To the Graduate Council:

I am submitting herewith a thesis written by George H. Miles entitled "An Algorithm for Error Correction in Context-Free Languages." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael G. Thomason 7/25/74
Michael G. Thomason, Major Professor

We have read this thesis
and recommend its acceptance:

R.C. Gonzalez

Sara R. Jordan

Accepted for the Council:

Vice Chancellor
Graduate Studies and Research

AN ALGORITHM FOR ERROR CORRECTION IN CONTEXT-FREE LANGUAGES

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee

George H. Miles

August 1974

ACKNOWLEDGEMENTS

The author wishes to thank his wife for her support and encouragement during the preparation of this thesis.

Appreciation is also expressed to his parents, especially to his mother who faithfully typed a thesis which is totally unintelligible to her.

A special "thank you" is given to Dr. Michael G. Thomason for his suggestions which resulted in this thesis.

1165451

ABSTRACT

The task of error correction in various applications of parsing is drawing more and more interest and concern. This paper presents an algorithm that corrects three common spelling errors in a context-free language and then describes an implementation on a digital computer.

The algorithm described initially applies transformations to the original grammar to produce three new grammars. Each new grammar will generate the strings in the original language with one occurrence of an error. Each new grammar is then used to define a pushdown transducer which will recognize a string in the error corrupted language. The output will be a string in the original language along with a special symbol to show where the error correction has occurred.

TABLE OF CONTENTS

SECTION		PAGE
INTRODUCTION		1
I.	DEFINITIONS	5
II.	NEW GRAMMARS	8
III.	SPECIFICATION OF PDT	14
	Formal Definitions	14
	Data Structures and Extraction Process	18
IV.	SIMULATION OF PDT	25
V.	CONCLUSION	35
BIBLIOGRAPHY		37
APPENDIX		39
VITA		56

INTRODUCTION

The primary function of a compiler is to translate the input source language into machine code. This typically consists of two parts--an analysis of the structure of the input called syntax analysis or parsing, and the mapping of the structured input to the machine language called semantic mapping. The process could move quite smoothly if it were not for the fact that the compiler must be prepared to handle incorrect input in the parsing stage. There are many different types of errors that range from misspelling of keywords or variables to omission of necessary symbols or even words.

In recent years, in an effort to make design of language processors more precise, a formal means of defining programming languages has been searched for. One approach has been the concept of formal grammars. A certain class of grammars, context-free grammars, can formalize most of the syntax of a programming language. Some parsing algorithms are based upon the structure of context-free grammars.

The problem of handling errors has not been resolved by this formalism. However, some algorithms will announce an error when a position in the input is reached such that there is no correct way to parse the remainder. These include Early's algorithm and the LL and LR parsing algorithms. The latter two call for a special subset of the class of context-free grammars. (Aho and Ullman, 1972)

Some compilers will attempt to make changes of some minimal nature to correct a string that is incorrect. Such compilers as CORC and PL/C (Conway and Maxwell, 1966; Morgan and Wagner, 1971) go even beyond this and attempt extensive error correction.

The problem of error correction is not limited to the area of language translators. Online control systems must give instant response to commands and often do not have time to prompt for a correct response. In computer-aided instruction, misspellings or incorrect syntax may be the norm rather than the exception. Speech recognition systems operate in an environment that demands quick response and where ill-formed input is a fact of life.

Morgan (1970) devised a way to correct spelling errors by means of a table lookup. Keywords were input to the algorithm, and all possible misspellings were calculated and stored. Then if a parse encountered an unrecognizable word, a tabular search was instituted to see if it were a misspelled keyword. If so, the word could then be corrected.

Leinius (1970) presented a simple-precedence parser which generated a stack discipline for recovering from errors without interfering with normal processing.

Wagner (1972; 1974) has developed algorithms for context-free and regular languages that employ a "preprocessor." The error-correction is done before parsing begins. The output is guaranteed to be syntactically legal, but is different from the input only if the input is illegal.

Lyon (1972; 1974) has developed several least-errors recognizers aimed toward practicality. The first is based on Younger's recognizer while the second uses Early's because of greater efficiency.

Little is known mathematically, however, about these, or other, means of error correction. Thomason (1974) has developed a formalized algorithm for error correction. Given a context-free grammar in Greibach Normal Form, the algorithm will generate pushdown transducers capable of correcting certain classes of errors. The technique employs transforms on the productions of the original grammar to generate new grammars which produce error-corrupted strings. These grammars then are used to define the pushdown transducer which will output a string in the original language.

This paper describes an implementation of this algorithm on a digital computer. It should be stressed that this algorithm is formal in nature and no claims of efficiency are made for its implementation.

The classes of errors that are corrected are a subset of the ones given by Damerau (1963). He found that over 80% of the misspellings in source programs were caused by one of four errors:

(1) Deleting one symbol, denoted an α error;

(2) Inserting one symbol, denoted a β error;

(3) Changing one symbol into another, denoted a δ error;

(4) Transposing two adjacent symbols.

Since the fourth type of error can be considered a subset of two δ errors, the algorithm presented and implemented only handles α, β, and δ errors.

I. DEFINITIONS

A context-free grammar, appropriately denoted CFG, is a four-tuple in $G = (N,T,S,P)$ where

N is a finite set of nonterminals or variables,

T is a finite set of terminals, disjoint from N,

$S \in N$ is the start symbol,

P is a finite set of productions of the form $A \rightarrow \gamma$ where $A \in N$ and $\gamma \in (N \cup T)^*$ where X^* represents the set of all strings consisting of characters from the set X including λ, the empty string.

A sentential form of $G = (N,T,S,P)$ is defined recursively as follows:

(1) S is a sentential form,

(2) If $\alpha\beta\gamma$ is a sentential form and $\beta \rightarrow \delta$ is a production in P, then $\alpha\delta\gamma$ is also a sentential form.

A sentential form with no nonterminals is called a sentence generated by G. The set of all sentences generated by G is the language of G, denoted by L(G).

A CFG $G = (N,T,S,P)$ is said to be in Greibach Normal Form (GNF) if every production in P is of the form $A \rightarrow a\gamma$ where $A \in N$, $a \in T$, and $\gamma \in N^*$ (γ may be λ, the empty string). Grammars G_1 and G_2 are equivalent if $L(G_1) = L(G_2)$. Given an arbitrary CFG G_1, there exist algorithms for obtaining an equivalent G_2 in GNF (Aho and Ullman, 1972, Algorithms 2.14 and 2.15).

The standard notation for a set of GNF productions for $A \in N$ is:

$$A \rightarrow a\gamma_1 \mid b\gamma_2 \mid c\gamma_3 \mid \ldots$$

where a, b, c, . . . $\in T$, and $\gamma_1, \gamma_2, \gamma_3, \ldots \in N^*$. This notation can be represented in a one-to-one correspondence by a set of defining equations in associative but noncommuting factors of the form

$$A = a\gamma_1 + b\gamma_2 + c\gamma_3 + \ldots$$

(Aho and Ullman, 1972). This latter representation will be used throughout the remainder of the paper.

Given a CFG G and a string $w \in T^*$, a device (machine) is needed that will determine if $w \in L(G)$. Such a device is the pushdown automation (PDA), defined as a six-tuple $P = (Q, I, L, f, q_0, Z_0)$ where

- Q is a finite set of states,
- I is a finite input alphabet,
- L is a finite pushdown list alphabet,
- f is a mapping from $Q \times (I \cup \{\lambda\}) \times L$ into finite subsets of $Q \times L^*$,
- $q_0 \in Q$ is the starting state,
- $Z_0 \in L$ is the initial pushdown list symbol.

The f map of a PDA which halts with an empty pushdown list if and only if the input w is in L(G) is directly obtainable from the productions P of G in GNF as follows:

$$(q_1, \gamma) \in f(q_1, a, A) \text{ if } A \rightarrow a\gamma \in P.$$

That is, the PDA, in state q_1, with the input symbol a being scanned and the pushdown symbol A on top of the stack, will move to state q_1, and replace A on top of the stack by γ if a production of the form $A \rightarrow a\gamma$ is found in P. Any CFL can be

recognized by a single-state PDA which operates in real time, i.e. which reads one input symbol for each transition (Hopcroft and Ullman, 1968, Theorem 5.2). It is important to note that a PDA is a formal model of a top-down parse algorithm which is in general nondeterministic in operation.

However, the PDA is not sufficient to correct a string. A pushdown transducer (PDT) can be used to correct strings since it does output symbols. A PDT is defined as a seven-tuple $T = (Q,I,L,Y,f,q_o,Z_o)$ where Q, I, L, q_o, and Z_o are the same as for a PDA and

Y is a finite output alphabet,

f is a mapping from $Qx(I \cup \{\lambda\})xL$ into finite subsets of QxL^*xY^*.

For the purposes of this paper, Y will consist of $(I \cup \{<\})$ where the symbol < is used to denote that a correction involving one symbol was made immediately before this symbol appears. The output of a PDT will therefore be a string $w \in L(G)$ with the possible occurrence of the symbol < to denote an error correction.

Since a PDA is nondeterministic, then so is a PDT as defined above. The PDT is also nondeterministic because there may exist more than one way to correct an incorrect string, and it is impossible to guarantee which correction will result.

Let an e error represent an α, β, or δ error where all three error types may be discussed in a similar manner. That is, to talk about an e error is to talk about an error that may be either an α, β, or δ error.

II. NEW GRAMMARS

The first step in the specification of an error correcting PDT is to generate a new grammar G_e from G such that

$L(G_e) = \{w \mid w \in T^*$, w is obtained from a string in L(G) by the occurrence of one e error$\}$

where e is taken to mean either α, β or δ. Therefore, three new grammars must be generated: G_α, G_β, and G_δ. To do so, transformations are applied to the defining equations of G as follows:

(1) for $A \in N$,

$e(A) = A_e$ a new nonterminal;

(2) for $X, Y \in (T \cup N)^*$,

$e(X + Y) = e(X) + e(Y)$

$e(XY) = e(X)Y + Xe(Y)$;

(3) for $a \in T$,

$\alpha(a) = \lambda$

$\beta(a) = (\{b \mid b \in T\})a + a(\{b \mid b \in T\})$

$\delta(a) = \{b \mid b \in T,\ b \neq a\}$.

New variables are introduced via rule (1) that, along with the original set N, make up an augmented set N_e, the set of nonterminals for the error-generating grammar G_e.

Rule (2) may be explained as follows: the distributivity of the transformations through + shows that the error may occur through the use of any one production. The second equation indicates that the error may be in a lefthand substring with the righthand substring correct or vice versa.

Rule (3) represents the actual effect of the error on a terminal, different for each error. In the case of the α error, the effect is the deletion of the terminal; in the case of the β error, an extra terminal is introduced, either before or after the current terminal; finally, in the case of the δ error, the current terminal is mapped to another terminal.

As an example, consider the grammar G = ({E,G,T,Q,F,P}, {(,X,/,X,)},E,P) with the following defining equations:

E = (EP + X + (EPQ + XQ + (EPG + XG + (EPQG + XQG,

G = /T + /TG,

T = (EP + X + (EPQ + XQ,

Q = *F + *FQ,

F = (EP + X,

P =).

The language L(G) is the set of arithmetic expressions in one variable (X) involving multiplication and division and using matched parentheses.

Consider the effect of each transformation--α, β, and δ--on the defining equation F = (EP + X:

$F_\alpha = \alpha(F)$

$= \alpha(\ (EP + X\ )$

$= \alpha(\ (EP\ ) + \alpha(X)$

$= \alpha(\ (\ )EP + (\alpha(E)P + (E\alpha(P) + \alpha(X)$

$= \lambda EP + (E_\alpha P + (EP_\alpha + \lambda;$

$$F_\beta = \beta(F)$$
$$= \beta(\ (EP + X\ )$$
$$= \beta(\ (EP\ ) + \beta(X)$$
$$= \beta(\ (\ )EP + (\beta(E)P + (E\beta(P) + \beta(X)$$
$$= (\ ((\ +\ )(\ +\ ()\ +\ *(\ +\ (*\ +\ X(\ +\ (X\ +\ /)\ +\ (/)\ EP$$
$$+\ (E_\beta P + (EP_\beta + (X + X(\ +\ )X + X(\ +\ *X + X* + XX$$
$$+\ /X + X/;$$

$$F_\delta = \delta(F)$$
$$= \delta(\ (EP + X)$$
$$= \delta(\ (EP\ ) + \delta(X)$$
$$= \delta(\ (\ )EP + (\delta(E)P + (E\delta(P) + \delta(X)$$
$$=)EP + *EP + XEP + /EP + (E_\delta P + (EP_\delta + (\ +\ )\ +\ *\ +\ /.$$

The new grammar G_e then is the four tuple $G = (N_e, T, S_e, P_e)$ where

N_e is a finite set of terminals consisting of N from G and the new nonterminals formed by application of the transformations (if $A \in N$, then $A, A_e \in N_e$),

T is a finite set of terminals (the same as in G),

$S_e \in N_e$ is the new starting symbol,

P_e is the set of productions consisting of both the original productions P and the new productions formed by the transformations.

The task of forming these three new grammars consisted almost exclusively of forming the new productions. This task could be handled relatively easily since the original grammar was in GNF. This meant that one and only one terminal

appeared in each term of the defining equation and it was always the first character encountered.

Informally, it should be obvious that for each leftmost derivation in G, there exists parallel derivations in G_e in which at most one subscripted variable occurs at each step. When that subscript disappears, the error has occurred, and the derivation is continued using the original productions.

Formally, let $L(G)_e$ denote $\{x \mid x \in T^*$, x is obtained from string(s) in L(G) by one e error$\}$. We need to show $L(G_e) = L(G)_e$. To do so, we shall first show $L(G_e) \subseteq L(G)_e$ and then $L(G)_e \subseteq L(G_e)$. We shall make use of the one-to-one correspondence between strings in L(G) and the frontiers of derivation trees obtained from G (Aho and Ullman, 1972).

Note first that $P \subseteq P_e$. Additionally there are the following similarities between productions in P and P_e:

(1) For each production $A \rightarrow c\alpha B\beta$, $A,B \in N$, $\alpha,\beta \in N^*$, $c \in T$, in P there is a new production $A \rightarrow c\alpha B_e\beta$ in P_e which is identical except for the subscript.

(2) For each production $A \rightarrow a\gamma$, $A \in N$, $a \in T$, $\gamma \in N^*$, in P there are new productions of the form $A \rightarrow b\gamma$ in P_e.
For $e = \alpha$, there is only one such production, where $b = \lambda$.
For $e = \beta$, there are several resulting productions, where $b = ca$ or $b = ac$, for all $c \in T$.
For $e = \delta$, there are also several resulting productions, where $b \in T$, $b \neq a$.

Let T_e denote a G_e derivation tree of m levels with root S_e and frontier $w \in T^*$. Due to the form of the productions in P_e, there is in T_e a level n, $1 \leq n \leq m$, such that exactly one subscripted variable appears in each level 1, 2, . . ., n, but none appears at levels n+1, . . ., m; a production of the form $A_e \rightarrow b\gamma$ is used at level n. We can therefore write w = xby where x is the frontier prefix generated at levels 1 through n, and y is the frontier prefix generated at levels n+2 through m; b is the output generated at level n+1. Remember that b may be λ, a single terminal, or two terminals.

A corresponding G derivation tree T with m levels, root S and frontier z can be obtained as follows:

(1) For each production of the form $A_e \rightarrow c\alpha B_e\beta$ used in T_e, use the corresponding production $A \rightarrow c\alpha B\beta$ at the same location in T; this will reproduce the prefix x along the T frontier.

(2) At level n, where a production of the form $A_e \rightarrow b\gamma$ is used in T_e, use $A \rightarrow a\gamma$ in T.

(3) Complete T at levels n+1 through m by using the same productions used in T_e; this will reproduce the suffix y along the T frontier.

We can write z = xay. Thus any string w = xby in $L(G_e)$ has at least one corresponding string z = xay in L(G) differing by the occurrence of an e error. Therefore $L(G_e) \subseteq L(G)_e$.

Similarly, let T be a G derivation tree of m levels with root S and frontier $z = xay$, $a \in T$, $x, y \in T^*$, where a is introduced at level n by a production of the form $A \rightarrow a\gamma$.

There exists a G_e derivation tree of m levels with root S_e and frontier w obtained as follows:

(1) For each production of the form $A \rightarrow c\alpha B\beta$ in T through level n, use the corresponding production $A \rightarrow c\alpha B_e\beta$ in T_e; this will reproduce prefix x along the T frontier.

(2) At level n, where a production of the form $A \rightarrow a\gamma$ is used, in T, use $A_e \rightarrow b\gamma$ in T_e.

(3) Complete T_e at levels n+1 through m by using the same productions in T (since the subscript has disappeared); this will reproduce the suffix y along the T_e frontier.

We can write $w = xby$. Thus any string $x = xay$ in L(G) has at least one corresponding string $w = xby$ in $L(G_e)$. Therefore $L(G)_e \subseteq L(G_e)$.

Finally, we have $L(G_e) = L(G)_e$, so that the new grammars will generate the classes of strings we are interested in correcting--those strings in G corrupted by an α. β, or δ error.

III. SPECIFICATION OF PDT

Now that the grammars have been identified and a process specified for obtaining them from the original, we must specify the form of the PDT's that will correct strings in G_e. To develop a PDA, D, that will recognize a string in L(G) where G is the original CGF in GNF, let $D = (\{q_o\}, T, N, f, q_o, S)$ for $G = (T,N,S,P)$ with $(q_o,\gamma) \in f(q_o,a,A)$ if $A \to a\gamma \in P$. This is precisely the PDA that Hopcroft and Ullman (1968, Theorem 5.2) identify as the recognizer of L(G).

Formal Definitions

To develop the more powerful PDT, T^e, that will map strings in $L(G_e)$ to strings in L(G), let $T^e = (\{q_o\}, T_e, (N_e \cup E), (I_e \cup \{<\}), f_e, q_o, S_e)$ for $G_e = (T_e, N_e, S_e, P_e)$. The specification of the stack alphabet requires some additional explanation: if $e = \alpha$ or δ, then E is the null set; if $e = \beta$, then E contains some additional stack symbols that will be introduced later. The symbol $<$ in the output string shows where a correction has occurred. The choice of this correction notation is symbolic as it "points" to where the correction occurred.

The definition of f_e is as follows:

(1) $(q_o,\gamma,a) \in f_e(q_o,a,A_e)$ for each $A_e \to a\gamma \in P_e$, $\gamma \in N^* N_e N^*$;

(2) $(q_o,\gamma,a) \in f_e(q_o,a,A)$ for each $A \to a\gamma \in P_e$, $\gamma \in N^*$;

(3) For $e = \alpha$,

$(q_o,\gamma,a<) \in f_\alpha(q_o,\lambda,A_\alpha)$ for each pair of productions $A_\alpha \to \gamma$ and $A \to a\gamma \in P_\alpha$, $\gamma \in N^*$;

For $e = \beta$,

$(q_o, A, <) \in f_\beta(q_o, a, A_\beta)$ for each pair of productions $A_\beta \rightarrow ab\gamma$ and $A \rightarrow b\gamma \in P_\beta$, $a,b \in T$, $\gamma \in N^*$, and $(q_o, \underline{A}, a) \in f_\beta(q_o, a, A_\beta)$ and $(q_o, \gamma, <) \in f_\beta(q_o, b, \underline{A})$ for each pair of productions $A_\beta \rightarrow ab\gamma$ and $A \rightarrow a\gamma \in P_\beta$, $a,b \in T$, $\gamma \in N^*$, where a new pushdown list symbol $(\underline{A}, \underline{\underline{A}}, \ldots)$ is introduced for each $a \in T$;

For $e = \delta$,

$(q_o, \gamma, b>) \in f_\delta(q_o, a, A_\delta)$ for each pair of productions $A_\delta \rightarrow a\gamma$ and $A \rightarrow b\gamma \in P_\delta$, $\gamma \in N^*$.

Rules (1) and (2) follow directly from the f map for the PDA,D, defined for L(G). If the error occurs at level n, rule (1) applies at levels 1 through n-1 of the m-level derivation tree of $w \in L(G_e)$ and rule (2) applies at levels n+1 through m. Indeed, rule (2) simply states that $f \subseteq f_e$; that is, the f map for D is contained in the f map of T^e, with suitable extensions for an output in f, of course.

Rule (3) applies the actual correction techniques. When an α error occurs, a symbol is deleted; therefore, an error-correcting PDT, T^α, must insert a proper terminal in a correct place in the output string. It should be obvious that only a terminal that appears in a "corresponding" production in the correct grammar may be used.

When a β error occurs, an extra symbol is introduced and the error-correcting PDT, T^β, must delete a terminal to leave a correct string. Since there is no way of knowing whether

a terminal being scanned has an extra symbol following it at the time it is scanned, extra stack variables must be introduced to test this hypothesis. For each nonterminal in N, |T| new symbols may be introduced, where |T| is the number of terminals in G. Not necessarily all of these symbols will be used, but no more than (|N|)(|T|) new symbols will be added to L_β. An interesting point observable at this time is that b may equal a since any additional symbol is an error, even if it is identical to its neighbor. This will cause multiple members of the f map to be generated which are identical. This causes no trouble in the formal definition, but will waste space and time in an actual implementation.

The case of the f map for T^δ is the most straightforward since G_δ is still in GNF while G_α and G_β are not. In this case, the terminal in a production that introduces an error must be mapped into any terminal that may be associated with the production in G.

Consider as an example, the grammars mentioned earlier with the defining equations F = (EP + X; F_α = EP + (E_αP + (EP_α + λ; F_β = (((+)(+ *(+ X(+ X(+ /) + (/) EP + (E_βP + (EP_β + (X + X) +)X + X) + *X + X* + XX + /X + X/; and finally F_δ = ( ) + * + X + /)EP + (E_δP + (EP_δ + (+) + * + /.

The productions F→(EP and F_α→EP yield

$$(q_o, EP, (<) \in f_\alpha(q_o, \lambda, F_\alpha).$$

The productions $F \rightarrow (EP$ and $F_\beta \rightarrow (*EP$ yield

$(q_o, \underline{F}, () \in f_\beta(q_o, (, F_\beta)$ and

$(q_o, EP, <) \in f_\beta(q_o, *, \underline{F})$

while the productions $F \rightarrow (EP$ and $F_\beta \rightarrow *(EP$ yield

$(q_o, EP, <) \in f_\beta(q_o, *, F_\beta)$.

The productions $F \rightarrow (EP$ and $F_\delta \rightarrow *EP$ yield

$(q_o, EP, (<) \in f_\delta(q_o, *, F_\delta)$.

It remains to be shown that the f map defined for T^e will produce an output string in L(G) when the input string is in $L(G_e)$. Suppose w is in $L(G_e)$ and v in L(G) is a corresponding correct string.

In order for T^e to produce useful output, it must accept the input string w by an empty stack. This means two things immediately: 1) w is incorrect in one and only one position, and 2) T^e empties its stack of a subscripted variable. The second point is important when we consider how this must be done.

The initial stack variable is S_e, a subscripted variable. As long as a subscripted variable remains on the stack, the output symbol generated by T^e is identical to the input symbol scanned and the e error has not been detected. Entries from $f_e(q_o, a, A_e)$ of the form (q_o, γ_e, a) are used here. When the subscripted variable is removed from the stack an f map entry of the form $(q_o, \gamma, a<)$ from $f_e(q_o, b, A_e)$ is used, and the error correction occurs. At this point, the output symbol(s) is not equal to the input symbol (possibly λ) being scanned.

Once this transformation has taken place, however, T^e continues, with the output once more equal to the input. The f map entries are from $f_e(q_o,a,A)$.

Therefore, the output string v of T^e will differ from the input string w by only one character, and this difference will occur at precisely the point an error could have occurred. The output symbol at this point is defined to be an input symbol that would be scanned by a PDA for G operating on v given the same stack variable and resulting new stack symbol(s), with the addition of the error flag $<$. The output v differs from w by only two characters, one of them the flag $<$.

Data Structures and Extraction Process

The problem of the f map for D is the simplest--it involves no output and the definition itself is much simpler. Since D is a single state PDA, representing the f map narrows to the problem of finding a way to store the new stack symbol(s) for each possible combination of a terminal from the input string and a nonterminal from the stack. If a two dimensional array-type structure, where one dimension represents the terminals and the other nonterminals, were used, indices based upon the current input symbol and the current stack symbol could locate the new stack symbol(s). This structure alone is, however, insufficient since there may be more than one group of new stack symbol(s) in the f map for any given input symbol and stack symbol. Another dimension

must be added to account for this multiplicity, but the number of these multiple new stack symbols must be known so that indexing in them can be done. Since the number of new stack symbol(s) will be of numeric type and the new stack symbol(s) will themselves be character, the following PL/1 structure construction is chosen:

```
1  F_MAP(#_TERMS,#_NON_TERMS),
   5 #_FS  numeric,
   5 STACK(#)  character;
```

where #_TERMS contains the number of terminals and #_NON_TERMS contains the number of nonterminals for the grammar under consideration. The variable #_FS contains the current number of new stack symbol(s) for each level. The symbol(s) are actually stored in the elements of the character array STACK. The only unknown, or unspecified, parameter is #, which should be the maximum number of new stack symbol(s) for any level. This figure could not be determined by anything given in the grammar until the f map is computed. Therefore, a default value is used; if this value proves too small, the value is reassigned on subsequent runs.

With this data structure, the f map for the PDA D is easily computed as follows:

(0) Set #_FS to zeros.

(1) Locate a production of the form $A \rightarrow a\gamma$; set J to the index for A among the nonterminals, and set I to the index for a among the terminals.

(2) Increment #_FS(I,J) by one.

(3) If #_FS(I,J) is greater than #, then output an error message.

(4) Store γ in F_MAP(I,J).STACK(#_FS(I,J)).

(5) Go to (1) unless defining equations have been exhausted.

The data structures for the f maps of the PDT's are more complex. In fact, each one presents different problems to be resolved in different ways. In order of complexity, the f maps are arranged: δ, α, β.

A PDT, in general, has an output associated with each move while a PDA does not. This output, of course, represents an integral part of the f map of a PDT and must be stored. The output will either be a single character, in which case it is the same as the input, or else a corrected output symbol and $<$. This output is directly associated with the new stack symbol(s), and a new substructure is suggested:

```
5 DUMMY(#),
  10 STACK  character,
  10 OUTPUT  character;
```

where DUMMY is simply a "dummy" array that will associate STACK and OUTPUT.

A second observation is also in order. Since $G \subseteq G_e$, then $f \subseteq f_e$ if we add the proper output to f. This means that the f map for D, with an output associated with each step, is contained in the f map for T^e. Considerable computation can be saved if this f map (for D) is computed only once and then restored as needed. Since a derivation tree

for $w \in L(G_e)$ has a distinct point at which the error occurs and the productions without subscripts are used, a distinct division of the f map for T^e is suggested. This division will be based on whether the stack variable on the top of the stack is subscripted or not. The following structure can be used for the f map of T^e:

```
1 e_F_MAP(2,#_TERMS,#_NON_TERMS),
  5 #_FS  numeric,
  5 DUMMY (#),
    10 STACK  character,
    10 OUTPUT  character;
```

where the first subscript holds the key as to whether the stack variable on the top is subscripted. Precisely the above data structure is used to contain the f map for T^{δ}, computed by the following algorithm:

(0) Set #_FS to zeros.

(1) Locate a production of the form $A_{\delta} \rightarrow a\gamma$; set J to the index of A, and set I to the index of a.

(2) If γ contains a subscript then increment #_FS(2,I,J) by 1, store γ in D_F_MAP(2,I,J).STACK(#_FS(2,I,J)), and store a in D_F_MAP(2,I,J).OUTPUT(#_FS(2,I,J)); go to (4).

(3) Else, for each production $A \rightarrow b\gamma$ located in G, increment #_FS(I,J), store γ in D_F_MAP(I,J). STACK(#_FS(2,I,J)), and store b< in D_F_MAP(I,J). OUTPUT(#_FS(2,I,J)).

(4) Go to (1) unless defining equations of the form A_δ = . . . have been exhausted.

T^α is unusual when compared with T^δ or D: it can move without scanning an input symbol, that is, it is not a real-time machine. This occurs when a correction takes place, as a symbol is inserted to replace one which has been deleted. Therefore, some means of representing a null input character must be devised. This can be realized by adding an additional value to the subscript for the index of the input as follows:

```
1  A_F_MAP(2,0:#_TERMS,#_NON_TERMS).
```

This has the effect of allowing a 0 subscript, a convenient representation for a λ transformation. Notice that all corrections will be made with the second subscript equal to 0. The f map for T^α is obtained as follows:

(0) Set #_FS to zeros.

(1) Locate a production of the form $A_\alpha \rightarrow a\gamma$; set J to the index of A.

(2) If $a \neq \lambda$, then set I to the index of a, increment #_FS(2,I,J) by 1, store γ in A_F_MAP(2,I,J).STACK(#_FS(2,I,J)), and store a in A_F_MAP(2,I,J).OUTPUT(#_FS(2,I,J)); go to (4).

(3) If $a = \lambda$, then for each production $A \rightarrow b\gamma$ in G increment #_FS(2,0,J) by 1, store γ in A_F_MAP(2,0,J).STACK(#_FS(2,0,J)), and store b$\prec$ in A_F_MAP(2,0,J).OUTPUT(#_FS(2,0,J)).

(4) Go to (1) unless all defining equations A_α = . . . have been exhausted.

The PDT T^β has a different problem associated with it. Recall that new symbols are introduced into L_β. Some way of representing these new symbols, which are derivatives of the original set of nonterminals, must be found. The method chosen is to subscript the variables with different symbols, so that by examining whether the stack variable on the top of the stack has a subscript, and if so, which one, the set of productions to be used can be determined. The structure for this is as follows:

```
1 B_F_MAP(2+#_TERMS,#_TERMS,#_NON_TERMS).
```

Each additional symbol (there may be as many as #_TERMS of these per nonterminal) will be a different level of operation when the top stack variable is subscripted by the error operator, and when it is not subscripted at all. The algorithm for the f map of T^β is:

(0) Set #_FS to zeros.

(1) Locate a production of the form $A_\beta \rightarrow a\gamma$; set J to the index of A, and set I to the index of a.

(2) If γ contains a β subscript, then increment #_FS(2,I,J) by 1, store γ in B_F_MAP(2,I,J). STACK(#_FS(2,I,J)), and store a in B_F_MAP(2,I,J). STACK(#_FS(2,I,J)); go to (6).

(3) Else $\gamma = b\gamma_2$; set I_2 to the index of b,

(4) For each production $A \rightarrow a\gamma_2$ in G, increment #_FS(2,I,J) by 1, store $\underline{A}$ in B_F_MAP(2,I,J).

STACK(#_FS(2,I,J)), and store a in B_F_MAP(2,I,J). OUTPUT(#_FS(2,I,J)); increment #_FS(2+I,I_2,J) by 1, store γ_2 in B_F_MAP(2+I,I_2,J). STACK(#_FS(2+I,I_2,J)), and store $\prec$ in B_F_MAP(2+I,I_2,J).OUTPUT(#_FS(2+I,I_2,J)).

(5) For each production A→bγ_2 in G, increment #_FS(2,I,J0 by 1, store A in B_F_MAP(2,I,J). STACK(#_FS(2,I,J)), and store < in B_F_MAP(2,I,J). OUTPUT(#_FS(2,I,J)).

(6) Go to (1) unless all defining equations A_β = . . . have been exhausted.

Note: for later speed efficiency and space savings, a further test at steps (4) and (5) should be made to ensure that duplicate entries are not stored.

IV. SIMULATION OF PDT

Now that T^e, the PDT for correcting e errors in strings and outputting a string in L(G), has been specified by defining the f map, we are ready to simulate its operation.

Recall that a PDT is nondeterministic, that it generates left-most derivations in $L(G_e)$ until the input either is accepted via an empty stack or is rejected. During this process, whenever the current top symbol on the stack becomes inconsistent with the input string, backtracking must be instituted. Backtracking refers to the process by which the input and stack are "backed up" or restored so that an alternative entry in the f map may be used in the attempt to find a left-parse. Only after all possibilities have been tried can the string be rejected.

The simulation of a PDT is not an exercise in efficient parsing. As will be seen, in fact, the algorithm used is not efficient. Rather the simulation is simply the next step in the development of the formal model being presented. The basic algorithm used is Algorithm 4.1 from Aho and Ullman (1972). After brief summary of this algorithm, the remainder of this section will deal with modifications made to it for the purposes of the model being developed.

Top-down backtrack parsing.

Input: A non-left recursive CFG G = (N,T,S,P) and an input string $w = a_1 a_2 \ldots a_n$, $n>0$. The productions in P are numbered 1,2,...,p.

Output: One left parse for w, if one exists. Otherwise, an error message.

Method:

(1) For each nonterminal A in N, order the alternates for A. Let A_i be the index for the ith alternate in A.

(2) A four-tuple (s,i,α,β) is used to denote the configuration of the algorithm where

- s is the state of the algorithm.
- i is the location of the input pointer. The (n+1)st "input symbol" is \$, the right endmarker.
- α represents L1, a pushdown list.
- β represents L2, a second pushdown list.

The top of α is on the right, and the top of β is on the left. L2 represents the current sentential form. L1 represents the current history of the choices made and the input symbols scanned. The algorithm will be in one of three states: q, normal operation; b, backtracking; or t, termination.

(3) The initial configuration of the algorithm is $(q,1,\lambda,S\$)$.

(4) There are six steps, which will be discussed in terms of their effect on the configuration of the algorithm. The basic operation is "goes to" represented by $\vdash$ which will be used to compute successive configurations. The following hold: $1 \leq i \leq n+1$; $\alpha \in (T \cup I)^*$, where I is the set of indices for the alternates; and $\beta \in (N \cup T)^*$. The six moves are:

(a) Tree expansion

$$(q,i,\alpha,A\beta)\vdash(q,i,\alpha A_1,\gamma_1\beta)$$

where $A\rightarrow\gamma_1$ is the first alternate for A in P.

(b) Successful match of input symbols and derived symbol

$$(q,i,\alpha,a\beta)\vdash(q,i+1,\alpha a,\beta)$$

if $a_i=a$, $i\leq n$.

(c) Successful conclusion

$$(q,n+1,\alpha,\$)\vdash(t,n+1,\alpha,\lambda)$$

The input has been recognized. The left parse may be recovered from α by means of a homomorphism, h.

(d) Unsuccessful match of input symbol and derived symbol

$$(q,i,\alpha,a\beta)\vdash(b,i,\alpha,a\beta)$$

if $a_i\neq a$.

(e) Backtracking on input

$$(b,i,\alpha a,\beta)\vdash(b,i-1,\alpha,a\beta)$$

(f) Try next alternate

$$(b,i,A_j,\gamma_j\beta)\vdash$$

(i) $(q,i,A_{j+1},\gamma_{j+1}\beta)$, if γ_{j+1} is the j+1st alternate for A.

(ii) No configuration if i=1, A=S, and there only j alternates for S.

(iii) $(b,i,\alpha,A\beta)$ otherwise.

The algorithm executes as follows:

Step 1: Starting with the initial configuration, compute successive next configurations $C_0 \vdash C_1 \vdash \ldots \vdash C_i \vdash \ldots$ until no further configurations can be computed.

Step 2: If the last computed configuration is $(t,n+1,\alpha,\lambda)$, emit the left parse $h(\alpha)$ and halt. Else, emit the error signal.

The algorithm above is given in terms of the original grammar using the notation of productions rather than in terms of a PDT using the f map notation. The output produced is a list of the productions used on the left parse if one exists, and an error message otherwise (Aho and Ullman, 1972, Theorem 4.1). These points notwithstanding, it will be modifiable to our purposes here. It is proved by Aho and Ullman (1972) that the algorithm given above will operate in dn space in c^n time where n is the length of the input and d and c are suitable constants (Theorems 4.2 and 4.3).

As indicated, the above algorithm is not used directly but rather is modified to take advantage of the f map structure developed as well as certain simplifying assumptions that can be made because the grammar is in GNF which is a special case of non-left-recursion (Aho and Ullman, 1972, pp. 153-157).

The entries in the f map structures presented in the previous section are indexed based upon the current input symbol and the current stack variable on the top of the stack. If there are more entries for an index when

backtracking starts, then another alternate is tried; otherwise, an alternate for an earlier index is attempted.

The algorithm used is as follows:

Input: A GNF grammar G = (N,T,S,P) and an input string $w = a_1 a_2 \ldots a_n$, $n>0$.

Output: A string in L(G).

Method:

(1) Compute the f map for D or T^e, whichever the case may be. The entries will be ordered arbitrarily depending on the order of the defining equations.

(2) A four-tuple (s,i,α,β) denotes the configuration of the algorithm where

- s is state of the algorithm (q, b, or t);
- i is the input pointer; the (n+1)st input symbol is \$;
- α represents a pushdown list, L1, which essentially contains the output, $\alpha \in T^*$;
- β represents a pushdown list, L2, which contains the current sentential form, $\beta \in N^*$.

(3) The initial configuration is $(q,1,\lambda,S\$)$.

(4) There are the following four types of steps:

(a) Tree expansion by successful match of input symbol and derived symbol

$$(q,i,\alpha,A\beta) \vdash (q,i+1,\alpha b_1,\gamma_1\beta)$$

if $f(q,a_i,A)$ contains (q,γ_1,b_1) as its first entry.

(b) Successful conclusion

$(q,n+1,\alpha,\$) \vdash (t,n+1,\alpha,\lambda)$

the output of the algorithm is $\alpha \in L(G)$.

(c) Unsuccessful match of input symbol

$(q,i,\alpha,A\beta) \vdash (b,i,\alpha A\beta)$

if $f(q,a_i,A)$ contains no entries.

(d) Try next alternate

$(b,i,\alpha b_j,\gamma_j\beta) \vdash$

(i) $(q,i,\alpha b_{j+1},\gamma_{j+1}\beta)$ if $f(q,a_i,A)$ contains (q,γ_{j+1},b_{j+1}) as its j+1st entry.

(ii) Unsuccessful termination if i=1, A=S, and there are only j entries in S.

(iii) $(b,i-1,\alpha,A\beta)$ otherwise.

The execution is as follows:

Step 1: Compute successive configurations $C_0 \vdash C_1 \vdash \ldots \vdash C_i \vdash \ldots$ until no further configurations can be computed.

Step 2: If the last state of the algorithm is $(t,n+1,\alpha,\lambda)$, then output α; otherwise, output an error message.

A close examination of the two algorithms reveals that they are virtually identical and will produce the same configurations when given the same GNF grammar.

The actual implementation of L1 is more complicated than is indicated, however. The items that must be kept for successful backtracking include the top symbol on L2, the current input symbol a_i, the length of the new top of L2 γ_k, the output symbol, and which entry in the f map is currently

being used. But this does not change the basic operation of the algorithm; L1 is simply represented as a PL/I structure.

There are two primary items which prevent the above algorithm from being equally applicable to D and T^e. First, the f map of D does not include any output. Second, the structure of the f map for T^e requires an indication as to whether the current variable on top of L2 is subscripted or not. If the variable is subscripted, then the productions used are those generated by application of the error operators to the original grammar; otherwise, the original productions are used. Furthermore, in the case of T^β, L2 may contain variables subscripted but not with the error subscript, so which subscript is present must be determined in T^β. The determination of which level of the f map (the first subscript) to use is made at move (a) of the algorithm by checking the subscript, if it exists. This is the only change required to adapt the algorithm for T^δ and T^β, but it does necessitate an additional structure in L1--an indication of which level of f map was used to ensure correct backtracking.

T^α presents additional considerations. In addition to the modifications detailed above, a means of simulating the λ transitions in the f map had to be developed. It was decided arbitrarily that a λ transition would be attempted initially whenever the variable on top of L2 was subscripted. This means that a_i, the input symbol being scanned, is ignored and an entry from $f(q,\lambda,A_\alpha)$ is used to insert a character in the

output. On backtracking, at move (d).(iii), if a λ transition has been made, then the configuration of the algorithm is returned as it was before the input symbol a_i was ignored and an entry from $f(q,a_i,A_\alpha)$ is used.

A separate subroutine for D, T^α, T^β, and T^δ was written to apply the algorithm utilizing the proper f map. The subroutines are called serially in the following order: D, T^δ, T^β, and T^α. Whenever any one subroutine accepts the input string, then the remainder are ignored, since a correct string has been found. The sequence of calling was chosen as the order from least complicated to most. It takes many more moves for T^α to reject a string than T^β, and more for T^β to reject a string than T^δ. Naturally, if a string is correct, there is no need to try to correct it.

Some actual results of the simulation programmed for the example grammar used previously are given in Table 1 for T^α; Table 2 contains examples for T^β; and Table 3 for T^δ. The program itself is listed in the Appendix.

TABLE 1

EXAMPLES OF CORRECTIONS DONE BY ALPHA PDT

INCORRECT STRING	CORRECTED STRING
()	(X<)
(X*())	(X*(X<))
(X)(X)	(X)*<(X)
X*()*X	X*(X<)*X
()*X/X	(X<)*X/X
(X*X)/()	(X*X)/(X<)
(X*X)(X)	(X*X)*<(X)

TABLE 2

EXAMPLES OF CORRECTIONS DONE BY BETA PDT

INCORRECT STRING	CORRECTED STRING
(XX)	(<X)
(X*(X)	<X*(X)
(X)/XX/X	(X)/<X/X
X*(XX)*X	X*(<X)*X
()X)*X/X	(<X)*X/X
X*(X//X)	X*(X</X)
(((X*X)/X)/(X)	(<(X*X)/X)/(X)

TABLE 3

EXAMPLES OF CORRECTIONS DONE BY DELTA PDT

INCORRECT STRING	CORRECTED STRING
X*(()	X*(X<)
X*(X//)	X*(X/X<)
(X)X(X)	(X)*<(X)
X*(/)*X	X*(X<)*X
X/X(X*X	X/X/<X*X
((X*X)/()	((X*X)/X<)
((*X)/X*(X)	(X<*X)/X*(X)

V. CONCLUSION

An algorithm that will correct certain classes of errors in context-free languages has been developed and implemented. Given a context-free grammar in Greibach Normal Form, the algorithm first applies transformations to the productions of the grammar to compute three new grammars. These grammars will produce all strings derivable from the original language by a single occurrence of an α error (the deletion of a symbol), a β error (the insertion of a symbol), or a δ error (the changing of a symbol into another symbol). Pushdown transducers capable of mapping these strings back into the original language are then defined by specifying the f map of each. Once the f map for each PDT is calculated and stored, the operation of each PDT is simulated by a separate subroutine.

Further analysis of the present implementation would probably lead to an attempt to use a list structure for the storage of the f map as opposed to the array-structure approach chosen. This would cut down on the storage locations used, since much of the current space allocated goes unused. The bookkeeping function added would probably be little more than is now incurred by the use of multidimensional arrays.

Another possible improvement would be the incorporation of probabilities into the model. If the relative frequency of occurrence of the productions and error types were determined, then the productions and corresponding f map entries

could be ordered to reflect these probabilities. The execution times of the algorithm would, on the average, go down. Even more powerful would be some means of dynamic ordering of f map entries, if the probability of usage changes over the length of the string.

Multiple errors can also be corrected using this algorithm by successively applying the error transforms to obtain still more grammars. This area should be developed further to the point that multiple errors can be corrected--at least within the same formal framework that this implementation has operated.

These suggestions still fall short of the ultimate goal--practical error correction.

BIBLIOGRAPHY

BIBLIOGRAPHY

Aho, A. V., and Ullman, J. D. The Theory of Parsing, Translation, and Compiling, Vol. 1. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1972.

Conway, R. W., and Maxwell, W. L. "CORC--The Cornell Computing Language." Communications of the ACM 6, 9 (Sept. 1966), 317-321.

Damerau, F. "A Technique for Computer Detection and Correction of Spelling Errors." Communications of the ACM 7, 2 (Feb. 1964), 171-176.

Hopcroft, J. E., and Ullman, J. D. Formal Languages and Their Relation to Automata. Addison-Wesley, Reading, Mass., 1969.

Leinius, R. P. Error Detection and Recovery for Syntax Directed Compiler Systems. Ph.D. Thesis, U. of Wisconsin, Madison, Wis., 1970.

Lyon, G. E. "Least-Errors Recognition of Mutated Context-Free Sentences in Time $n^3 \log n$." Proc. Sixth Princeton Conf. on Inform. Syst. and Sci., Princeton, N. J., 1972, pp. 115-118.

Lyon, G. E. "Syntax-Directed Least-Errors Analysis for Context-Free Language: A Practical Approach." Communications of the ACM 17, 1 (Jan. 1974), 3-14.

Morgan, H. L. "Spelling Correction in Systems Programs," Communications of the ACM 13, 2 (Feb. 1970), 90-94.

Morgan, H. L., and Wagner, R. A. "PL/C--A High Performance Compiler for PL/I." Proc. 1971 SJCC, Vol. 38, AFIPS Press, Montvale, N. J., pp. 503-510.

Thomason, M. G. "Error Control in Context-Free Languages." ACM 12th Annual Southeast Region Conference, (April 18-20, 1974), pp. 320-330.

Wagner, R. A. "An n^3 Minimum Edit Distance Correction Algorithm for Context-Free Languages." Tech. Rep., Systems and Information Science Dep., Vanderbilt U., Nashville, Tenn., 1972.

Wagner, R. A. "Order-n Correction for Regular Languages." Communications of the ACM 17, 5 (May, 1974), 265-268.

APPENDIX

PROGRAM LISTING

```
ERROR_CORRECTOR: PROC OPTIONS(MAIN) REORDER;
DCL (OUT,I,J,L,II,JJ,#_TERMS,#_NON_TERMS,PTR_L1,ALTERN,INDEX_L2,
      INDEX_IN,IND1,IND2,IND3,CURRENT,CONTROL,CONTROL2,CONTROL3,K,N)
        FIXED BIN(16);
DCL (M,#F,#A,#B,#D,#_TESTS) FIXED BIN(16);
DCL (SUB,DUM) CHAR(10) VAR,
    OLDTERM CHAR(1),
    L2 CHAR(500) VAR,
    STRING CHAR(500),
    CHARS(80) CHAR(1);
DCL (START,START2) CHAR(3) VAR;
DCL (EOS,FIND_PARSE) BIT(1);
DCL TEST(5) CHAR(50) VAR;
DCL SPECIAL CHAR(19) INIT('#@%*.-/& )_+>:;¬( ?');
DCL S1 CHAR(21) INIT('THE INCORRECT STRING'),
    S2 CHAR(27) INIT(' HAS BEEN CORRECTED BY THE'),
    S3(3) CHAR(5) VAR INIT('ALPHA','BETA','DELTA'),
    S4 CHAR(13) INIT(' MACHINE TO:');
DCL IY FIXED BIN(31);
DCL I1 FIXED BIN(31) STATIC INIT(65539),
    I2 FIXED BIN(31) STATIC INIT(214743647),
    I3 FIXED BIN(31) STATIC INIT(1);
  (NOFIXEDOVERFLOW): IY=ABS(IY*2)+1;
ON ENDFILE(SYSIN) GO TO END_ERROR_CORRECTOR;
  OPEN FILE(SYSPRINT) PRINT LINESIZE(132);
DO CONTROL=1 TO 100;
#F,#A,#B,#D=5;
  GET DATA(#F,#A,#B,#D);
  GET LIST(#_TERMS,#_NON_TERMS);
BEGIN_BLOCK:  BEGIN;
DCL (INPUT,OUTPUT) CHAR(50) VAR;
DCL OLD(#_NON_TERMS) CHAR(80) VAR;
```

```
/*                              INPUT INSTRUCTIONS
   INPUT IS A GRAMMAR AT A TIME; MULTIPLE GRAMMARS MAY BE SUBMITTED
TOGETHER. FIRST THE NUMBER OF F MAP ENTRIES FOR EACH F MAP IS ENTERED;
THESE ARE NAMED #F, #A, #B, AND #D. THERE IS A DEFAULT VALUE OF 5 FOR
EACH. NOTHING NEED BE CODED UNLESS AN ERROR MESSAGE FOR THE GRAMMAR IN
QUESTION WAS PREVIOUSLY GENERATED. THE FORM IS PL/I DATA DIRECTED; THE
INPUT MUST BE FOLLOWED BY A SEMICOLON(;) EVEN IF NO VALUES ARE INPUT.
AS MANY VARIABLES AS NEED TO BE CHANGED CAN BE INCLUDED, EACH SEPARATED
BY A COMMA. SECOND IS ENTERED THE NUMBER OF TERMINALS AND THE NUMBER
OF NONTERMINALS, SEPARATED BY BLANKS, OR A COMMA (PL/I LIST DIRECTED
FORMAT). THIRD, ALSO IN LIST DIRECTED FORMAT, THE TERMINAL, NONTERMINAL
AND START CHARACTERS ARE ENTERED. THESE SYMBOLS ARE ENCLOSED IN SINGLE
QUOTES, SEPARATED BY BLANKS OR COMMAS. THEN COME THE PRODUCTIONS IN
DEFINING EQUATION FORM. THERE IS ONE CARD FOR EACH NONTERMINAL; EACH
EQUATION BEGINS ON A NEW CARD AND IS TERMINATED BY A SEMICOLON(;). THE
SPACING ON THE CARDS WITH THE DEFINING EQUATIONS IS UNIMPORTANT; BLANKS
WILL BE IGNORED. FINALLY THE TEST STRINGS THAT WILL BE PASSED TO THE
SIMULATIONS ARE INPUT. FIRST THE NUMBER OF SUCH STRINGS IS CODED AS
"#_TESTS=#;" (PL/I DATA DIRECTED FORMAT - NO QUOTES). FOLLOW THIS BY
THAT MANY STRINGS (#), EACH ENCLOSED IN SINGLE QUOTES. A NEW GRAMMAR
MAY FOLLOW. AN EXAMPLE WITH 2 GRAMMARS IS GIVEN BELOW:
//GO.SYSIN DD *
  ;  2,3  '0','1'   'S','A','B'   'S'
 S=0A+1B+0SA+1SB;
 A=0;
  B=1;
#_TESTS=3;  '010100010'  '0110110' '010'
#A=12,#D=16,#B=14;
5,6  '(',')','*','X','/'   'E','G','T','Q','F','P'   'E'
E=(EP+X+(EPQ+XQ+(EPG+XG+(EPQG+XQG;
G=/T+/TG;
T=(EP+X+(EPQ+XQ;
Q=*F+*FQ;
F=(EP+X;
P=);
#_TESTS=4;  'X*( )*X'  'X*(/)*X'  'X*(XX)*X'  '(X*X)/(X)'              */
```

```
DCL NEW(3,#_NON_TERMS) CHAR(500) VAR;
DCL TERM CHAR(#_TERMS) VAR,  TERMS(#_TERMS) CHAR(1);
DCL NON_TERM CHAR(#_NON_TERMS) VAR, NON_TERMS(#_NON_TERMS) CHAR(1);
DCL FANCY CHAR(#_TERMS+1) VAR INIT(',');
DCL 1 F_MAP(#_TERMS,#_NON_TERMS),
      5 #_FS FIXED BIN(16),
      5 DUMMY(#F),
        10 STACK CHAR( 5) VAR,
        10 OUTPUT CHAR(1) VAR;
DCL 1 A_F_MAP(2,0:#_TERMS,#_NON_TERMS),
      5 #_FS FIXED BIN(16),
      5 DUMMY(#A),
        10 STACK CHAR( 5) VAR,
        10 OUTPUT CHAR(2) VAR;
DCL 1 D_F_MAP(2,#_TERMS,#_NON_TERMS),
      5 #_FS FIXED BIN(16),
      5 DUMMY(#D),
        10 STACK CHAR( 5) VAR,
        10 OUTPUT CHAR(2) VAR;
DCL 1 B_F_MAP(2+#_TERMS,#_TERMS,#_NON_TERMS),
      5 #_FS FIXED BIN(16),
      5 DUMMY(#B),
        10 STACK CHAR( 5) VAR,
        10 OUTPUT CHAR(1) VAR;
DCL (FMAP,AMAP,BMAP,DMAP) BIT(1) INIT('0'B);

GENERATE:  PROCEDURE;
DCL (STR,SUB1,SUB2,OUT) CHAR(32) VAR,  SUBO CHAR(1);
RAND:  PROCEDURE(MAX,RANDUM);
DCL (MAX,RANDUM) FIXED BIN(16);
  (NOFIXEDOVERFLOW): IY=IY*I1;
  IF IY <  0 THEN IY=IY+I2+I3;
  UFL=IY;
  UFL=UFL*.4656613E-9;
  RANDUM=UFL*MAX+1E0;
END RAND;
```

```
   OUT=TEST(1);
   N=LENGTH(OUT);
   CALL RAND(N,L);
   SUB0=SUBSTR(OUT,L,1);
   IF L=1 THEN SUB1='';
          ELSE SUB1=SUBSTR(OUT,1,L-1);
   IF L=N THEN SUB2='';
          ELSE SUB2=SUBSTR(OUT,L+1);
   J,K=INDEX(TERM,SUB0);
   DO WHILE(J=K);     CALL RAND(#_TERMS,J);     END;
   TEST(2)=SUB1||SUB2;
   TEST(3)=SUB1||TERMS(J)||SUB0||SUB2;
   TEST(4)=SUB1||TERMS(J)||SUB2;
END GENERATE;

NEW_GRAMMARS:  PROCEDURE;
DO OUT=1 TO #_NON_TERMS;
 STRING=SUBSTR(OLD(OUT),3);
 EOS='0'B;   L=0;
 DO WHILE(¬EOS);
  STRING=SUBSTR(STRING,L+1);
  OLDTERM=SUBSTR(STRING,1,1);
  J=INDEX(TERM,OLDTERM);
  L=INDEX(STRING,'+');
  IF L=0 THEN DO;   EOS='1'B;    L=INDEX(STRING,';');    END;
  I=L-2;
  IF I=0 THEN SUB='';
         ELSE SUB=SUBSTR(STRING,2,I);
  NEW(1,OUT)=NEW(1,OUT)||'+'||SUB;
  DO JJ=1 TO #_TERMS;
      IF JJ¬=J THEN NEW(3,OUT)=NEW(3,OUT)||'+'||TERMS(JJ)||SUB;
      IF TERMS(JJ)=OLDTERM THEN NEW(2,OUT)=NEW(2,OUT)||'+'||TERMS(JJ)||
                                           OLDTERM||SUB;
                           ELSE NEW(2,OUT)=NEW(2,OUT)||'+'||TERMS(JJ)||
                                           OLDTERM||SUB||'+'||OLDTERM||
                                           TERMS(JJ)||SUB;
```

```
    END;
  DO II=1 TO I;
      NEW(*,OUT)=NEW(*,OUT)||'+'||OLDTERM;
      IF II¬=1 THEN NEW(*,OUT)=NEW(*,OUT)||SUBSTR(SUB,1,II-1);
      NEW(*,OUT)=NEW(*,OUT)||SUBSTR(SUB,II,1)||',';
      IF II¬=I THEN NEW(*,OUT)=NEW(*,OUT)||SUBSTR(SUB,II+1);
    END;
  END;
 NEW(*,OUT)=NEW(*,OUT)||';';
 NEW(*,OUT)=SUBSTR(OLD(OUT),1,1)||',='||SUBSTR(NEW(*,OUT),2);
END NEW_GRAMMARS;

GET_F_MAP:  PROCEDURE;
DCL K(#_TERMS) FIXED BIN(16);
DO L=1 TO #_NON_TERMS;
  I=INDEX(NON_TERM,SUBSTR(OLD(L),1,1));
  STRING=SUBSTR(OLD(I),INDEX(OLD(I),'=')+1);
  EOS='0'B;      K=0;      J=0;
  DO WHILE(¬EOS);
      STRING=SUBSTR(STRING,J+1);
      J=INDEX(STRING,'+');
      IF J=0 THEN DO;   EOS='1'B;   J=INDEX(STRING,';');   END;
      DUM=SUBSTR(STRING,1,J-1);
      OLDTERM=SUBSTR(DUM,1,1);
      IF LENGTH(DUM)>1 THEN SUB=SUBSTR(DUM,2);
                       ELSE SUB='';
      OUT=INDEX(TERM,OLDTERM);
      K(OUT)=K(OUT)+1;
      F_MAP(OUT,I,K(OUT)).OUTPUT=OLDTERM;
      F_MAP(OUT,I,K(OUT)).STACK=SUB;
    END;
  F_MAP.#_FS(*,I)=K;
DO M=1 TO #_TERMS;  IF K(M)>#F THEN DO; FMAP='1'B;  #F=K(M); END;  END;
END GET_F_MAP;

GET_ALPHA_F_MAP:  PROCEDURE;
```

```
DCL KK(0:#_TERMS) FIXED BIN(16);
DO L=1 TO #_NON_TERMS;
  A_F_MAP.#_FS(1,0,L)=0;
  I=INDEX(NON_TERM,SUBSTR(NEW(1,L),1,1));
  STRING=SUBSTR(NEW(1,I),INDEX(NEW(1,I),'=')+1);
  EOS='0'B;    KK=0;    J=0;
  DO WHILE(¬EOS);
      STRING=SUBSTR(STRING,J+1);
      J=INDEX(STRING,'+');
      IF J=0 THEN DO;  EOS='1'B;   J=INDEX(STRING,';');   END;
      IF J=1 THEN DO;  DUM='';   GO TO NULL;   END;
      DUM=SUBSTR(STRING,1,J-1);
      OLDTERM=SUBSTR(DUM,1,1);
      OUT=INDEX(TERM,OLDTERM);
      IF OUT=0 THEN DO;
          NULL: DO II=1 TO #_TERMS;
              DO JJ=1 TO F_MAP.#_FS(II,I);
                  IF DUM=F_MAP(II,I,JJ).STACK THEN GO TO FOUND;
                END;
              GO TO DID_NOT_FIND;
              FOUND: KK(0)=KK(0)+1;
              A_F_MAP(2,0,I,KK(0)).STACK=DUM;
              A_F_MAP(2,0,I,KK(0)).OUTPUT=F_MAP(II,I,JJ).OUTPUT||'<';
            DID_NOT_FIND:  END;
        END;
                ELSE DO;
          KK(OUT)=KK(OUT)+1;
          A_F_MAP(2,OUT,I,KK(OUT)).STACK=SUBSTR(DUM,2);
          A_F_MAP(2,OUT,I,KK(OUT)).OUTPUT=OLDTERM;
        END;
    END;
  A_F_MAP.#_FS(2,*,I)=KK;
DO M=0 TO #_TERMS; IF KK(M)>#A THEN DO; AMAP='1'B; #A=KK(M); END; END;
END GET_ALPHA_F_MAP;

GET_BETA_F_MAP:  PROCEDURE;
```

```
DCL K(2+#_TERMS,#_TERMS) FIXED BIN(16),
    SUB2 CHAR(10) VAR,   OLDTERM2 CHAR(1),   (OUT2,OUT3) FIXED BIN(16);
DCL LOCATE ENTRY(FIXED BIN(16), FIXED BIN(16), FIXED BIN(16),
      CHAR(*), CHAR(*) VAR);
LOCATE:  PROCEDURE(J,K,L,OUT,STK);
DCL (J,K,L) FIXED BIN(16),  OUT CHAR(*),  STK CHAR(*) VAR;
  DO JJ=1 TO L;
      IF B_F_MAP(J,K,I,JJ).OUTPUT=OUT THEN
        IF B_F_MAP(J,K,I,JJ).STACK=STK THEN RETURN;
    END;
  L=JJ;
  IF L>#B THEN DO;  BMAP='1'B;  #B=L;  END;
  B_F_MAP(J,K,I,L).OUTPUT=OUT;
  B_F_MAP(J,K,I,L).STACK=STK;
END LOCATE;
  DO L=1 TO #_NON_TERMS;
      I=INDEX(NON_TERM,SUBSTR(NEW(2,L),1,1));
      STRING=SUBSTR(NEW(2,I),INDEX(NEW(2,I),'=')+1);
      EOS='0'B;      K=0;      J=0;
      DO WHILE(¬EOS);
          STRING=SUBSTR(STRING,J+1);
          J=INDEX(STRING,'+');
          IF J=0 THEN DO;  EOS='1'B;   J=INDEX(STRING,';');   END;
          DUM=SUBSTR(STRING,1,J-1);
          OLDTERM=SUBSTR(DUM,1,1);
          OUT=INDEX(TERM,OLDTERM);
          IF INDEX(DUM,',')¬=0 THEN DO;
              K(2,OUT)=K(2,OUT)+1;
              B_F_MAP(2,OUT,I,K(2,OUT)).OUTPUT=OLDTERM;
              B_F_MAP(2,OUT,I,K(2,OUT)).STACK=SUBSTR(DUM,2);
            END;
            ELSE DO;
              OLDTERM2=SUBSTR(DUM,2,1);
              OUT2=INDEX(TERM,OLDTERM2);
              IF LENGTH(DUM)>2 THEN SUB=SUBSTR(DUM,3);
                               ELSE SUB='';
```

```
                IF INDEX(OLD(I),OLDTERM||SUB||'+')=0 &
                     INDEX(OLD(I),OLDTERM||SUB||';')=0 THEN;
                  ELSE DO;
                    SUB2=NON_TERMS(I)||SUBSTR(FANCY,OUT+1,1);
                    CALL LOCATE(2,OUT,K(2,OUT),OLDTERM,SUB2);
                    OUT3=OUT+2;
                    CALL LOCATE(OUT3,OUT2,K(OUT3,OUT2),'<',SUB);
                  END;
                IF INDEX(OLD(I),OLDTERM2||SUB||'+')=0 &
                     INDEX(OLD(I),OLDTERM2||SUB||';')=0 THEN;
                  ELSE CALL LOCATE(2,OUT,K(2,OUT),'<',NON_TERMS(I));
                END;
        END;
      B_F_MAP.#_FS(*,*,I)=K;
END GET_BETA_F_MAP;

GET_DELTA_F_MAP:  PROCEDURE;
DCL K(#_TERMS) FIXED BIN(16);
DO  L=1 TO #_NON_TERMS;
  I=INDEX(NON_TERM,SUBSTR(NEW(3,L),1,1));
  STRING=SUBSTR(NEW(3,I),INDEX(NEW(3,I),'=')+1);
  EOS='0'B;     K=0;     J=0;
  DO WHILE(¬EOS);
      STRING=SUBSTR(STRING,J+1);
      J=INDEX(STRING,'+');
      IF J=0 THEN DO;  EOS='1'B;   J=INDEX(STRING,';');   END;
      DUM=SUBSTR(STRING,1,J-1);
      OLDTERM=SUBSTR(DUM,1,1);
      IF LENGTH(DUM)>1 THEN SUB=SUBSTR(DUM,2);
                       ELSE SUB='';
      OUT=INDEX(TERM,OLDTERM);
      IF INDEX(SUB,',')¬=0 THEN DO;
          K(OUT)=K(OUT)+1;
          D_F_MAP(2,OUT,I,K(OUT)).STACK=SUB;
          D_F_MAP(2,OUT,I,K(OUT)).OUTPUT=OLDTERM;
        END;
```

```
                              ELSE DO;
            DO II=1 TO #_TERMS;
                DO JJ=1 TO F_MAP.#_FS(II,I);
                    IF SUB=F_MAP(II,I,JJ).STACK THEN GO TO FOUND;
                  END;
                GO TO DID_NOT_FIND;
                FOUND: K(OUT)=K(OUT)+1;
                D_F_MAP(2,OUT,I,K(OUT)).STACK=SUB;
               D_F_MAP(2,OUT,I,K(OUT)).OUTPUT=F_MAP(II,I,JJ).OUTPUT||'<';
              DID_NOT_FIND:  END;
          END;
      END;
    D_F_MAP.#_FS(2,*,I)=K;
DO M=1 TO #_TERMS; IF K(M)>#D THEN DO; DMAP='1'B; #D=K(M); END; END;
END GET_DELTA_F_MAP;

MACHINE:  PROCEDURE(INPUT,FOUND);
DCL (INPUT     ) CHAR(*) VAR, FOUND BIT(*);
DCL 1 L1(N),
       5 STACK FIXED BIN(16),
       5 WHICH FIXED BIN(16),
       5 LEN FIXED BIN(16),
       5 INPUT FIXED BIN(16);
  I=1;  L2=START2;     PTR_L1=0;
NEXT_SYMBOL:
  IF I = N THEN IF L2='$' THEN DO;  FOUND='1'B;  RETURN;  END;
  INDEX_IN=INDEX(TERM,SUBSTR(INPUT,I,1));
  IF INDEX_IN=0 THEN GO TO BACKTRACK;
  INDEX_L2=INDEX(NON_TERM,SUBSTR(L2,1,1));
  IF INDEX_L2=0 THEN GO TO BACKTRACK;
  IF F_MAP.#_FS(INDEX_IN,INDEX_L2)=0 THEN GO TO BACKTRACK;
  PTR_L1=PTR_L1+1;
  L1.STACK(PTR_L1)=INDEX_L2;
  L1.INPUT(PTR_L1)=INDEX_IN;
  L1.WHICH(PTR_L1)=1;
  L1.LEN(PTR_L1)=LENGTH(F_MAP.STACK(INDEX_IN,INDEX_L2,1));
```

```
  L2=F_MAP.STACK(INDEX_IN,INDEX_L2,1)||SUBSTR(L2,2);
  I=I+1;
  GO TO NEXT_SYMBOL;
BACKTRACK:
  IF I=1 THEN IF L2=START2 THEN DO;  FOUND='0'B;  RETURN;  END;
  IND1=L1.STACK(PTR_L1);
  IND2=L1.INPUT(PTR_L1);
  IF L1.WHICH(PTR_L1) < F_MAP.#_FS(IND2,IND1) THEN DO;
      ALTERN,L1.WHICH(PTR_L1)=L1.WHICH(PTR_L1)+1;
      L2=F_MAP.STACK(IND2,IND1,ALTERN)||SUBSTR(L2,L1.LEN(PTR_L1)+1);
      L1.LEN(PTR_L1)=LENGTH(F_MAP.STACK(IND2,IND1,ALTERN));
      GO TO NEXT_SYMBOL;
    END;
  L2=NON_TERMS(L1.STACK(PTR_L1))||SUBSTR(L2,L1.LEN(PTR_L1)+1);
  PTR_L1=PTR_L1-1;
  I=I-1;
  GO TO BACKTRACK;
END MACHINE;

ALPHA_MACHINE:  PROCEDURE(INPUT,FOUND,OUT);
DCL (INPUT,OUT) CHAR(*) VAR, FOUND BIT(*);
DCL 1 L1(N),
      5 INPUT FIXED BIN(16),
      5 OUTPUT CHAR(2) VAR,
      5 STACK CHAR(2) VAR,
      5 MACHINE FIXED BIN(16),
      5 WHICH FIXED BIN(16),
      5 LEN FIXED BIN(16);
  I=1;     L2=START;     PTR_L1=0;
NEXT_SYMBOL:
  IF I = N THEN IF L2='$' THEN DO;
      OUT='';  DO I=1 TO PTR_L1;  OUT=OUT||L1.OUTPUT(I);  END;
      FOUND='1'B;     RETURN;     END;
  INDEX_IN=INDEX(TERM,SUBSTR(INPUT,I,1));
  INDEX_L2=INDEX(NON_TERM,SUBSTR(L2,1,1));
  IF INDEX_L2=0 THEN GO TO BACKTRACK;
```

```
  IF SUBSTR(L2,2,1)=',' THEN DO;
      CURRENT=2;
      INDEX_IN=0;
    END;
                          ELSE IF INDEX_IN=0 THEN GO TO BACKTRACK;
                                             ELSE CURRENT=1;
TEST_STORE:
  IF A_F_MAP.#_FS(CURRENT,INDEX_IN,INDEX_L2)=0 THEN GO TO BACKTRACK;
  PTR_L1=PTR_L1+1;
  L1.STACK(PTR_L1)=SUBSTR(L2,1,CURRENT);
  L1.OUTPUT(PTR_L1)=A_F_MAP.OUTPUT(CURRENT,INDEX_IN,INDEX_L2,1);
  L1.INPUT(PTR_L1)=INDEX_IN;
  L1.WHICH(PTR_L1)=1;
  L1.LEN(PTR_L1)=LENGTH(A_F_MAP.STACK(CURRENT,INDEX_IN,INDEX_L2,1));
  L1.MACHINE(PTR_L1)=CURRENT;
  L2=A_F_MAP.STACK(CURRENT,INDEX_IN,INDEX_L2,1)||SUBSTR(L2,CURRENT+1);
  IF INDEX_IN¬=0 THEN I=I+1;
  GO TO NEXT_SYMBOL;
BACKTRACK:
  IF I=1 THEN IF L2=START THEN DO;  FOUND='0'B;  RETURN;  END;
  IND1=INDEX(NON_TERM,SUBSTR(L1.STACK(PTR_L1),1,1));
  IND2=L1.INPUT(PTR_L1);
  IND3=L1.MACHINE(PTR_L1);
  IF L1.WHICH(PTR_L1)<A_F_MAP.#_FS(IND3,IND2,IND1) THEN DO;
      ALTERN,L1.WHICH(PTR_L1)=L1.WHICH(PTR_L1)+1;
      L2=A_F_MAP.STACK(IND3,IND2,IND1,ALTERN)||
                        SUBSTR(L2,L1.LEN(PTR_L1)+1);
      L1.OUTPUT(PTR_L1)=A_F_MAP.OUTPUT(IND3,IND2,IND1,ALTERN);
      L1.LEN(PTR_L1)=LENGTH(A_F_MAP.STACK(IND3,IND2,IND1,ALTERN));
      GO TO NEXT_SYMBOL;
    END;
  L2=L1.STACK(PTR_L1)||SUBSTR(L2,L1.LEN(PTR_L1)+1);
  PTR_L1=PTR_L1-1;
  IF IND2=0 THEN DO;
      INDEX_IN=INDEX(TERM,SUBSTR(INPUT,I,1));
      IF INDEX_IN=0 THEN GO TO BACKTRACK;
```

```
      INDEX_L2=IND1;
      CURRENT=IND3;  /*  ALWAYS 2  */
      GO TO TEST_STORE;
    END;
  I=I-1;
  GO TO BACKTRACK;
END ALPHA_MACHINE;

BETA_MACHINE:  PROCEDURE(INPUT,FOUND,OUT);
DCL (INPUT,OUT) CHAR(*) VAR, FOUND BIT(*);
DCL 1 L1(N),
      5 INPUT FIXED BIN(16),
      5 OUTPUT CHAR(2) VAR,
      5 STACK CHAR(2) VAR,
      5 MACHINE FIXED BIN(16),
      5 WHICH FIXED BIN(16),
      5 LEN FIXED BIN(16);
DCL CUR_LEN FIXED BIN(16);
  I=1;  L2=START;     PTR_L1=0;
NEXT_SYMBOL:
  IF I = N THEN IF L2='$' THEN DO;
      OUT='';  DO I=1 TO PTR_L1;  OUT=OUT||L1.OUTPUT(I);  END;
      FOUND='1'B;     RETURN;     END;
  INDEX_IN=INDEX(TERM,SUBSTR(INPUT,I,1));
  IF INDEX_IN=0 THEN GO TO BACKTRACK;
  INDEX_L2=INDEX(NON_TERM,SUBSTR(L2,1,1));
  IF INDEX_L2=0 THEN GO TO BACKTRACK;
  CURRENT=INDEX(FANCY,SUBSTR(L2,2,1))+1;
  IF CURRENT>1 THEN CUR_LEN=2;
               ELSE CUR_LEN=1;
  IF B_F_MAP.#_FS(CURRENT,INDEX_IN,INDEX_L2)=0 THEN GO TO BACKTRACK;
  PTR_L1=PTR_L1+1;
  L1.STACK(PTR_L1)=SUBSTR(L2,1,CUR_LEN);
  L1.OUTPUT(PTR_L1)=B_F_MAP.OUTPUT(CURRENT,INDEX_IN,INDEX_L2,1);
  L1.INPUT(PTR_L1)=INDEX_IN;
  L1.WHICH(PTR_L1)=1;
```

```
  L1.LEN(PTR_L1)=LENGTH(B_F_MAP.STACK(CURRENT,INDEX_IN,INDEX_L2,1));
  L1.MACHINE(PTR_L1)=CURRENT;
  L2=B_F_MAP.STACK(CURRENT,INDEX_IN,INDEX_L2,1)||SUBSTR(L2,CUR_LEN+1);
  I=I+1;
  GO TO NEXT_SYMBOL;
BACKTRACK:
  IF I=1 THEN IF L2=START THEN DO;  FOUND='0'B;  RETURN;  END;
  IND1=INDEX(NON_TERM,SUBSTR(L1.STACK(PTR_L1),1,1));
  IND2=L1.INPUT(PTR_L1);
  IND3=L1.MACHINE(PTR_L1);
  IF L1.WHICH(PTR_L1)<B_F_MAP.#_FS(IND3,IND2,IND1) THEN DO;
      ALTERN,L1.WHICH(PTR_L1)=L1.WHICH(PTR_L1)+1;
      L2=B_F_MAP.STACK(IND3,IND2,IND1,ALTERN)||
                        SUBSTR(L2,L1.LEN(PTR_L1)+1);
      L1.OUTPUT(PTR_L1)=B_F_MAP.OUTPUT(IND3,IND2,IND1,ALTERN);
      L1.LEN(PTR_L1)=LENGTH(B_F_MAP.STACK(IND3,IND2,IND1,ALTERN));
      GO TO NEXT_SYMBOL;
    END;
  L2=L1.STACK(PTR_L1)||SUBSTR(L2,L1.LEN(PTR_L1)+1);
  PTR_L1=PTR_L1-1;
  I=I-1;
  GO TO BACKTRACK;
END BETA_MACHINE;

DELTA_MACHINE:  PROCEDURE(INPUT,FOUND,OUT);
DCL (INPUT,OUT) CHAR(*) VAR, FOUND BIT(*);
DCL 1 L1(N),
      5 INPUT FIXED BIN(16),
      5 OUTPUT CHAR(2) VAR,
      5 STACK CHAR(2) VAR,
      5 MACHINE FIXED BIN(16),
      5 WHICH FIXED BIN(16),
      5 LEN FIXED BIN(16);
  I=1;  L2=START;     PTR_L1=0;
NEXT_SYMBOL:
  IF I = N THEN IF L2='$' THEN DO;
```

```
      OUT='';  DO I=1 TO PTR_L1;  OUT=OUT||L1.OUTPUT(I);  END;
      FOUND='1'B;     RETURN;     END;
   INDEX_IN=INDEX(TERM,SUBSTR(INPUT,I,1));
   IF INDEX_IN=0 THEN GO TO BACKTRACK;
   INDEX_L2=INDEX(NON_TERM,SUBSTR(L2,1,1));
   IF INDEX_L2=0 THEN GO TO BACKTRACK;
   IF SUBSTR(L2,2,1)=',' THEN CURRENT=2;
                         ELSE CURRENT=1;
   IF D_F_MAP.#_FS(CURRENT,INDEX_IN,INDEX_L2)=0 THEN GO TO BACKTRACK;
   PTR_L1=PTR_L1+1;
   L1.STACK(PTR_L1)=SUBSTR(L2,1,CURRENT);
   L1.OUTPUT(PTR_L1)=D_F_MAP.OUTPUT(CURRENT,INDEX_IN,INDEX_L2,1);
   L1.INPUT(PTR_L1)=INDEX_IN;
   L1.WHICH(PTR_L1)=1;
   L1.LEN(PTR_L1)=LENGTH(D_F_MAP.STACK(CURRENT,INDEX_IN,INDEX_L2,1));
   L1.MACHINE(PTR_L1)=CURRENT;
   L2=D_F_MAP.STACK(CURRENT,INDEX_IN,INDEX_L2,1)||SUBSTR(L2,CURRENT+1);
   I=I+1;
   GO TO NEXT_SYMBOL;
BACKTRACK:
   IF I=1 THEN IF L2=START THEN DO;  FOUND='0'B;  RETURN;  END;
   IND1=INDEX(NON_TERM,SUBSTR(L1.STACK(PTR_L1),1,1));
   IND2=L1.INPUT(PTR_L1);
   IND3=L1.MACHINE(PTR_L1);
   IF L1.WHICH(PTR_L1)<D_F_MAP.#_FS(IND3,IND2,IND1) THEN DO;
      ALTERN,L1.WHICH(PTR_L1)=L1.WHICH(PTR_L1)+1;
      L2=D_F_MAP.STACK(IND3,IND2,IND1,ALTERN)||
                       SUBSTR(L2,L1.LEN(PTR_L1)+1);
      L1.OUTPUT(PTR_L1)=D_F_MAP.OUTPUT(IND3,IND2,IND1,ALTERN);
      L1.LEN(PTR_L1)=LENGTH(D_F_MAP.STACK(IND3,IND2,IND1,ALTERN));
      GO TO NEXT_SYMBOL;
    END;
   L2=L1.STACK(PTR_L1)||SUBSTR(L2,L1.LEN(PTR_L1)+1);
   PTR_L1=PTR_L1-1;
   I=I-1;
   GO TO BACKTRACK;
```

```
END DELTA_MACHINE;

 /*  BODY OF MAIN PROGRAM   */
FMT1:FORMAT(A,A(N-1),4 A);
  GET LIST(TERMS,NON_TERMS);
  TERM='';  DO I=1 TO #_TERMS;  TERM=TERM||TERMS(I);  END;
  NON_TERM='';
  DO I=1 TO #_NON_TERMS;  NON_TERM=NON_TERM||NON_TERMS(I);  END;
  I=1;
  DO L=1 TO #_TERMS;
      J=VERIFY(SUBSTR(SPECIAL,I),TERM||NON_TERM);
      I=I+J;
      FANCY=FANCY||SUBSTR(SPECIAL,I-1,1);
    END;
GET LIST(START);
  START2=START||'$';
  START=START||',$';
  OLD='';  NEW='';
  DO OUT=1 TO #_NON_TERMS;
      GET EDIT(CHARS)(COL(1),80 A(1));
      DO J=1 TO 80;
          IF CHARS(J)¬=' ' THEN OLD(OUT)=OLD(OUT)||CHARS(J);
        END;
    END;
  CALL NEW_GRAMMARS;
  PUT PAGE EDIT('RESULTS OF ERROR CORRECTING ALGORITHM FOR CONTEXT-'||
        'FREE GRAMMARS','N (NONTERMINALS): ',NON_TERMS,
        'T (TERMINALS): ',TERMS,'S (START SYMBOL): ',START,
        'ORIGINAL PRODUCTIONS:',OLD,'ALPHA PRODUCTIONS:',NEW(1,*),
        'BETA PRODUCTIONS:',NEW(2,*),'DELTA PRODUCTIONS:',NEW(3,*))
    (COL(20),A,COL(10),A,(#_NON_TERMS)(A,X(1)),COL(10),A,(#_TERMS)
        (A,X(1)),COL(10),2 A,4 (COL(1),A,(#_NON_TERMS)(COL(3),A)));
  CALL GET_F_MAP;
  CALL GET_ALPHA_F_MAP;
  CALL GET_BETA_F_MAP;
  CALL GET_DELTA_F_MAP;
```

```
  DO I=1 TO #_TERMS;
      DO J=1 TO #_NON_TERMS;
          A_F_MAP(1,I,J)=F_MAP(I,J);
          B_F_MAP(1,I,J)=F_MAP(I,J);
          D_F_MAP(1,I,J)=F_MAP(I,J);
        END;
    END;
IF FMAP|AMAP|BMAP|DMAP THEN PUT DATA(FMAP,#F,AMAP,#A,BMAP,#B,DMAP,#D);
GET DATA(#_TESTS);
DO CONTROL2=1 TO #_TESTS;
 GET LIST(TEST(1));
 CALL GENERATE;
 DO CONTROL3=1 TO 4;
  INPUT=TEST(CONTROL3)||'$';
  N=LENGTH(INPUT);
  CALL MACHINE(INPUT,FIND_PARSE);
  IF FIND_PARSE THEN PUT SKIP(2) EDIT('THIS IS A CORRECT STRING: ',
                        INPUT)   (A,A(N-1));
                ELSE DO;
      CALL DELTA_MACHINE(INPUT,FIND_PARSE,OUTPUT);
      IF FIND_PARSE THEN PUT SKIP(2) EDIT(S1,INPUT,S2,S3(3),S4,OUTPUT)
        (R(FMT1));
                    ELSE DO;
          CALL BETA_MACHINE(INPUT,FIND_PARSE,OUTPUT);
          IF FIND_PARSE THEN PUT SKIP(2) EDIT(S1,INPUT,S2,S3(2),S4,
            OUTPUT)    (R(FMT1));
                       ELSE DO;
              CALL ALPHA_MACHINE(INPUT,FIND_PARSE,OUTPUT);
              IF FIND_PARSE THEN PUT SKIP(2) EDIT(S1,INPUT,S2,S3(1),
                 S4,OUTPUT)    (R(FMT1));
                            ELSE PUT SKIP(2) EDIT(S1,INPUT,
                                 ' CANNOT BE CORRECTED')    (R(FMT1));
END BEGIN_BLOCK;
END_ERROR_CORRECTOR:  END ERROR_CORRECTOR;
```

VITA

George Hensley Miles was born in Madison, Wisconsin, on March 4, 1951. He attended elementary schools in Starkville, Mississippi and Knoxville, Tennessee and was graduated from Bearden High School in 1969. The following September he entered The University of Tennessee, and in June 1973 he received a Bachelor of Science degree in Mathematics. He has been employed by The University of Tennessee Computing Center since June 1973.

He entered the Graduate School at The University of Tennessee in June 1973, and received the Master of Science degree with a major in Computer Science in August 1974. He is a member of Phi Eta Sigma, Phi Beta Kappa, Phi Kappa Phi, and the Association of Computing Machinery.

He is married to the former Rita Ann Sweet of Knoxville, Tennessee.