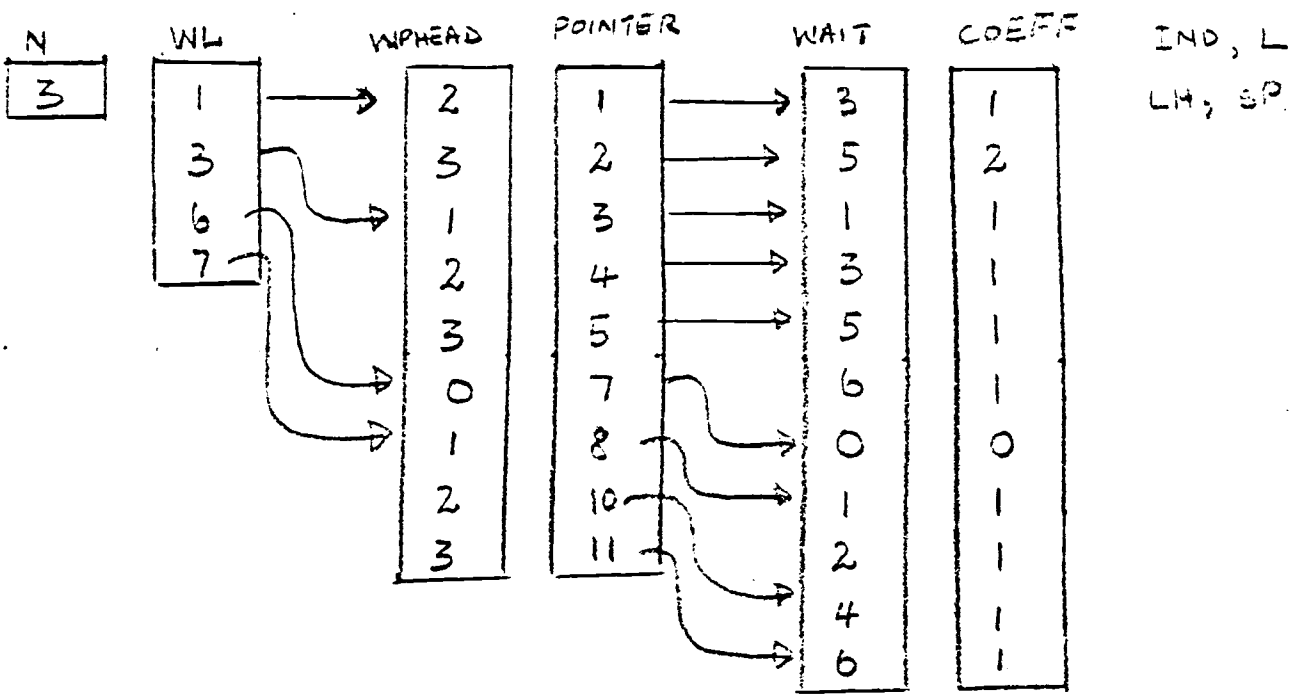


Figure 4.2. Modified Data Structure.

This data structure enables us to gather information about the HEAD of a production rule before accessing the rule.

For special grammars such as linear, regular and Chomsky Normal Form our algorithm is of $O(P_G)$. Because a linear and a regular grammar

have at most a variable on the right hand side of a production rule and a CFG in the Chomsky Normal Form has at most two variables on the right hand side of each production rule.

4.3 Computer Program for the New Algorithm

We shall adopt the data structure in Figure 4.1 and use the array STAR - which will indicate whether or not a shortest terminal string has been found for a variable to achieve the effect of the data structure in Figure 4.2.

We present the program in three procedures.

1. STSTRING - This is the main procedure.
2. SELECT - This procedure determines the length LL of the next set of shortest terminal strings and generates the set VIP which is the set of variables whose shortest terminal strings are of length LL. Finally it stores the remaining variables which have, already, terminal strings derived from them but no shortest terminal strings yet, in the set WSS.

   If VIP = Ø, the algorithm stops.
3. UPDATE - This procedure uses the set VIP to generate a new set of terminal strings and generate the set SS - the set of variables without shortest terminal strings which were terminal strings derived from them through the current UPDATE.

The individual arrays are as explained in Section 4.2.

L - stores the current number of terminal symbols in a production rule.

LH - stores the shortest length of the terminal string known so far for each of the variables.

SP - stores the productionrules used in generating LH.

The detailed program now follows:

```
 1.  PROCEDURE  STSTRING (WL, WAIT, COEFF, HEAD, IND, L, LH, SP, N, MM, PP);
 2.  INTEGER  N, MM, PP;
 3.  INTEGER ARRAY  WL, WAIT, COEFF, HEAD, IND, L, LH, SP;
 4.  BEGIN
 5.       INTEGER  I, J, K, KK, LHH, LP, P, HP, LL, K1, LL,
 6.       COUNT, COUNT1, KK1;
 7.       INTEGER ARRAY  VIP, WSS, SS [1:N ], STAR, CHARGE [1:N];
 8.       FOR I:=1  STEP 1  UNTIL N  DO
 9.       VIP[I]:=SS[I]:=WSS[I]:=0;
10.       FOR I:=1  STEP 1  UNTIL N  DO
11.       STAR[I]:=CHARGE[I]:=0;
12.       OLDCOUNT1:=KK:=COUNT:=COUNT1:=0;
13.       K1:=0;
14.  L1:  SELECT (VIP, SS, WSS, CHARGE, LL, KK, COUNT, OLDCOUNT1, COUNT1, K1);
15.       IF KK=0 THEN GO TO FIN;
16.       FOR I:=1 STEP 1 UNTIL COUNT DO
17.       SS[I]:=0;
18.       FOR I:=1  STEP 1 UNTIL N DO
19.       CHARGE [I]:=0;
20.       KK1:=KK:  COUNT:=0;
21.       UPDATE (VIP, SS, CHARGE, KK1, COUNT, LL);
22.       K1:=K1+1;
23.       COUNT1:=0;
24.       FOR I:=1 STEP 1 UNTIL COUNT DO
25.       VIP [I]:=0;
26.       KK:=0;
27.       GO TO L1;
```

```
28.  FIN:  OUTPUT (LH, SP, STAR, K1);
29.  END;

30.  PROCEDURE  UPDATE (VIP, SS, CHARGE, KK1, COUNT, LL);
31.  INTEGER ARRAY  VIP, SS, CHARGE;
32.  INTEGER KK1, COUNT, LL;
33.  BEGIN
34.       INTEGER I, J, K, I1, I2;
35.       FOR J:=1  STEP 1 UNTIL KK1 DO
36.       BEGIN
37.           K:=VIP [J];
          IF WAIT [WL[K]] ≠ 0 THEN
          BEGIN
38.           I1:=WL[K];
39.           I2:=WL[K+1]-1;
40.           FOR I:=I1  STEP 1  UNTIL  I2  DO
41.           BEGIN
42.               P:=WAIT[I];
43.               HP:=HEAD[P];
44.               IF  STAR [HP]=0 THEN
45.               BEGIN
46.               COMMENT  HP HAS NO SHORTEST TERMINAL STRING YET;
47.                   LP:=L[P]:=L[P]+COEFF[I]*LL;
48.                   IND[P]:=IND[P]-1;
49.                   IF IND [P]=0 THEN
50.                   BEGIN
51.                   COMMENT WE HAVE A TERMINAL STRING FROM PRODUCTION P;
52.                       LHH:=LH[HP];
53.                       IF LP<LHH THEN
```

```
54.                         BEGIN
55.                         COMMENT  THE TERMINAL STRING IS SHORTER THAN THE
56.                         CURRENTLY HELD FOR HP;
57.                               LH[HP]:=HP;
58.                               SP[HP]:=P;
59.                               IF CHARGE [HP]=0 THEN
60.                               BEGIN
61.                               COMMENT  WE PUT HP IN SS;
62.                                   COUNT:=COUNT+1;
63.                                   SS[COUNT]:=HP;
64.                                   CHARGE[HP]:=1;
65.                               END;
66.                         END;
67.                     END;
68.                 END;
69.             END;
            END:
70.     END:
71. END;

72. PROCEDURE  SELECT(VIP, SS, WSS, CHARGE, LL, KK, COUNT, OLDCOUNT1, COUNT1, K1
73. INTEGER ARRAY  VIP, SS, WSS, CHARGE;
74. INTEGER  LL, KK, COUNT, OLDCOUNT1, COUNT1, K1;
75. BEGIN
76.     INTEGER I1, I2, I, J, K, X;
77.     IF K1=0  THEN
78.     BEGIN
79.         I1:=WL[N+1];
            IF WAIT [WL[N+1]] ≠ 0 THEN
            BEGIN
```

```
        COMMENT  MM IS THE DIMENSION OF WL;
            I1:=WL[N+1];
80.         I2:=MM;
81.         FOR I:=I1 STEP 1  UNTIL I2  DO
82.         BEGIN
83.             P:=WAIT [I];
84.             IF IND[P]=0  THEN
85.             BEGIN
86.                 LP:=L[P]:=COEFF[I];
87.                 HP:=HEAD[P];
88.                 IF CHARGE [HP]=0  THEN
89.                 BEGIN
90.                 COMMENT  PUT HP IN SS;
91.                     COUNT:=COUNT+1;
92.                     SS[COUNT]:=HP;
93.                     SP[HP]:=P;
94.                     LH[HP]:=LP;
95.                     CHARGE[HP]:=1;
96.                 END
97.                 ELSE
98.                 IF LP<LH[HP]  THEN
99.                 BEGIN
100.                COMMENT  THE NEW TERMINAL STRING IS SHORTER THAN THE
101.                ONE CURRENTLY HELD FOR HP;
102.                    LH[HP]:=LP;
103.                    SP[HP]:=P;
104.                END;
105.            END;
106.        END;
```

```
107.        FOR I:=2  STEP 1  UNTIL  COUNT DO
108.        IF  LH[SS[I-1]]<LH[SS[I]]  THEN
109.        BEGIN
110.            X:=LH[SS[I]];
111.            LH[SS[I]]:=LH[SS[I-1]];
112.            LH[SS[I-1]]:=X;
113.        END;
114.        KK:=0;
115.        COUNT1:=0;
116.        LL:=LH[SS[COUNT]];
117.        FOR I:=1  STEP 1   UNTIL  COUNT DO
118.        IF LH[SS[I]]=LL  THEN
119.        BEGIN
120.            KK:=KK+1;
121.            VIP[KK]:=SS[I];
122.            STAR[VIP[KK]]:=1;
123.        END
124.        ELSE
125.        BEGIN
126.            COUNT1:=COUNTL+1;
127.            WSS[COUNT1]:=SS[I];
128.        END;
129.            FORI:=1  STEP 1  UNTIL  COUNT DO
130.            BEGIN
131.                CHARGE [SS[I]]:=0;
132.                SS[I]:=0;
133.            END;
```

```
134.        END
135.        ELSE
136.        BEGIN
137.            FORI:=1  STEP 1  UNTIL  OLDCOUNT1  DO
138.            BEGIN
139.                IF CHARGE [WSS[I]]=0  THEN
140.                BEGIN
141.                COMMENT  WSS[I] HAS A TERMINAL STRING EARLIER BUT NONE
142.                THIS ROUND.
143.                PUT WSS[I] IN SS:
144.                    COUNT:=COUNT+1;
145.                    SS[COUNT]:=WSS[I];
146.                END;
147.                WSS[I]:=0;
148.            END;
149.        END;
150.        FOR I:=2 STEP 1  UNTIL  COUNT  DO
151.        IF LH[SS[I-1]] < LH[SS[I]]  THEN
152.        BEGIN
153.            X:=LH[SS[I]];
154.            LH[SS[I]]:=LH[SS[I-1]];
155.            LH[SS[I-1]];
156.        END;
157.        KK:=0;
158.        COUNT1:=0;
159.        LL:=LH[SS[COUNT]];
160.        FOR I:=1  STEP 1 UNTIL  COUNT  DO
```

```
161.            IF  LH [SS[I]]=LL  THEN
162.            BEGIN
163.            COMMENT PUT SS[I] IN VIP;
164.                KK:=KK+1;
165.                VIP [KK]:=SS[I];
167.                STAR [VIP[KK]]:=1;
168.            END
169,            ELSE
170.            BEGIN
171.            COMMENT  PUT SS[I] in WSS:
172.                COUNT1:=COUNT1+1;
173.                WSS[COUNT1]:=SS[I];
174.            END;
175.        END;
176.        OLDCOUNT1:=COUNT1;
177.  END;
```

The selection process is done in two halves, first for K1=0 and then for K1 > 0. This is why lines 107-128 are repeated in lines 150-174.

4.4 Analysis of the Time Complexity of the New Algorithm

The main procedure STSTRING calls the procedure SELECT at line 14 at most N times and similarly calls the procedure UPDATE at line 22 at most N times.

First we examine what happens when the procedure UPDATE is called:

UPDATE is the re-writing procedure. If $S_R^{(k)}$ and $P_G^{(k)}$ denote the number of symbols (variables) on the right hand side to be re-written during the k^{th} step (call of UPDATE) and number of production rules deriving terminal strings during this step respectively, then UPDATE makes the following number of comparisons:

at line 44	$S_R^{(k)}$
at line 49	$S_R^{(k)}$
at line 53	$P_G^{(k)}$
and at line 59	$P_G^{(k)}$

Summing up over k we have a total number of comparisons

$$2^* \sum_{k=0}^{N} (S_R^{(k)} + P_G^{(k)}) \leq 2^*S_R + 2^*P_G$$

Now, the SELECT procedure operates slightly differently: Let $H^{(k)}$ be the number of variables with terminal strings at step k.

SELECT handles the cases k = 0 and k ≠ 0 separately.

For k = 0, it makes $P_G^{(o)}$ comparisons at line 84, at line 88 or 98 it makes $H^{(0)}$ comparisons, while at lines 108 and 118 it makes $H^{(0)}$ comparisons to select the shortest length and the elements in VIP respectively. Thus we have a total of

$$P_G^{(0)} + 3^*H^{(0)} \text{ comparisons}$$

For $k \neq 0$, the relevant comparisons are made at lines 139, 151, and 161. At each of these lines a total of $H^{(k)}$ comparisons are made. Again we have a total of

$$3*H^{(k)} \text{ comparisons}$$

Adding we have

$$P_a^{(0)} + 3* \sum_{k=0}^{N} H^{(k)} \leq P_a + 3* \frac{N(N-1)}{2}$$

since $H^{(k)} \leq N-k$. And usually $H^{(k)} << N-k$.

Thus from both procedures we have at most

$$2*S_R + 3*P_G + 3* \frac{N(N-1)}{2} \quad \text{comparisons}$$

which by the earlier inequalities involving S_R, P_G and N is $O(S_R + N^2)$.

Except in very sparse cases, that is very few production rules per variable and each production rule having very few symbols on the right,

$S_R \geq N^2$ (in most cases greater than).

Therefore, we have finally $O(S_R)$.

Lemma 4.4: If there are no production rules of the type A::=B, $A,B \in V_N$ in G and if for every K, $0 \leq K \leq N$, $H^{(K)} = N - K$ then

$$S_R > \frac{N\,(N+1)}{2}$$

Proof: We impose the condition "no production rules of the type A::=B — $A,B \in V_N$ to insure that $L_K < L_{K+1}$, $1 \leq K \leq N$. L_K is the minimum length at step K.

$H^{(0)} = N \Rightarrow \forall A_j \in V_N$ there is a production rule $A_j::=\alpha_j$, $\alpha_j \in (V_T)$, $1 \leq j \leq N$. We number the variables in increasing order of magnitude of $|\alpha_j|$. If we can show that $|\alpha_j| \geq j$, $1 \leq j \leq N$, then the lemma follows; since

$$S_R \geq \sum_{j=1}^{N} |\alpha_j| \geq \sum_{j=1}^{N} j = \frac{N\,(N+1)}{2}$$

We shall again assume $|\alpha_j| \neq 0$.

Since $H^{(K)} = N - K$, $0 \leq K \leq N$, it follows that $|VIP| = 1$ at each step and in particular only $|\alpha_1| = \min_j |\alpha_j|$. Therefore $|\alpha_1| \geq 1$.

If for j, $1 \leq j \leq r < N$, $|\alpha_j| \geq j$, we claim $|\alpha_{r+1}| \geq r + 1$.

Suppose not, since $|\alpha_r| \geq r$ then $r \leq |\alpha_r| \leq |\alpha_{r+1}| < r + 1 \rightarrow$ $|\alpha_{r+1}| = r = |\alpha_r|$. This means $|\alpha_j| = j$, $1 \leq j \leq r$ and $L_j = j$.

Since $L_r = r = |\alpha_{r+1}| \rightarrow L_{r+1} = L_r$. This contradicts the fact that $L_{r+1} > L_r$. Therefore the lemma follows.

With this lemma we have now shown that our Algorithm is $O(S_R)$.

We give an example in which the set SS has N-k elements at the k^{th} step, $0 \leq k \leq N$.

A_1:: = a

A_2:: = bA_1|ab

A_3:: = bba|A_1A_2

A_4:: = aaaa|$A_1A_2A_3$

A_5:: = baaba|A_4A_1b

$N = 5$, $N^2 = 25$, $S_R = 25$, $\frac{N\,(N+1)}{2} = 15$

k	UPDATE	SS	L	VIP	WSS
0	$A_1 \rightarrow$ a	$A_1 \rightarrow$ a	1	$A_1 \rightarrow$ a	$A_2 \rightarrow$ ab
	$A_2 \rightarrow$ ab	$A_2 \rightarrow$ ab			$A_3 \rightarrow$ bba
	$A_3 \rightarrow$ bba	$A_3 \rightarrow$ bba			$A_4 \rightarrow$ aaaa
	$A_4 \rightarrow$ aaaa	$A_4 \rightarrow$ aaaa			$A_5 \rightarrow$ baaba
	$A_5 \rightarrow$ baaba	$A_5 \rightarrow$ baaba			

1	$A_2 \to ba$	$A_2 \to ab$	2	$A_2 \to ab$	$A_3 \to bba$
	$A_3 \to aA_2$	$A_3 \to bba$			$A_4 \to aaaa$
	$A_4 \to aA_2A_3$	$A_4 \to aaaa$			$A_5 \to baaba$
	$A_5 \to A_4ab$	$A_5 \to baaba$			
2	$A_3 \to aab$	$A_3 \to bba$	3	$A_3 \to bba$	$A_4 \to aaaa$
	$A_4 \to aabA_3$	$A_4 \to aaaa$			$A_5 \to baaba$
		$A_5 \to baaba$			
3	$A_4 \to aabbba$	$A_4 \to aaaa$	4	$A_4 \to aaaa$	$A_5 \to baaba$
		$A_5 \to baaba$			
4	$A_5 \to aaaaab$	$A_5 \to baaba$	5	$A_5 \to baaba$	$\emptyset$
5	$\emptyset$	$\emptyset$		$\emptyset$	$\emptyset$

V. CONCLUSIONS

Our algorithm is completely independent of the order of arrangement of the variables, and the production rules. This again is an added advantage.

In such cases as referred to in section 4.4 as exceptional cases the number of variables with terminal strings at each step is very small so that $H^{(k)} << N-k$. Therefore the overall order of the new algorithm is valid and should be expected to perform faster than the Mclean and Johnston Algorithm under most conditions.

It is not unlikely that absorbing part of the selection process done by the procedure SELECT in UPDATE itself might help step up the algorithm, whether this can improve upon the time complexity we are not sure.

So far we have not analyzed the expected time of both algorithms. A thorough analysis of this will no doubt be very informative; because an algorithm with the best worst case complexity is not necessarily the best algorithm as far as a expected time complexity is concerned.

We believe therefore analyzing the expected time complexity might lead to further improvements.

REFERENCES

1. Aho, Alfred and Ullman, Jeffery D. — The Theory of Parsing, Translation and Compiling, Vol. 1: Parsing. Prentice-Hall Series in Automatic Computation (1972).
2. Aho, A.V., Hopcroft J.E. and Ullman, J.D. — The Design and Analysis of Computer Algorithms. Addison-Wesley (1974).
3. Cremers, Armin and Ginsberg, Seymour. — Context-Free Grammar Forms: Comp. Syst. Sci. 11(1975) 86-117.
4. Forster, J.M. — Automatic Syntactic Analysis Macdonald (1970).
5. Froberg, Ekman — Introduction to Algol Programming OUP (1967).
6. Ginsberg, Seymour and Lynch, Nancy — Size Complexity in Context-Free Grammar Forms: J. ACM Vol. 23 No. 4 (1976) 582-598.
7. Gries, David — Compiler Construction for Digital Computers: John Wiley & Sons Inc. (1971).
8. Hopcroft, J.E. and Ullman, J.D. — Formal Lanaguages and Their Relation to to Automata. Addison-Wesley (1969).

9. Mclean Michael J. and Johnston, Daniel B. An Algorithm for finding the shortest terminal Strings which can be produced from non-terminals in CFG. 3rd Austr. Conf. University of Queensland May (1974). Combinatorial Mathematics III Springer Verlag, Lecture Notes in Mathematics Vol. 452, 180-196.

10. Salomaa, Arto. Formal Languages. Academic Press (1973).

APPENDIX I

THE COMPUTER PROGRAM FOR THE MCLEAN AND JOHNSTON ALGORITHM

We now give the computer program for the Part I of the Mclean and Johnston Algorithm as contained in [9].

```
1.  PROCEDURE  PART I (PP, D, N, P, L)
2.  COMMENT  THIS PROCEDURE IMPLEMENTS PART I OF THE
3.  ALGORITHM;
4.  INTEGER ARRAY  PP, D, P, L;
5.  BOOLEAN ARRAY  E;
6.  INTEGER N;
7.  BEGIN
8.      INTEGER ARRAY  G, LINK [1:N];
9.      INTEGER  X, I, J, K, OLDI, LASTCHANGED, SUMG, COUNT, SHORTEST;
10.     BOOLEAN  CHANGED, FIRST, FOUND;
11.     FORI:=1  STEP 1  UNTIL  N  DO
12.     BEGIN
13.         G [I]:=0;
14.         E[I]:=FALSE
15.         L[I]:=0;
16.         LINK[I]:=I+1;
17.     END;
18.     LINK[N]:=1;
19.     I:=N;
20.     LASTCHANGED:=N;
21. L1: OLDI:=I;
```

```
22.       FIRST:=TRUE;
23.       CHANGED:=FALSE;
24.       IF I=1
25.       THEN J:=1
26.       ELSE J:=D[I-1] + 1;
27.  L2:  IF J ≤ D[I]  THEN
28.       BEGIN
29.       COMMENT  WE PROCESS ANOTHER PRODUCTION RULE
30.       ASSOCIATED WITH I.
31.       COUNT STORES THE LENGTH OF SHORTEST TERMINAL STRING
32.       KNOWN SO FAR TO BE PRODUCEABLE FROM CURRENT
33.       PRODUCTION RULE.
34.       SUMG STORES THE ESTIMATED LOWER BOUND FOR THE
35.       LENGTH OF THE SHORTEST TERMINAL STRING PRODUCEABLE
36.       FROM CURRENT RULE;
37.            COUNT:=0;
38.            SUMG:=0;
39.            FOUND:=TRUE;
40.            K:=J;
41.  L3:       X:=PP[K];
42.            K:-K+1;
43.            IF X≠ 0 THEN
44.            BEGIN
45.            COMMENT  WE HAVE NOT REACHED THE END OF
46.            CURRENT PRODUCTION RULE;
47.                 IF X IS A TERMINAL THEN
48.                 BEGIN
```

```
49.                     SUMG:= SUMG + 1;
50.                     COUNT:= COUNT + 1;
51.                 END
52.                 ELSE
53.                 BEGIN
54.                     SUMG:= SUMG + G [X];
55.                     IF E [X]
56.                     THEN COUNT:= COUNT + L[X]
57.                     ELSE FOUND:= FALSE
58.                 END;
59.                 GO TO L3;
60.             END;
61.             IF FIRST THEN
62.             BEGIN
63.             COMMENT  WE HAVE FOR THE FIRST TIME IN THIS
64.             ROUND REACHED THE END OF A PRODUCTION RULE
65.             ASSOCIATED WITH I;
66.                 FIRST:=FALSE;
67.                 SHORTEST:=SUMG;
68.             END
69.             ELSE
70.             IF SUMG < SHORTEST THEN
71.             BEGIN
72.             COMMENT  SHORTEST STORES THE CURRENT
73.             ESTIMATE OF THE MINIMUM OF ESTIMATED LOWER
74.             BOUNDS OF THE LENGTH OF A SHORTEST TERMINAL
75.             STRING DERIVABLE FROM I;
```

```
76.             SHORTEST:=SUMG;
77.         END;
78.         IF FOUND AND NOT (E[I] AND COUNT ≤ L[I]) THEN
79.         BEGIN
80.         COMMENT  A TERMINAL STRING SHORTER THAN THE
81.         PREVIOUS ONE HELD FOR I HAS BEEN FOUND;
82.             L[I]:=COUNT;
83.             E[I]:=TRUE;
84.             P[I]:=J;
85.             CHANGED:=TRUE;
86.         END;
87.         J:=K;
88.         GO TO L2;
89.         END;
90.         IF SHORTEST G[I] THEN
91.         BEGIN
92.         COMMENT  THE LOWER BOUND OF THE LENGTH OF
93.         A SHORTEST TERMINAL STRING DERIVABLE FROM I
94.         HAS INCREASED;
95.             G[I]:=SHORTEST;
96.         END;
97.         IF E[I] AND G[I] = L[I] THEN
98.         BEGIN
99.         COMMENT  A SHORTEST TERMINAL STRING HAS
100.        BEEN FOUND FOR I;
101.            IF I = LINK [I] THEN
102.        GO TO FIN
103.        ELSE
```

```
104.          BEGIN
105.          COMMENT  DELETE I;
106.               LINK [OLDI]:=LINK[I];
107.               I:=LASTCHANGED:=OLDI;
108.               GO TO L1;
109.          END;
110.          IF CHANGED THEN
111.          BEGIN
112.               LASTCHANGED:=I;
113.               GO TO L1;
114.          END
115.          ELSE
116.          IF LASTCHANGED ≠ I THEN
117.          GO TO L1;
118.     END;
FIN: END;
```

APPENDIX II

THE PART II OF THE MCLEAN AND JOHNSTON ALGORITHM

Having obtained for each variable A the length of a shortest terminal string and a production rule P(A) that can be used for deriving a shortest terminal string, if one exists, the derivation tree of the string is constructed as follows:

(A.II.1)
```
IF A IS A TERMINAL THEN
    OUTPUT SYMBOL A
    ELSE
    BEGIN
        LET P(A) BE THE RULE
        A::=A1A2...Am
        FOR K:=1 STEP 1 UNTIL M DO
        CONSTRUCT THE SHORTEST TERMINAL STRING OF AK;
    END;
    REPEAT FOR ALL VARIABLES A FOR WHICH E(A) IS TRUE;
```