74-6967

LAZARUS, Paul Shantraj, 1944-
AUTOMATIC ERROR CORRECTION IN SYNTAX-
DIRECTED COMPILERS.

The University of Oklahoma, Ph.D., 1973
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

THE UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

AUTOMATIC ERROR CORRECTION IN SYNTAX-DIRECTED COMPILERS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

degree of

DOCTOR OF PHILOSOPHY

BY

PAUL SHANTRAJ LAZARUS

Norman, Oklahoma

1973

AUTOMATIC ERROR CORRECTION IN SYNTAX-DIRECTED COMPILERS

APPROVED BY

DISSERTATION COMMITTEE

ACKNOWLEDGEMENTS

I wish to express my gratitude to Dr. Stephen Feyock for his guidance, patience and especially for editing the writing of this thesis. I would also like to thank Doctors J. A. Payne, V. B. Gylys and J. Christensen for serving on my committee and for giving valuable comments. I appreciate Ms. Judy Thain for her encouragement and helping me in preparing the draft and the final copy of this thesis.

TABLE OF CONTENTS

CHAPTER ONE
INTRODUCTION

CHAPTER TWO
SYNTAX-DIRECTED COMPILING

CHAPTER THREE
AN ERROR CORRECTION ALGORITHM

CHAPTER FOUR
IMPLEMENTATION OF THE ALGORITHM

CHAPTER FIVE
CONCLUSION

CHAPTER ONE
INTRODUCTION

1.1 THE PURPOSE OF THIS WORK

Computer programs written to solve non-trivial programs almost invariably contain errors. Every programmer knows the long and tedious chore of correcting errors or "bugs" in the program. Therefore, methods are being developed to enable compilers to share the burden of debugging with the programmer. The error correction methods in existing compilers are geared to the particular language they are dealing with. In this thesis we shall present a technique for error correction that is language-independent.

In writing computer programs, errors are committed at various levels. At the highest level* we have the logical errors. The computer accepts a program with logical errors and executes it but does not produce what the program is meant to produce. This is (logical) correctness of a program and we do not deal with it here. Theoretical and practical work has been done in this area by Rustin R. (1971), Floyd, R. W. (1967).

At the lowest level the user may make errors in the instructions to the operating system. These errors are actually caused by violating the syntax of the language of the operating system. Therefore, we can regard them as syntax errors.

*We consider the level of an error higher than the level of another error if the former is not detected until the latter has been corrected.

Compile-time errors, which we are going to consider, occur for two reasons. In a program either the syntax or the semantics may be unacceptable to the compiler; accordingly, we have syntactic or semantic errors. We have chosen to work with syntactic error correction, since the systematic approach to compilation represented by syntax-directed compilers makes possible a similarly systematic approach to the automatic correction of syntactic errors. Also, without resolving syntactic errors compilation cannot be continued. At or soon after the occurrence of a syntactic error, the compiler "gets stuck" and cannot proceed unless the compiler is provided with a scheme either for correcting the error or for making certain changes in the "state of compiling". The objective of this work is to develop a scheme that will correct the errors that we believe are "most likely" to occur. In the event of an "unlikely error", the scheme will enable the compiler to proceed further by merely changing the "state of compiling".

Even though we are confining ourselves to syntactic errors, we do not completely ignore semantic errors. Semantic errors cause compilers to assign unexpected meaning to the program. We are not concerned with the problem of correcting semantic errors committed by the programmer. However, we are concerned with the semantic errors introduced by corrections of the syntactic errors.

1.2 SURVEY OF PREVIOUS WORK DONE IN RELATED FIELDS

Hopcroft and Ullman (1966) establish fundamental results on error correction in formal languages. For a language L, they define the set $E_e(L)$ consisting of all strings w, such that there is a string x in L with the same number of symbols as w and differing from w in at most e symbols. In other words, $E_e(L)$ is the set of all strings within e-Hamming distance of the strings in L. They prove that the set of regular languages, the set of context-free languages and the set of context-sensitive languages are closed under the operation E_e. However, the set of deterministic languages is not closed under E_e.

One of the earliest papers on error correction is by E.T. Irons (1963). Irons uses top-down parsing. In order to avoid backup, he constructs all possible parses in parallel. At any step during the parse, one or more parse trees have been constructed; some branches are incomplete. An error is detected when no partial tree can be further built. Then all input symbols are successively examined and discarded until one is found which can be a node of some incomplete branch. A string of symbols is constructed such that, if inserted before this input symbol, it will allow the parsing to continue.

If an error is not detected at its occurrence in the string, the correction suggested by Irons may not be what the programmer intended. The only way to find the correct interpretation of the string is to go back and reinterpret the string from the point of error. J. P. Lévy (1971) introduces the notion of "backward move". After the parser detects the existence of an error it starts scanning right to left finding the least number of characters in which a correction may be needed. This substring he calls the left context of the error. Then the parser "moves forward" constructing all possible interpretations until all interpretations are equivalent*. Lévy admits that this model is not practical for the conventional context-free description of programming languages. In order to make it more practical, he proposes the use of "bracketed context-free" description of programming languages. He also proposes some heuristic restrictions on the type of errors.

*Lévy defines equivalence of strings as follows: Two prefixes x and y of a language L are equivalent with respect to L iff

for every string z, $xz \in L \Leftrightarrow yz \in L$.

J. E. LaFrance (1971) describes an automatic error recovery technique for parsers using Floyd production language; he also extends his technique to parsers that use recursive descent. Techniques for the generation of production language parsers have been developed by a number of workers, including Beals (1969), Beals et al (1969), DeRemer (1968), Early (1966), Haynes and Schutte (1970). LaFrance uses the technique of Beals et al. to produce a top-down parser. The parser automatically detects an error when it observes unexpected symbols either on the stack or in the look-ahead symbols. Since the parser is predictive (top down), it knows what to expect in the look-ahead symbols. The existing string of input symbols is transformed according to the expectations of the parser along with a change on the top of the stack.

C. J. Burgess (1972) gives a method of error diagnostics for syntax-directed compilers. He considers the left-factor (LF) grammars, which constitute rather a large subclass of context-free grammars. He uses top-down parsing. To a given BNF grammar he adds what he calls "error categories", which will aid in detecting errors in the input string during parsing.

Compilers for CORC (a dialect of ALGOL), CUPL (a dialect of PL/1) and PL/C (a dialect of PL/1) try to correct all the errors in programs and execute them in spite of all errors. The error correction techniques in these compilers are ad hoc rather than systematic. In PL/C (Conway 1970) the syntactic analyzer, at each step, uses a transition table to decide what is to be done next. The rows in a transition table correspond to the last "state" of the analyzer and the columns correspond to the next input symbol. The entries in the transition table corresponding to an illegal combination of the last state of the analyzer and the next input symbol have addresses of error correction routines. PL/C also corrects semantic errors. Since the semantic analysis is performed as an independent pass rather than concurrently with the syntactic analysis, the syntactic corrections are performed without considering their effect on the semantics of the program. If the syntactic analyzer makes a correction which is syntactically correct but does not conform to the semantic conventions the semantic analyzer is unable to retract the decision of the syntactic analysis. PL/C includes the spelling correction scheme of Morgan (1970).

The IBM PL/1 (F level) compiler also corrects syntactic and semantic errors in source programs. The user has the option to indicate if the machine code for his program is to be executed in spite of errors. The diagnostics and the corrections are not very clear for two reasons: First, the messages are not printed with the offending source statements. Several messages for the same statement appear in different places. Second, messages often make references to statements and not to the exact position in the statement. For example, the message may indicate that a certain symbol was inserted in a certain statement, but there may be more than one place where the particular symbol could be inserted in that statement.

The error recovery scheme used in the XPL system (McKeeman et al. 1970) is rather primitive. The compiler writer gives a list of symbols, like ";" , "DO" , "IF" etc. which indicate the end of a statement or the beginning of a new statement. When an error is detected, input symbols are examined and discarded until one is found which is in the list. Then the symbols on the top of the stack are successively examined until the current input symbol can legally follow what remains on the stack.

Leinius (1970) presents an elaborate method of recovery for bottom-up parsing of simple precedence grammars. His technique is automatic. He also explains how his technique can be extended to languages that are not simple precedence.

L. R. James (1972) implements Leinius' method for SPL, a subset of PL/1. He uses Morgan's (1970) spelling correction algorithm, and compiles statistics from samples of programs written in SPL. Besides the implementation and the statistics there is nothing novel in this work.

1.3 THE PHILOSOPHY OF THIS WORK

The basic philosophy of our approach is to restrict our efforts to the "most likely" errors. It is assumed that the most likely errors are:

1) a missing symbol,
2) a wrong symbol,
3) a symbol in excess,
4) two adjacent symbols permuted.

We make a further assumption that there is only one error per "substructure" (to be defined in detail below). This second assumption is made not because multiple errors in a substructure are rare, but because automatic correction techniques that correct multiple errors become impractical to implement for practical programming languages. The theory of an automatic method to correct multiple errors has been developed by Lévy (1971).

As mentioned, our algorithm is automatic rather than ad hoc. In compilers that use ad hoc correction techniques the correction algorithm consists of a collection of "hand made" routines. After the detection of an error it is determined which one of these hand-made routines should handle the error. Each of these special routines can correct an error more efficiently than a general automatic algorithm. However, our philosophy is to present an algorithm that is language-independent. Therefore, our algorithm corrects errors using only the information in the grammar of the language. This makes our algorithm very portable.

It is true that errors committed by naive programmers, who know little about the structure of the language, may fail to satisfy the above requirements. In such cases our approach is to delete the offending statements and proceed. The purpose of providing the compiler with an error correction facility is not to encourage the programmers to develop the attitude that "the compiler will correct the errors anyway". However, errors occur in spite of careful programming, and an attempt by the compiler to correct errors will save human time as well as computer time.

1.4 APPROACH AND OUTLINE

Treatment of syntax errors in the literature is mostly heuristic. Most often, it is recovery rather than correction which is undertaken. Except for Lévy's theoretical treatment, the existing error correction techniques insist on making corrections at the point where the existence of error is detected. The existence of an error, however, is not always detected at the point of its occurrence. Also, delay in detecting the existence of error occurs more often with some parsers than others. Therefore, if an error correction technique is to be applicable to a large class of parsers it must solve the problem of locating the exact position of error. In Chapter Three we present a method of locating the position of error.

Lévy's model is both formal and fairly realistic but its implementation becomes difficult for most programming languages. We have therefore simplified Lévy's model so that its implementation is feasible, yet realistic enough to correct the most likely errors.

1.41 An Outline of the Dissertation

Chapter Two introduces three important classes of parsers: LR parsers, LL parsers and Mixed Strategy parsers. We choose these parsers since syntax-directed techniques for these are widely known. A discussion of syntax-directed parsing is included. Formal definitions and the most important properties of the above-mentioned parsers are given.

Chapter Three describes our algorithm for correcting errors. After the parser detects the existence of an error a string between the previous delimiter and the next delimiter and the next delimiter is isolated. This string corresponds to a "substructure" in the language. From this erroneous input string, strings called correction strings are generated which differ at most by one symbol from the input string. These correction strings are then subjected to a series of stringent tests. After all the correction strings undergo tests a decision about the final correction is made. The first section discusses detection of errors. Capabilities of different parsers to detect errors early in the string are discussed and causes for delay in detecting errors are given. The second section considers the generation and testing of correction strings.

Chapter Four describes the implementation of our algorithm. The XPL System which was used to generate the compiler of the implementation in briefly described in the first section. Section Two is "Detection and Location of Errors". Section

Three describes the generation of correction strings. Testing of these correction strings, for syntactic and semantic correctness is given in Section Four. After testing all the correction strings, a decision is made about the conclusion of the correction process for the particular error; Section Five considers such correction decisions. Section Six explains the process of backing up the parser.

Chapter Five contains a few concluding remarks. First, the significance of this research is given. Then the performance of the implementation is evaluated. Finally, topics are mentioned where further work would improve our error correction algorithm.

The appendix is divided into three parts. Appendix A contains BNF grammar for the XPL language. Appendix B is a listing of the important procedures comprising EXPL, the compiler with our error correction algorithm. Appendix C contains results of sample programs run under EXPL with our error correction algorithm. One sample program is run both under EXPL and PL/1 F level compilers; the results show how EXPL corrects certain errors when the PL/1 F compiler fails.

CHAPTER TWO
SYNTAX-DIRECTED COMPILING

In this chapter we shall discuss three important classes of parsers: LR parsers, LL parsers and Mixed Strategy parsers. Our purpose is twofold. First, formal definitions and properties of the parsers are given with references to the sources where the proofs and further discussion can be found. Second, a simple example is used to illustrate the working of each of these parsers. Sources are quoted where more formal algorithms and their proofs are to be found. Sections 2.1 contains definitions of the terms to be used in the rest of the chapter. Section 2.2 is on sytax-directed parsing. The three deterministic parsers mentioned are described in Section 2.3. Section 2.4 briefly describes semantic analysis.

2.1 DEFINITIONS OF THE TERMINOLOGY

In this section we shall define the terms to be used in the rest of the chapter.

2.11 Vocabulary and Strings

We will use the basic terminology of set theory without definition.

A vocabulary or alphabet is a non-empty finite set of elements called symbols.

A string is a finite sequence of symbols from a vocabulary. The empty string, denoted by e, is the sequence containing no symbols.

The length of a string s, written $|s|$, is the number of symbols in it. If s and t are two strings, their concatenation st is the string obtained by writing the string t after the string s. For any string s, we see that

$$es = se = s.$$

If r, s, and t are three strings such that $r = st$ then s is the head of r, written $s = head(r)$. If $|s| = n$, then s is the n-head of r, written $s = head_n(r)$. Also, t is called the tail of r, written $t = tail(r)$, and if $|t| = n$, then t is the n-tail of r, written $t = tail_n(r)$.

For vocabulary V, the set of all sequences of symbols of V is denoted by V^*. This includes the empty string e. The set of all non-empty strings is V^+. Thus $V^* = V^+ \cup \{e\}$.

2.12 Grammars, Sentential Forms and Languages

Let V be an alphabet. A context-free (cf) production or rewriting rule is an ordered pair (A, x), usully written A ::= x, where A is a symbol and x is a string in V. A is the left part and x is the right part of the production. A production A ::= e is an e-production.

A context-free grammar (cfg) is a 4-tuple G = (N,T,P,S) where

1) P is a finite set of productions.
2) N is a set of non-terminals. A non-terminal is a symbol that appears as the left part of a production.
3) T is a set of terminals. A terminal is a symbol in V which is not a non-terminal.
4) S is a distinguished non-terminal called the goal or start symbol.

We shall use the following conventions to represent various symbols concerned with a grammar:

1) a,b,c,d and f represent terminals.
2) A,B,C,D and S represent non-terminals; S represents the start symbol.
3) R,S,T,U,...,Z represent either non-terminals or terminals.
4) r,s,t,u,...,z represent strings of non-terminals and terminals.

We say a string v <u>directly produces</u> the string w, written

$$v \Rightarrow w,$$

if we can write

$$v = xUy, \text{ and } w = xuy$$

for some string x and y, where U ::= u is a rule of G. We also way that w is a <u>direct derivation</u> of v, or that w <u>directly reduces</u> to v. We say v <u>produces</u> w, or w <u>reduces</u> to v, written v $\Rightarrow$ w, if there exists a sequence of direct derivations

$$v = u_0 \Rightarrow u_1 \Rightarrow u_2 \ldots \Rightarrow u_n = w \quad \text{where } n \quad 0.$$

The sequence is called a <u>derivation of length n</u>. Also, we write

$$v \Rightarrow * w \quad \text{if} \quad v \Rightarrow w \quad \text{or} \quad v = w.$$

A direct derivation xUy $\Rightarrow$ xuy is <u>rightmost</u>, written

$$xUy \underset{rm}{\Rightarrow} xuy,$$

if y contains only terminals. A direct derivation xUy $\Rightarrow$ xuy is called <u>leftmost</u>, written

$$xUy \underset{lm}{\Rightarrow} xuy,$$

if x contains only terminals. A derivation w $\Rightarrow$ v is called a rightmost derivation, witten $w \underset{rm}{\Longrightarrow} v$, if every direct derivation in it is rightmost. Similarly we define leftmost derivation.

A string s is called a sentential form if it is derivable from the dishtinguished symbol S, that is, if $S \Longrightarrow * s$. A sentential form consisting only of terminals is called a sentence. The set of all sentences:

$$L(G) = \{ w \mid S \Longrightarrow * w, \text{ and } w \in T^* \}$$

is the language generated by G.

Let $w = xuy$ be a sentential form in grammar G. Then u is called a phrase of the sentential form w for a non-terminal U if

$$S \Longrightarrow * xUy \text{ and } U \Longrightarrow u.$$

u is called a simple phrase if $S \Longrightarrow * xUy$ and $U ::= u$. The handle of a sentential form is its leftmost simple phrase.

We say that a cfg $G = (V,T,P,S)$ is e-free if either

1. P has no e-productions, or
2. There is exactly one e-production: $S ::= e$, and S does not appear on the right side of any production in P.

In the future, we will assume cfgs to be e-free. This is justified by the following theorem:

> Given any context-free grammar, $G = (V,T,P,S)$, we can find an e-free cfg $G' = (V',T',P',S')$ such that $L(G) = L(G')$. (See Ullman and Hopcroft, 1969, for proof.)

We shall end this section with the definition of FIRST(s), where s is a string of symbols.

$$\text{FIRST}(s) = \{ x \mid s \Rightarrow^* xs' \text{ and } |x| = k, \\ \text{or } s \Rightarrow^* x \text{ and } |x| < k, \\ \text{where } x \text{ is a string of terminals only.} \}$$

That is, FIRST(s) consists of all terminal prefixes of length k or less.

2.2 SYNTAX-DIRECTED PARSING

2.21 Syntax-Directed Vs. Ad Hoc Methods

Since the late fifties tools have been developed to make the job of compiler writing easier and more efficient. Many systems, called compiler compilers (cc) have been invented. Compiler compilers aid compiler writing the same way programming languages aid writing algorithms for computers. BMCC (Brooker-Morris compiler compiler), Floyd's (1961) Production Language, Shorre's (1964) META, McClure's (1965) TMG, Cheatham's (1965) TGS-II, Feldman's (1966) FSL (formal semantic language), Mercer's (1970) TWINKLE and SKELETON of McKeeman et al. (1970) are examples of such systems.

A compiler written using a cc system may require more memory space than the compiler written ad hoc in assembler language for the same purpose. However, using a cc has the following advantages:

1. Formality,
2. Portability,
3. Programming ease.

Formality: Compiler compiler make compiler writing formal and systematic. In traditional ad hoc compiler writing, heuristic knowledge of the language is used. Heuristic knowledge of a language differs from person to person. Hence, each person using the ad hoc method ends up inventing his own "tricks" for his compiler.

Portability: Since syntax-directed compiling uses the syntax (or the tables derived from the syntax) of the language rather than heuristic knowledge of the language, it is easy to modify the compiler when the language is changed. Changes in the language are expressed in terms of changes in the syntax of the language which in turn produce changes in the tables. With the new tables the compiler can parse the new language.

Ease of Programming: The languages in which compiler compilers are written are higher-level languages which are easier to use than assembler language. Debugging is easier in higher-level languages than in assembler languages. Also, changes from machine to machine becomes easier if the programming is in a higher-level language, since higher-level languages are less machine dependent than assembler languages.

Syntax-directed compiling is done in the following steps:

1. lexical analysis,
2. syntactic analysis,
3. semantic analysis.

Lexical analysis and syntactic analysis are discussed in Section 2.22 and 2.23. Semantic analysis will be considered in Section 2.4.

2.22 Lexical Analysis or Scanning

Lexical analysis (scanning) is the simplest part of compiling. To make the storing and transfer of the source program between different phases of compiling efficient, the terminal symbols of the language can be represented by integers called tokens. In other words, each terminal symbol can be associated with a unique integer. For example, the scanner of our implementation for the XPL language associates

;	with	1,
)	with	2,
(	with	3,
	etc.	

There are certain terminal symbols for which the semantic analyzer and the code generator need to know the actual symbol as well as the token. The names of the identifiers and the values of the constants are such symbols. A scanner can be as simple as just reading a single character on the input medium and converting it to an integer. This makes the job of the syntactic analyzer more burdensome. A scanner could recognize the whole symbol, for example, BAL_OF_TODAY as an identifier, or 125.6 E 03 as a decimal floating point constant. In other words, the scanner does a small amount of parsing. The scanner can also do some error-correction in numerical constants. For example, the scanner can detect and correct the error in the constant 1.2U25 better than the syntactic analyser. Finally, the scanner can recognize comments (remarks) and delete them from the information sent to the future phases of compiling.

2.23 Syntactic Analysis

Syntactic analysis (or parsing) is the process of determining if a given input string is a sentence in the given language. From the definition of "sentence" (Section 2.1) this implies the construction of a derivation for the string. Consider the language L described by the production set P:

1. E ::= E + T
2. E ::= T
3. T ::= T * F
4. T ::= F
5. F ::= a

The string a + a * a is a sentence in the language L, since there is a derivation for it. For example,

(2.231) $E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * a \Rightarrow E + F * a$
$\Rightarrow E + a * a \Rightarrow T + a * a$
$\Rightarrow F + a * a \Rightarrow a + a * a.$

is a rightmost derivation for a + a * a. Also, there is a leftmost derivation:

(2.232) $E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T * F$
$\Rightarrow a + F * F \Rightarrow a + a * F \Rightarrow a + a * a.$

The leftmost derivation as well as the rightmost derivation can be represented by a diagram called the syntax tree.

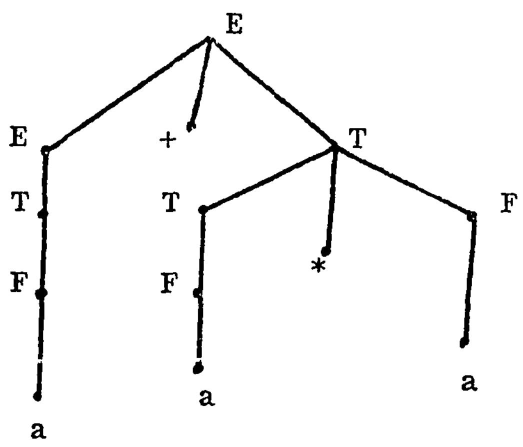


Figure 2.1 Syntax Tree of a + a * a.

A derivation to derive a sentence is also called a <u>parse</u>. The leftmost derivation is called the <u>left parse</u>. The rightmost derivation, with the direct derivations written in reverse order, is called the <u>right parse</u>. For example, the parse (2.232) for the sentence $a + a * a$ in L, is a left parse. The reverse of (2.231), namely

$$
\begin{aligned}
a + a * a &\Rightarrow F + a * a \Rightarrow T + a * a \Rightarrow E + a * a \\
&\Rightarrow E + F * a \Rightarrow E + T * a \\
&\Rightarrow E + T * F \Rightarrow E + T \Rightarrow E.
\end{aligned}
$$

is a right parse. For a given sentence the process of finding left parse (right parse) is called the <u>top-down parsing</u> (<u>bottom-up parsing</u>).

In top-down parsing, we start with the goal symbol and build the (parse) tree down to the terminals. In bottom-up parsing, on the other hand, we start with the terminals and build the tree toward the goal symbol. In either case, we observe the following facts:

1. The root of the tree is the goal symbol.
2. The leaves are terminal symbols and the nodes are non-terminal symbols.
3. Each node that is not a leaf is the left side of a production and the immediate branches from the node represent the right side of that production.

All the parsing methods described above are left-to-right in the sense that they scan the input string from left to right. We could similarly define right-to-left parsing methods.

2.3 DETERMINISTIC PARSERS

2.31 <u>Pushdown Automaton</u>

We now introduce the pushdown automaton--a recognizer that is a natural model for syntactic analyzers of context-free languages.

<u>Definition</u>: A <u>pushdown automaton</u> (PDA) is a 7-tuple

$$P = (Q, I, \Gamma, M, q_0, T_0, F)$$

where

1. Q is a finite set of state symbols representing the possible states of the finite state control.
2. I is a finite <u>input alphabet</u>.
3. Γ is a finite alphabet of <u>pushdown list symbols</u>.
4. M is a mapping from $Q \times (I \cup \{e\}) \times \Gamma$ to the finite subsets of $Q \times T^*$.
5. $q_0 \in Q$ is the <u>initial state</u> of the finite control.
6. $T_0 \in \Gamma$ is the symbol that appears initially on top of the pushdown list.
7. $F \subseteq Q$ is the set of <u>final states</u>.

A configuration of P is a triple (q,w,t) in QxI^*xT^*, where

1. q represents the current state of the finite control.
2. w represents the unused portion of the input. The first symbol of w is under the input head. If w is e, then it is assumed that all of the input tape has been read.
3. t represents the contents of the pushdown list.

The leftmost symbol of t is the topmost pushdown symbol. If t = e, then the pushdown list is assumed to be empty.

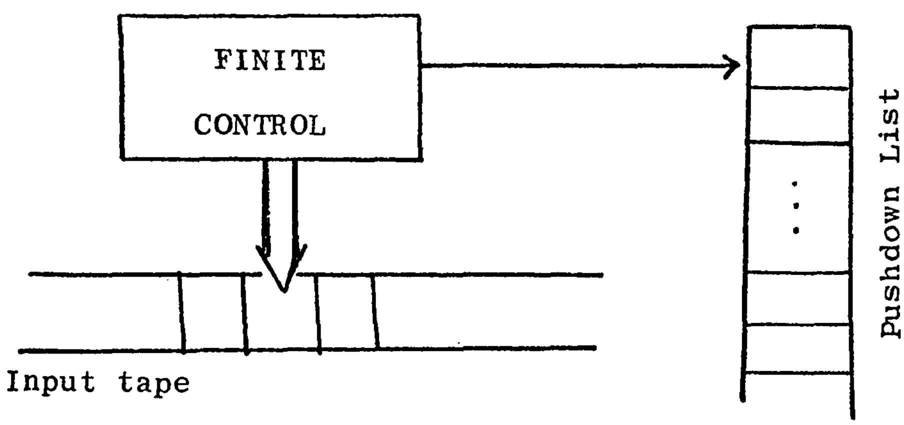


Figure 2.2 Pushdown Automaton

A PDA $P = (Q, I, \Gamma, M, q_0, T_0, F)$ is said to be <u>deterministic</u> (DPDA) if for each q in Q and T in Γ either

1. $M(q,i,T)$ contains at most one element for each i in I and $M(q,e,T) = \phi$; or
2. $M(q,i,T) = \phi$ for all i in I, and $M(q,e,T)$ contains, at most, one element.

These two restrictions imply that a DPDA has at most one choice of most in any configuration. Thus in practice it is much easier to simulate a deterministic PDA than a non-deterministic PDA. The space and time requirements of deterministic PDA's are linear with respect to the length of input strings. We shall consider the following important classes of deterministic parsers:

1. LR(k) parsers
2. LL(k) parsers
3. Precedence parsers
4. Bounded context parsers.

2.32 <u>LR(k) Parsers</u>

<u>Definition</u>: Let G = (V,T,P,S) be a cfg. We say G is an <u>LR(k) grammar</u>, $k \geq 0$, if the three conditions

1. $S \underset{rm}{\Rightarrow}^* rAw \underset{rm}{\Rightarrow} rsw$,
2. $S \underset{rm}{\Rightarrow}^* rBx \underset{rm}{\Rightarrow} rsy$, and
3. $FIRST_k(w) = FIRST_k(y)$

imply that aAy = cBx. That is, r = t, A = B, and x = y.

A language generated by an LR(k) grammar is called an <u>LR(k) language</u>. A language has an LR(k) parser if it has an LR(k) grammar to describe it.

LR(k) grammars are the largest class of unambiguous grammars for which we can construct deterministic parsers. In fact,

> <u>Theorem</u>: For any deterministic language L there is an LR(k) grammar G, for some $k \geq 0$, such that G generates L.

(See Aho and Ullman (1972) for proof).

Let us consider a deterministic language and see how we can construct an LR(k) parser for it. Consider the language described by the production set P_1 :

EXPR ::= EXPR + TERM
EXPR ::= TERM
TERM ::= TERM * FACT
TERM ::= FACT
FACT ::= a

EXPR, TERM, FACT are the non-terminals. EXPR is the goal symbol. i, + and * are the terminal symbols. Throughout this chapter we shall refer to this language by the name "Expression Language."

We now show how to construct a PDA $P = (Q, I, \Gamma, M, q_0, T_0, F)$ for the above language. The finite control Q has four states:

1. Push: push the input symbol that is presently under the scanner onto the pushdown list.
2. Reduce: do not move the input tape, but reduce the alphabet symbols on the pushdown list by using a production.
3. Accept.
4. Error.

The alphabet I is the set:

(EXPR, TERM, FACT, a, t, * and $)

The mapping M is given in the form of a table in Figure 2.32. q_0 is the initial state when $ has been pushed on Γ. T_0 is the symbol initially on top of Γ. F is the final state "accept".

We now describe the construction of M and Γ. We may assume that only terminals and non-terminals are on Γ. Then to define the mapping M we will have to know the entire contents of Γ. Instead, suppose new symbols, say $T_0, T_1, \ldots, T_n$ are used to represent all possible correct configurations of the stacks of terminals and non-terminals on Γ. Then each time the contents of Γ are changed, a T_i can be placed on the top of Γ and M can consult only the top of Γ to decide the move. The possible configurations $T_0, T_1 \ldots$ and the transition table M are given in Figure 2.3. Figure 2.3 contains the tables, called LR(k) tables, for our Expression Language. The tables were computed by hand using the algorithm given by the flow chart of Figure 2.4. A more formal algorithm to compute LR(k) tables and a proof of its validity are given in Aho and Ullman (1972, Algorithm 5.12, Theorem 5.12). Their algorithm (5.7 Ibid) of parsing using these LR(k) tables is slightly different from ours. LR(k) parsers are quite powerful; they can parse any deterministic language, but the LR(k) tables become impractically large for practical programming languages. Optimization techniques for reducing the size of LR(k) tables are given (in Chapter Seven, Ibid) but seem less than convincing. Restrictions of LR(k) grammars have been proposed to make parsing practical. DeRemer introduced Simple LR(k) grammars and LALR(k) grammars (DeRemer 1969, 1971) which have efficient parsers.

	a	+	*	$
T_0	1, T_1	0	0	0
T_1	0	2, T_2	2, T_2	2, T_2
T_2	0	2, T_3	2, T_3	2, T_3
T_3	0	2, T_4	1, T_4	2, T_4
T_4	0	2, T_5	0	ACCEPT
T_5	1, T_7	0	0	0
T_6	1, T_8	0	0	0
T_7	0	2, T_9	2, T_9	2, T_9
T_8	0	2, T_{10}	2, T_{10}	2, T_{10}
T_9	0	2, T_{11}	2, T_{12}	2, T_{11}
T_{10}	0	2, T_3	2, T_3	2, T_3
T_{11}	0	2, T_4	2, T_{12}	2, T_4
T_{12}	1, T_{13}	0	0	0
T_{13}	0	2, T_{14}	2, T_{14}	2, T_{14}
T_{14}	0	2, T_{11}	2, T_{11}	2, T_{11}

T_0 :	$
T_1 :	$a
T_2 :	$F
T_3 :	$T
T_4 :	$E
T_5 :	$E'+
T_6 :	$T*
T_7 :	$E+a
T_8 :	$T*a
T_9 :	$E+F
T_{10}:	$T*F
T_{11}:	$E+T
T_{12}:	$E+
T_{13}:	$E+T*a
T_{14}:	$E+T*F

Figure 2.3

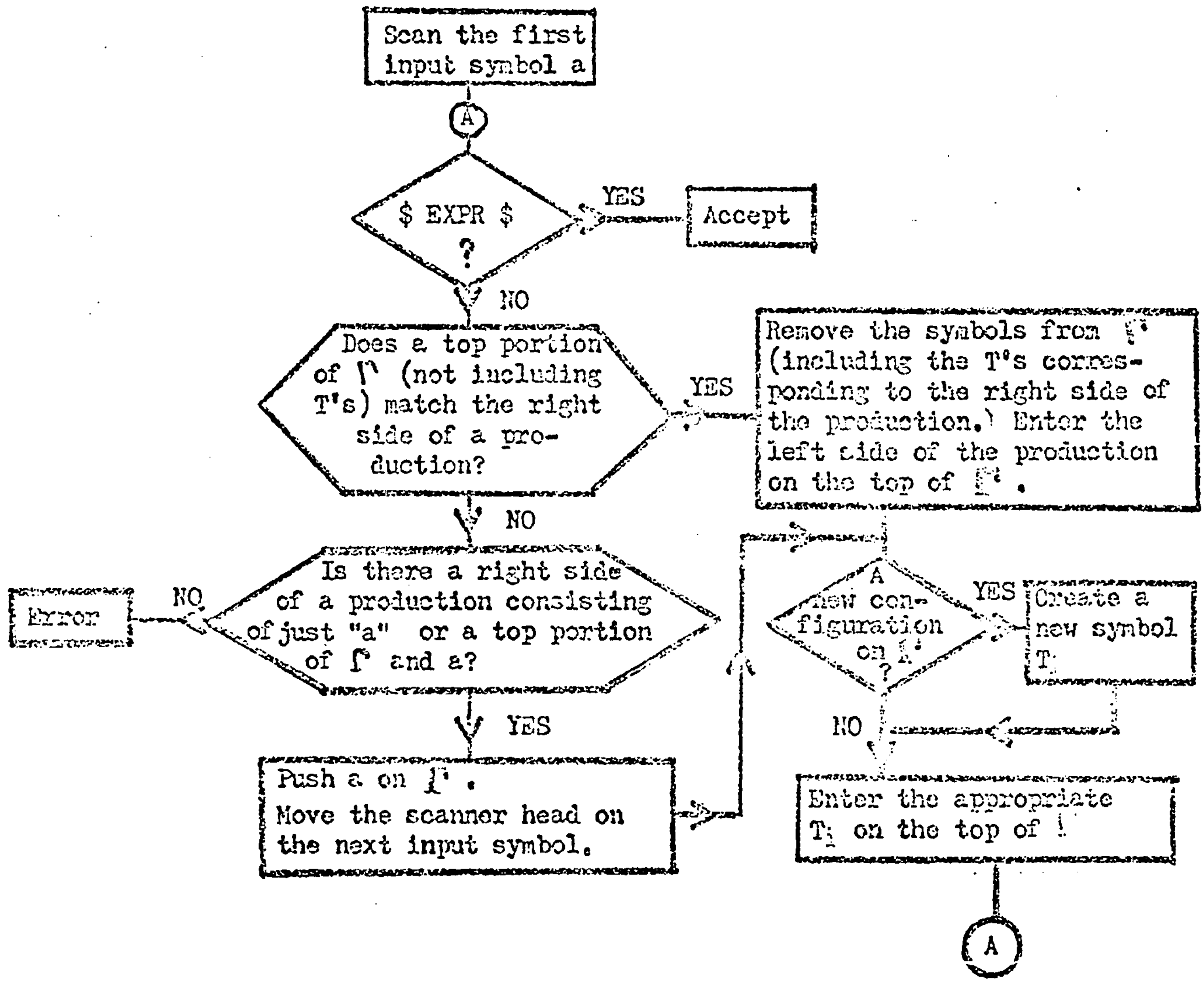

Scan the first input symbol a
A
$ EXPR $?
YES
Accept
NO
Does a top portion of Γ (not including T's) match the right side of a production?
YES
Remove the symbols from Γ (including the T's corresponding to the right side of the production.) Enter the left side of the production on the top of Γ.
NO
Is there a right side of a production consisting of just "a" or a top portion of Γ and a?
NO
Error
YES
Push a on Γ.
Move the scanner head on the next input symbol.
A new configuration on Γ?
YES
Create a new symbol Ti
NO
Enter the appropriate Ti on the top of Γ
A

FIGURE 2.4

2.33 LL(k) Parsers

Certain compilers that parse top-down must use back-tracking. For some languages back-tracking can be avoided by looking at certain input symbols in advance. For example, consider the grammar:

1. G ::= aE	2. G ::= Bd
3. E ::= bD	4. B ::= Ac
5. D ::= df	6. A ::= ab

G is the start (goal) symbol.

Suppose we want to parse the string abcd. We start with the goal symbol G. There are two productions to choose from: (1) and (2). At this point, just by looking at the first input symbol a we cannot tell which alternative to take. Suppose we choose (1).

$$G ::= aE$$

The first symbol of the input string, viz. a, matches. Next, we have to replace the non-terminal E. There is only one choice, hence we take it.

$$G \Rightarrow aE \Rightarrow abD \ .$$

A second match is found. We procede further with D. But this time

$$G \Rightarrow aE \Rightarrow abD \Rightarrow abcdf \ .$$

We failed to match the input string. Therefore, this is not the parse we were looking for. We made the wrong choice at the very first step. We start with G again and take the other alternate

$$G \Rightarrow Bd$$

which will lead to

$$G \Rightarrow Bd \Rightarrow Acd \Rightarrow abcd$$

the correct parse.

While making a choice between production (1) and (2), if enough (in this case, four) input symbols had been scanned, we could have made a correct choice. If such correct choices can always be made for any production the grammar is called an LL(k) grammar, where k is the maximum number of input symbols which must be scanned in advance. More formally,

Definition: Let G = (N,T,P,S) be a cfg. G is said to be LL(k), for some fixed integer $k \geq 0$, if whenever there are two leftmost derivations

1. $S \underset{lm}{\Rightarrow}^* wAr \underset{lm}{\Rightarrow}^* wrs \underset{lm}{\Rightarrow}^* wx$, and
2. $S \underset{lm}{\Rightarrow}^* wAr \underset{lm}{\Rightarrow}^* wtr \underset{lm}{\Rightarrow}^* wy$

such that $FIRST_k(x) = FIRST_k(y)$, then it follows that $s = t$. That is, the two productions

$$A ::= s \quad \text{and} \quad A ::= t$$

used in the two derivations are identical.

The grammar we just considered is LL(4). It should be noted that there are deterministic grammars that are not LL(k) for any k. For example, the grammar

(1) S ::= A	(2) S ::= B
(3) A ::= aaA	(4) A ::= aa
(5) B ::= aaB	(6) B ::= a

is LR(1), but there is no fixed k for which it is LL(k). This is true because to recognize the string a^n, all the n symbols have to be scanned before a choice between production (1) and (2) can be made.

2.34 Precedence Parsers

Bottom-up parsing involves repeated application of the following two steps:

1. finding the handle, and
2. reducing it to the appropriate non-terminal.

Each of these two steps calls for repeated consultation of the productions of the grammar. If the number of productions is large, the repeated consultation of these productions for every iteration of the above steps will slow down the parsing process. Therefore from these production tables are derived which can be consulted more efficiently than the productions themselves. Precedence relations are examples of such tables.

We shall begin with the definition of precedence relations.

Definition: Let G = (V,T,P,S) be a cfg. $\lessdot$, $\doteq$ and $\gtrdot$ are three relations on V defined as follows:

1. $X \lessdot Y$ if there is a rule A ::= aXBb in P such that $B \Rightarrow^* Yy$
2. $X \doteq Y$ if there is a rule A ::= axyb in P
3. $X \gtrdot Y$ if y is a terminal and if there is a rule A ::= aByb in P, such that $B \Rightarrow^+ cx$, and $Y \Rightarrow^* ad$.

The grammar G is called (1,1) precedence (or Wirth-Weber precedence) grammar if the three relations $\lessdot$, $\doteq$, $\gtrdot$ are pairwise disjoint and G is e-free.

A (1,1) precedence grammar is called a simple precedence grammar if the right sides of productions are unique, that is, G does not have two productions

$$U_1 ::= x \qquad \text{and} \qquad U_2 ::= x$$

where $U_1 \neq U_2$.

Intuitively, (1,1) precedence relations indicate the left and the right end of handles on the input string. Suppose we are scanning the input string from left to right. The relation $\lessdot$ indicates the beginning of a handle; more explicitely, if $X \lessdot Y$ holds, Y is the left end of a handle. $\doteq$ indicates the continuation of a handle, and $\gtrdot$ indicates the right end of a handle. When none of these relations holds between the last symbol scanned and the next input symbol an error is indicated.

Many naturally occurring grammars are not precedence grammars, and in many cases rather awkward grammars result from an attempt to find a simple precedence grammar for the language at hand. We can obtain a larger class of grammars which can be parsed using precedence techniques by relaxing the restriction that the $\lessdot$ and $\doteq$ precedence relations be disjoint.

Definition: A (1,1) precedence grammar in which we do not require the relations $\lessdot$ and $\doteq$ to be disjoint is called a (1,1) weak precedence grammar.

There are deterministic languages that cannot be described by simple precedence grammars. Therefore we need what are called extended precedence grammars.

<u>Definition</u>: Let G = (V,T,D,S) be a cfg. Let x and y be two strings of lengths m and n respectively, that is, $|x| = m$ and $|y| = n$. Then we define the three (m,n) precedence relations as follows:

$x \lessdot y$ if there is a canonical sentential form.... xy....where the head symbol of y is the head of the handle.

$x \doteq y$ if there is a canonical sentential form ....xy....where the head of symbol y and the tail symbol of y are in the handle.

$x \gtrdot y$ if there is a canonical sentential form.... xy....where the tail symbol of x is the tail of the handle.

G is said to be <u>(m,n) precedence grammar</u> if it is e-free and if the three relations $\lessdot$, $\doteq$ and $\gtrdot$ are pairwise disjoint.

G is called weak <u>(m,n) precedence grammar</u> if it is e-free, and the relation $\gtrdot$ is disjoint from the relation $\doteq$ and the relation $\lessdot$. To illustrate precedence parsing, consider the language described by the grammar G:

1. E ::= E + T	2. E ::= T
3. T ::= T * F	4. T ::= F
5. F ::= (E)	6. F ::= a

The weak (1,1) precedence tables for G are given in Figure 2.9.

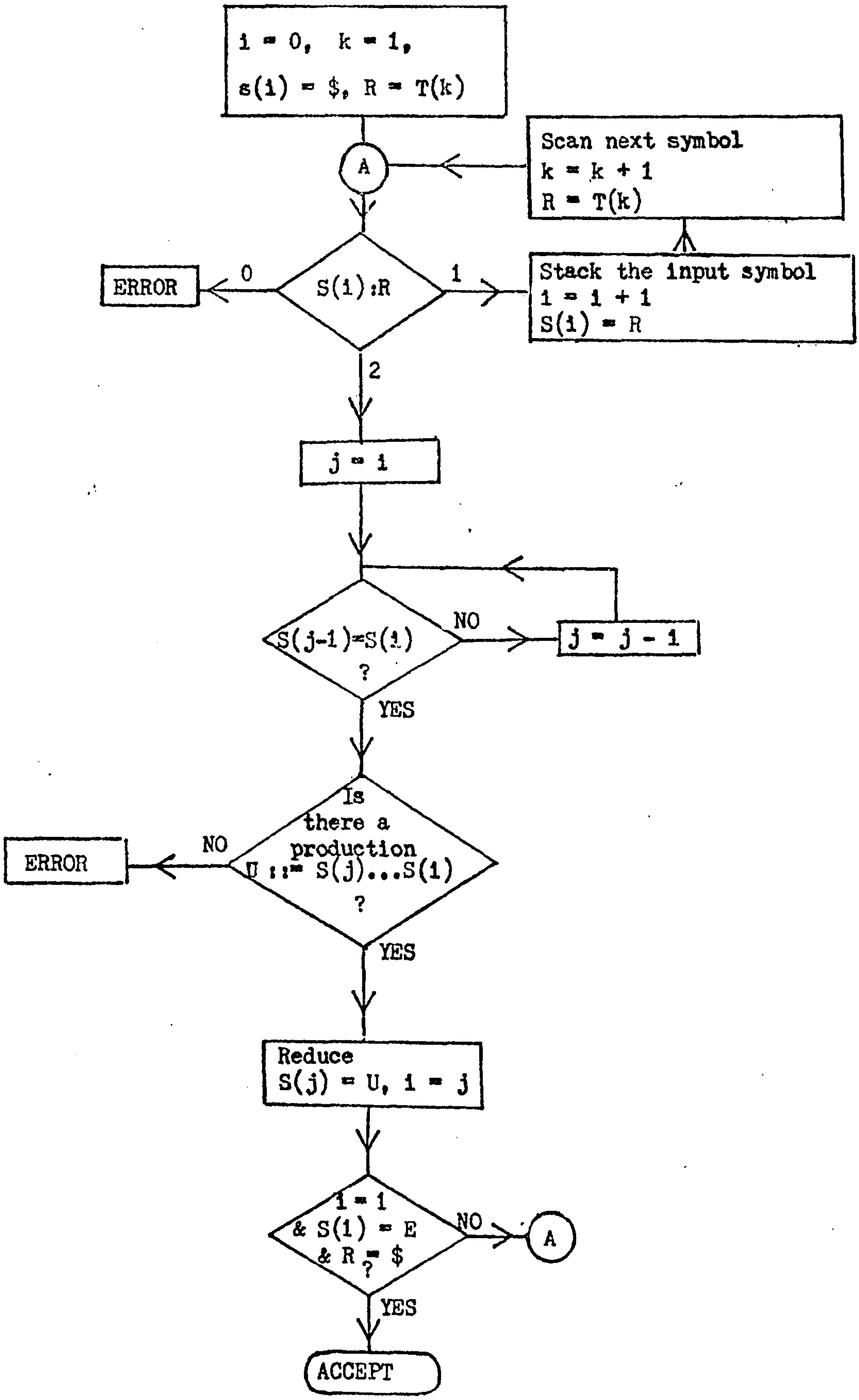


FIGURE 2.5
(1,1) Precedence Parser

	$	+	*	(	)	a
E	2	1	0	0	1	0
T	2	2	1	0	1	0
F	2	2	2	0	0	0
$	0	0	0	1	0	1
+	0	0	0	1	0	0
*	0	0	0	1	0	0
(	0	0	0	1	0	0
)	2	2	2	0	2	0
i	2	2	2	0	2	0

(1,1) precedence table for G.

1 represents $\lessdot$ or $\doteq$

2 represents $\gtrdot$

0 represents no relation

FIGURE 2.6

So far we have seen how precedence relations can be used to find handles in the input strings. The next step in bottom-up parsing is to find the correct non-terminal to which the handle reduces. To reduce the handle we need to find a production such that its right side matches the handle. This does not pose a problem if the right sides of all productions are unique. Grammars in which productions have unique right parts are called uniquely invertible (for short, UI). However, not all grammars are uniquely invertible. If a grammar is not uniquely invertible we need to look for some other property in the grammar which will enable us to perform the reduction of handles. The answer is given in the next section.

2.35 Bounded Context Parsers

The problem of reducing handles in the input strings to non-terminals is a trivial problem for uniquely invertible grammars. Theoretically, it is possible to convert a given cfg G into an equivalent cfg G' which is uniquely invertible. The following theorem and its proof are given in Graham (1970).

Theorem: Let $G = (V,T,P,S)$ be a context-free grammar.

a. There is an equivalent grammar G' which is uniquely invertible.

b. If G is LR(k) there is an equivalent LR(k) grammar G' which is uniquely invertible.

However, such transformations may leave the original grammar deformed badly. Besides, with the new grammar finding the handle may be considerably more inefficient than with the original grammar. Therefore it is necessary to find different ways of reducing handles for grammars that are not uniquely invertible.

Suppose x is a handle in a right sentential form and

$$U_1 ::= x \quad \text{and} \quad U_2 ::= x$$

are two productions such that $U_1 \neq U_2$. The left and right context of x in the sentential form may indicate which production to choose. The grammars in which we can tell which production to choose by looking at the context around the handle are the bounded context grammars. More formally:

Definition: A cfg $G = (V,T,P,S)$ is an (m,n) bounded right-context (BRC) grammar if the four conditions

1. $\$S\$ \underset{rm}{\Rightarrow}^* rAw \underset{rm}{\Rightarrow} rsw$, and
2. $\$S\$ \underset{rm}{\Rightarrow}^* tBx \underset{rm}{\Rightarrow} tux = r'st$

are rightmost derivations in grammar G,

3. $|x| \leq |y|$,
4. the last m symbols of r and r' coincide, and the first n symbols of w and y coincide

imply that $r'Ay = tBx$; that is, $r' = t$, $A = B$, and $y = x$.

A grammar is BRC if it is (m,n) BRC for some m and n.

The word "right" in bounded right-context is misleading. It is not the context that is "right"; rather, "right" refers to using rightmost derivations. Symmetrically, if we use leftmost derivations we have bounded left-context (or BLC) grammars. Further, in the definition of (m,n) BRC, if we replace "rightmost derivations" by "any derivations" we have (m,n) bounded context (or BC) grammars. Williams (1970) defines bounded parsable (BPC) grammars. BPC grammars are BC grammars in which a handle is not necessarily the leftmost simple phrase (i.e. reduction string). In his definition, a handle is any simple phrase.

2.36 Hierarchy of Grammars

The only difference between LR(n) parsers and (m,n) BRC parsers is that LR(n) parsers are allowed to look at any number of symbols on the stack (of the parsed symbols) whereas (m,n) BRC parsers are allowed to look at only a predetermined fixed number m of stack symbols. Therefore, every (m,n)BRC is LR(n). The converse is not true. The grammar G:

S ::= aA | bB
A ::= 0A | 1
B ::= 0B | 1

is an LR(0) grammar, but fails to be (m,n) BRC for any m and n.

However, the grammar G' :

$$S ::= aA \mid bA$$
$$A ::= 0A \mid 1$$

generates the same language and is (0,0)BRC. In general, given any deterministic language L There is a (1,1) BRC grammar to describe it.

Several investigators have worked on the transformation of deterministic context-free grammars to precedence grammars. Fisher (1969) proved:

> Simple precedence languages form a proper subclass of deterministic language.

$$L = \{ a0^n1^n \mid n \geq 1 \} \cup \{ b0^n1^{2n} \mid n \geq 1 \}$$

is a deterministic language that is not simple precedence.

However, any deterministic language can be described by a weak (1,1) precedence (not necessarily UI) grammar. Also, Graham (1970) established that

> Every deterministic language is generated by a UI (2,1) precedence grammar.

Figure 2.7 shows a hierarchy of deterministic grammars.

2.37 Mixed Strategy Parsers

BRC parsers do not make efficient parsers by themselves (Section 5.4.2, Aho and Ullman, 1972). Precedence relations speed up the task of finding handles, but some context around the handle may be needed to reduce the handle. Therefore it is efficient to have a parser that uses a (p,q) precedence relation to find the handle and (m,n) context to reduce the

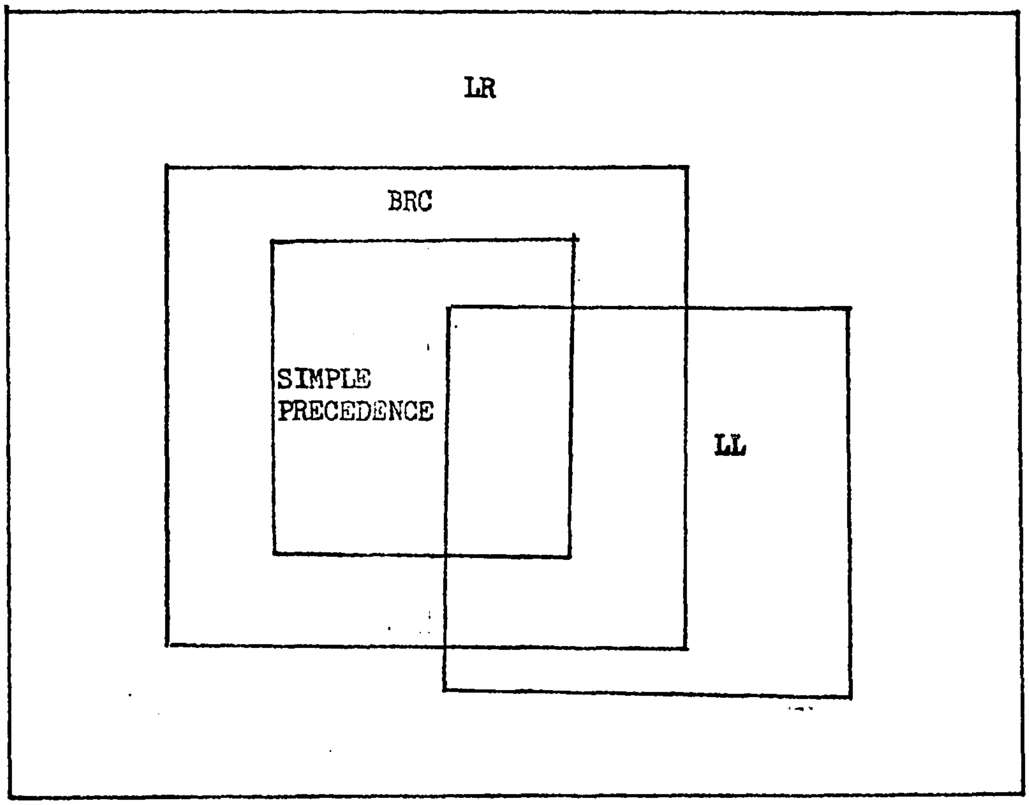


Figure 2.7 Hierarchy of Deterministic Grammars.

handle. Such a parser is called a (p,q;m,n) mixed-strategy parser (MSP). McKeeman (1966) introduced the MSP's. He gave the name stacking-decision function (C1) to the function that finds the handle and the name production selection function (2) to the one that reduces the handle. The domain and the range of the functions C1 and C2 are as follows. If αx is a canonical sentential form, where α is the part that is partially reduced and x is the unscanned part of the input string, the domain of C1 consists of pairs of the form $(t_p(\alpha), h_q(x))$. t_p stands for $TAIL_p$ and h_q stands for $HEAD_q$. The values of the C1 function are

$$
\begin{aligned}
C1\ (t_p(\alpha), h_q(x)) &= 0, && \text{when no relation holds between } t_p(\alpha) \text{ and } h_q(x)\\
&= 1, && t_p(\alpha) \lessdot h_q(x)\\
&= 2, && t_p(\alpha) \gtrdot h_q(x).
\end{aligned}
$$

The function C2 is called whenever the value of C1 is 2. The domain of C2 consists of triples of the form (l,h,r), where h is the handle, and l and r are the left and right context (of length p and q respectively) of h in the sentential form. The values of C2 are

$$
\begin{aligned}
C2(l,h,r) &= 0, && \text{the handle cannot be reduced}\\
&= p, && p>0, \text{ pth production is used in the reduction.}
\end{aligned}
$$

2.4 SEMANTIC ANALYSIS

In the syntactic analysis phase, it is decided if the given input program is an acceptable program; in the semantic analysis the accepted program is translated into a target language. During the process of translating the source program into target code the semantic analyzer constructs a table of symbols used in the source program. Some of the functions of symbol table construction of the semantic analyzer of a conventional programming language are:

1. Label identifiers are defined when they appear as labels of statements; the label and its (relative) position should be stored in the symbol table. Any forward references to this label should be marked appropriately.
2. Identifiers should be declared (explicitly or implicitly) once, and only once; their attributes (by declaration or by default) should be entered into the symbol table.

3. The arguments of function calls must be compatible both in number and in attributes with the definition of the function. At the time of definition, functions, along with their arguments, should be entered into the symbol table.

The significance of the symbol table to our error correction algorithm is as follows. After a candidate is selected for correction it is first checked for semantic correctness. The semantic analyzer is called to check the semantic correctness. If the semantic analysis of the candidate for correction introduces conflicts with the information in the symbol table constructed to that point the semantic analyzer announces a semantic error and the candidate for correction is rejected.

CHAPTER THREE
AN ERROR CORRECTION ALGORITHM

This chapter describes our algorithm for correcting syntax errors. Semantic errors are also considered, but correcting semantic errors is not the goal of this thesis. Semantic errors committed by the programmer and found in syntactically correct statements do not influence our algorithm. However, semantic errors matter in the following way:

> if an attempt to correct a syntax error introduces semantic errors then the correction is rejected.

Section 3.1 describes detection of errors. First, the terms "error correction" and "error recovery" are distinguished. Then the capabilities of different parsers to detect errors early in the string are discussed. Causes for the delay in detecting errors are also given. The two types of errors detected are considered in Section 3.12. Section 3.2 is the core of this thesis. Section 3.21 contains the rules which define the set of all correction strings for a given string. In Section 3.22 the ideal conditions required by our error correction algorithm are given. A method of finding the correction strings is presented in Section 3.23. A method for checking syntactic and semantic context is described in Section 3.24. Section 3.25 explains how the final correction, after screening out all the unwanted correction strings, is selected. In the process of screening out invalid correction strings, if no strings are left, the parser is moved backwards and the correction process is repeated. Section 3.26 illustrates how the backing of the parser works.

3.1 DETECTION OF ERRORS

3.11 Error Correction and Error Recovery

When the parser (a DPDA) enters the state "ERROR" we say a (syntax) error has been detected. If the parser halts after the detection of the first syntax error the remainder of the program remains unparsed and the programmer is not given any information about the rest of the syntax errors. This means that for each syntax error the programmer must resubmit his program. Therefore, the parser must find a way to get out of the state ERROR. There are two actions the parser can take:

1. Make appropriate changes in the input tape.
2. Make changes in the internal status (pushdown list) of the DPDA and in the input tape.

Action 1. is error correction and action 2. is error recovery. In our algorithm both error correction and error recovery are used. Correction is tried first. If it does not succeed, we resort to recovery action which is rather simple-minded but will always succeed.

3.12 Delay in Detecting Errors

The performance of the error correction algorithm depends very much on how early the error is detected.

A delay in the detection of an error may make error correction very difficult, and sometimes even impossible. The delay in detecting errors is caused by

1. limited left context
2. misinterpretation of the string.

Limited left context: If a language has an LR parser as well as a precedence parser, the precedence parser is usually faster than the LR parser. However, the precedence parser will not detect the errors as early as the LR parser. LR parsers, in general, detect an error at the earliest possible opportunity in a left-to-right scan of input string. LL parsers enjoy the fast speed and share the good error detecting capability of LR parsers at the same time. However, not every deterministic language has a LL grammar, and, in general, it is often possible to find a more "natural" LR grammar to describe a programming language and its translation.

Suppose that statement (1) was written when (2) was intended:

(1) A / B + C ;
(2) A = B + C ;

Assume the language under consideration is a subset of PL/1.* A LR(1) parser will find the error just after it stacks A on the pushdown list and scans the symbol /. However,

---

*PL/1 itself cannot be parsed by a (1,1) precedence parser. Therefore, a subset must be assumed.

```
            A   /   B   +   C   ;
                ↑               ↑
     LR(1) parser       (1,1) precedence
     detects error      parser detects
                        error
```

a (1,1) precedence parser will not detect the error until it scans the statement delimiter. A (2,1) precedence parser, of course, would have detected the error at the same time LR(1) did. Intuitively, the reason for the inability of the (1,1) precedence parser to detect the error early is the lack of sufficient left context. At the time the (1,1) precedence parser is scanning the symbol / and making a decision as to what should be done with it, it can look only at A (which is on the top of the pushdown list). Therefore, the (1,1) precedence parser cannot distinguish the two contexts of A / B + C

...; X = A / B + C ;...

and ...; A / B + C ;...

In general, suppose

$$\$ \quad S_0\ S_1\ S_2 \ldots S_n\ t_k\ t_{k+1} \ldots \quad \$$$

is a right sentential form. S_n is the top symbol on the pushdown list and t_k is the next input symbol. Let S_i, $i \leq n$, be the head symbol of the handle. A (p,1) precedence parser is allowed to look at p symbols S_{n-p+1}, S_{n-p+2}, ... , S_n from the top of the pushdown list. If $i < n - p + 1$, that is, if the parser is unable to look at the head of the handle, the parser may not be able to detect if t_k is an illegal symbol.

Now we shall consider the second reason for delay in the detection of errors. If the head of an erroneous string* x, by accident, happens to be the head of a correct string y in the language, the parser may be misled and a delay caused in detecting the error. Actually, as far as the parser is concerned, there was no delay, but according to the error correcting algorithm the string y may not be derivable from the string x. For example, in statement (3) (in PL/1)

```
(3)  A ( B + 1 ;
              ↑
```

the earliest place any parser with a left-to-right scanner can detect the error is the ; (semicolon). Statement (3) appears to be a head of statements of the form

```
(4)  A (B + 1 ...) = ... ;
```

Any statement of the form (4) cannot be derived from the statement (3) by using the transformation Rule (3.21) (to be given in Section 3.21). The shortest string of the form (4) is:

```
A (B + 1) = 0 ;
```

and cannot be derived from string (3) using the transformation (3.21).

Errors in statements that mislead a left-to-right parser into believing that the erroneous statement is a substring of another correct string are not rare. The following are a few examples of such errors in PL/1 statements:

---

*The string x is not all of the string from the beginning of the input; it is the string of terminals around the point of detection of the error corresponding to a substructure (substructure is defined in Section 3.22)

Incorrect Statement	Intended Statement	Another Possible Interpretation
X, Y, A(0) ;	X, Y = A(0) ;	X, Y, A(0) = 0 ;
A : B(C + 1) ;	A = B(C + 1) ;	A: B(C + 1) = 0 ;
A (B + 1 ;	A = B + 1 ;	A (B + 1) = 0 ;
declar (A,B) fixed;	declare (A,B) fixed;	declar (A,B) = fixed;
do i = 0 ;	do; i = 0;	do i = 0 to 1 ;

3.13 Types of Errors

When the parser enters the state ERROR we know that a syntax error has been detected. We categorize syntax errors into two classes:

1. Action error,
2. Reduction error.

In bottom-up parsing, "action error" corresponds to the case when the top symbol on the pushdown list and the next input symbol form an illegal pair. In LR(k) tables and precedence tables we have represented such cases (in Chapter 2) by a 0 (zero). In top-down parsing "action error" corresponds to the case when the top of the pushdown list is a terminal different from the next input symbol.

In bottom-up parsing, "reduction error" means the tail of the handle is reached but the handle cannot be successfully reduced to a non-terminal. In top-down (LL(k)) parsing "reduction error" means that the top of the pushdown list is a non-terminal A such that $FIRST_k(A)$ does not contain the string $HEAD_k(T)$, where T is the remaining input string.

3.2 CORRECTION OF ERRORS

A parser of a language recognizes correct strings in that language. An error-correcting parser recognizes the correct strings in the language, and in addition, recognizes certain strings called correctable strings. A precise description of correctable strings follows.

3.21 Correctable Strings

Let G = (V,T,P,S) be a deterministic cfg and let D be a parser for the language L = L(G). A PDA D' is said to be an error correcting parser for L if D' recognizes a set L' such that

1. L' is a subset of T*
2. L is a subset of L' , and
3. for string s' in L' there is a corresponding string s in L such that s and s' differ by at most one symbol, or in a permutation of two adjacent symbols.

The strings s' of L' are called the strings correctable by D'. When D' is understood, we shall just say s' is a correctable string. A string s in L that corresponds to string s' in L' is called a correction of s', written s = C(s').

Now we shall define the rule by which an incorrect string w is transformed to a correct string s. It should be noted that the string w does not consist of the terminal symbols the programmer wrote, but rather the tokens corresponding to them. The lexical analyzer converts the input terminals into tokens and passes them to the syntactic analyzer. A correctable string w is transformed to a correct string s by the following rule:

w_1, w_2, w_3, s_1, s_2 and s_3 are strings of tokens such that

(3.21)

1) $w = w_1 w_2 w_3$ and $s = s_1 s_2 s_3$,
2) $w_1 = s_1$, $w_3 = s_3$ and $w_2 = ab$
3) s_2 is one of the following:
cab, cb, b or ba
where a, b, and c are tokens.

In the future we shall refer to this rule as Rule (3.21).

3.22 A Model for Error Correction

No correction algorithm can correct all errors without being prohibitively expensive. Therefore, we do not attempt to correct all errors. The language and errors are required to satisfy certain conditions for our algorithm to be effective.

First, it is assumed that we can define a substructure in the language such that the programs in the language can be expressed in the form:

$ substructure substructure ... substructure $

where $ indicates the beginning or the end of program. In languages FORTRAN and BASIC this condition is easily satisfied. FORTRAN statements and BASIC statements are the required substructures. In block-structured languages like ALGOL and PL/1 we must be satisfied with less than ideal conditions. A substructure in the language is chosen so that:

1. the number of correctable errors is maximized, while
2. the average time spent on each error is minimized.

Block statement, compound statement and simple statement are possible candidates for the substructure. If the simple statement is chosen as substructure, the errors found while parsing a simple statement, but not local to the simple statement, may not be correctable. Among block statement, compound statement and simple statement the choice of simple statement will have the most global errors that will be immune to correction. An example will illustrate the problem. Suppose a programmer wrote (in PL/1)

```
if  B  then
            TEMP = X ;
            X = Y ;
            Y = TEMP ;
        end;
```

when he actually meant to write •

```
if  B  then
            do;
                TEMP = X ;
                X = Y ;
                Y = TEMP ;
            end;
```

The error is detected when the simple statement "end;" is being parsed. If the correction is confined to the simple statement substructure, the error cannot be corrected. However, if the correction is confined to the compound statement or block substructure, the amount of time spent on each error becomes prohibitively large.

Next, the correction strings are obtained by applying Rule (3.21). This implies that an erroneous string which cannot be transformed to a correct string by Rule (3.21) cannot be corrected. In most compilers the lexical analyzer removes the spaces between the terminal symbols and passes the tokens to the syntactic analyzer. Therefore, errors caused by

1. misplaced space(s), and
2. missing space

are not always correctable by our algorithm. For example, the string

```
dec lare  A  fixed ;
```

is not correctable, since it is tranformed to the tokens

T*(<identifier>) T(<identifier>) T(<identifier>) T(fixed) T(;)

which cannot be transformed to a correct string of tokens by Rule (3.21). For the same reason, the string

```
declareA  fixed ;
```

is not correctable. The lexical analyzer is better suited for correcting such errors.

3.23 Generating Correction Strings

After detecting the existence of error, the next step in the error correction algorithm is to generate the correction strings. Consider the incorrect statement (in PL/1):

(1) A = B C ;

The following are among the correction strings generated by Rule (3.21):

(2) A = B + C ;
(3) A = B - C ;
(4) A = B * C ;
(5) A = B / C ;
(6) A = B mod C ;

---

* T(symbol) means the token of the "symbol".

(7) A = B = C ;
(8) A = B < C ;
(9) A = B > C ;
(10) A = B | C ;
(11) A = B & C ;
(12) A = B || C ;
(13) A = B ;
(14) A = C ;

Obviously, all these strings need not be generated. In other words, there are groups of strings that can be represented by one string. For example, from (2) and (3) only (2) need be considered. From (4), (5) and (6) only (4) can be considered. This introduces the notion of syntactical equivalence. Two strings of terminals

$$s = S_1 S_2 \ldots S_k \qquad \text{and} \qquad t = T_1 T_2 \ldots T_m$$

are said to be <u>syntactically equivalent</u> if $k = m$ and the terminals S_i and T_i are syntactically equivalent for each i. Next, we shall define syntactical equivalence of terminals.

<u>Definition</u>: Let $G = (V,T,P,S)$ be a cfg. Two symbols (terminal or non-terminal) t_1 and t_2 are said to be <u>syntactically equivalent</u> if two new productions

$$\langle \text{NEW} \rangle ::= t_1$$
$$\langle \text{NEW} \rangle ::= t_2$$

can be such that:

if $P' \subseteq P$ is the set of productions containing either t_1 or t_2 or both, then if each occurrence of t_1 and t_2 is replaced by <NEW> then all the productions of P' become identical.

If t_1 and t_2 are terminals then <NEW> is called a pseudo-terminal.

In XPL, for example, the terminals *, / and mod are syntactically equivalent, since the productions

<term> ::= <term> * <primary>
<term> ::= <term> / <primary>
<term> ::= <term> mod <primary>

become identical when *, / and mod are replaced by a pseudo-terminal. Consequently, strings (4), (5) and (6) are syntactically equivalent. Similarly, strings (2) and (3) are equivalent and so are strings (8) and (9). Thus we can represent strings (2) through (9) by the following strings:

(2)' A = B + C ;
(4)' A = B * C ;
(7)' A = B = C ;
(8)' A = B < C ;

Intuitively, strings (10) and (11) appear equivalent. However, our definition of syntactical equivalence does not make them syntactically equivalent. Therefore, we need to modify our definition of syntactical equivalence.

Let G = (V,T,P,S) be a cfg. Two symbols t_1 and t_2 are said to be <u>essentially</u> <u>equivalent</u> if two new productions

$$\langle NEW \rangle ::= t_1$$
$$\langle NEW \rangle ::= t_2$$

can be introduced such that:

if $P' \subseteq P$ is the set of productions

$$\langle N_1 \rangle ::= \alpha_1$$
$$\langle N_2 \rangle ::= \alpha_2$$
$$\cdots$$
$$\cdots$$
$$\cdots$$
$$\langle N_k \rangle ::= \alpha_k$$

containing either t_1 or t_2, then after replacing each occurrence of t_1 and t_2 in each α_i by <NEW> and performing the reductions (if possible) α_i to β_i, where β_i is a string such that,

$$\beta_i \Rightarrow^* \alpha_i$$

the strings β_1, $\beta_2 \ldots \beta_k$ become identical. For example, in XPL we can define

<NEW> ::= |
<NEW> ::= &
<NEW> ::= *

and change the productions (See Appendix B)

<expression> ::= <expression> <logical factor>
<logical factor> ::= <logical factor> <logical secondary>
<term> ::= <term> * <primary>

to

<expression> ::= <expression> <NEW> <logical factor>
<logical factor> ::= <logical factor> <NEW> <logical secondary>
<term> ::= <term> <NEW> <primary>

Then the three right parts.

<expression> <NEW> <logical factor>
<logical factor> <NEW> <logical secondary>
<term> <NEW> <primary>

reduce to

<expression> .

This makes the strings (1)', (10) and (11) equivalent. Also, strings (2)' and (4)' can be made equivalent by the new definition. However, the terminal "+" and the terminal "*" are not always equivalent. For XPL (as some other programming languages) uses + (and -) for two purposes, viz. unary + and binary +. One solution is to use a distinct notation, say $+_1$ for unary +. The other solution is to distinguish them by means of context. For example, + in the left context of <arithmetic expression> , <term> , <primary> , <variable> , <identifier> , and <constant> is binary; but + in the left context of (, IF, =, TO and BY is unary. The same problem arises with the terminal "=".

The symbol "=" in the left context of ; <variable> is the assignment operator. In all other legal left contexts it is the relational operator.

Now we shall consider the actual problem of generating the correction strings. Let $t_0 t_1 \ldots t_n$ be a string of terminals containing an error such that

$$\text{substructure} \Rightarrow^* C(t_0 t_1 \ldots t_n)$$

where $C(t_0, \ldots, t_n)$ is a correction string of the string $t_0 t_1 \ldots t_n$. Let t_k, $k \leq n$, be the next input symbol at the time the error was detected. We shall call t_k the <u>position being corrected</u>. When after an error is detected

(1) $t_0 t_1 \ldots \underset{\uparrow}{t_k} \ldots t_n$

the error flag is set and the parser enters the error-correction mode. Then the correction strings of string (1) are generated. According to Rule (2.31) the following is the set of correction strings:

Strings obtained by inserting a terminal s in front of t_k.

(2) $t_0 t_1 \ldots t_{k-1} s t_k \ldots t_n$

Strings obtained by replacing t_k by a terminal s.

(3) $t_0 t_1 \ldots t_{k-1} s t_{k+1} \ldots t_n$

The string obtained by deleting t_k.

(4) $t_0 t_1 \ldots t_{k-1} t_k \ldots t_n$

The string obtained by interchanging the place of t_k and t_{k+1}.

(5) $t_0 t_1 \ldots t_{k-1} t_{k+1} t_k \ldots t_n$

If a language has NT terminals this will generate 2 * NT + 2 correction strings. Testing each one of these strings by parsing will be time consuming. Therefore, the following means are used to screen out the obviously ineligible or unwanted candidates.

a. If s_1 and s_2 are two syntactically equivalent terminals then the two strings

$$t_0 t_1 \ldots t_{k-1} s_1 t_k \ldots t_n \quad \text{and} \quad t_0 t_1 \ldots t_{k+1} s_2 t_k \ldots t_n$$

formed by substituting s_1 and s_2 for s in (2) are syntactically equivalent. Of syntactically equivalent strings only one needs to be generated. The same argument holds for the set of strings (3).

b. The string

$$t_0 t_1 \ldots t_{k-1} s' t_k \ldots t_n$$

generated by inserting s' in front of t_k need not be considered if s' does not satisfy the (1,1) context, viz., (t_{k-1}, t_k). For LR, LL and precedence parsers checking of (1,1) context can be done by a quick look at the parsing tables. The same argument holds for the strings generated by replacing t_k by a terminal s_1 which has an invalid (1,1) context.

3.24 Testing the Correction Strings

The correction strings that are generated are checked for

1) syntactic context in the substructure
2) semantic context in the part of the program that is already parsed.

If the parser announces an error while scanning the terminal t_k the syntactic context of the error is the string of terminals (tokens)

$$t_0 t_1 \ldots t_k \ldots t_n$$

such that

$$\text{substructure} \Rightarrow^* C(t_0 t_1 \ldots t_k \ldots t_n)$$

The terminals $t_0, \ldots, t_{k-1}$ are already parsed. The terminals $t_{k+1}, \ldots, t_n$ are to be read in advance and stored. The question arises about how many terminals are to be read in advance, i.e. what is n? It is assumed that the language uses one or more terminals to delimit the substructure; t_n should be the first such terminal that is beyond t_k. It is possible that the programmer has omitted a delimiter or misplaced one. If a delimiter is omitted the terminals up to the next delimiter are read in advance. If omission of the delimiter is the only error among the terminals $t_0, t_1, \ldots, t_n$, then it will be corrected. A misplaced delimiter t_n will cause the failure of the reduction

$$\text{substructure} \Rightarrow^* C(t_0 t_1 \ldots t_n).$$

For example, in our implementation of XPL ";" and "THEN" are considered to be "statement" delimiters. In the statement

$$\text{if} \quad X \neg Y \quad \text{then} \quad X = 0 \;;$$

the error (a missing =, $<$ or $>$) is detected while scanning the symbol "Y". The isolated string is: " if X¬Y then ". In the statement

$$X = Y \quad \underset{\uparrow}{Y} = \text{TEMP} \;;$$

the semicolon is missing. Therefore, the symbols up to the next semicolon are read in advance. In the following example

declare (A,B;C) fixed;

a semicolon is written in place of a comma. The isolated string is "declare (A,B;" which cannot be corrected by our algorithm.

A correction string $C(t_0 \ldots t_n)$ is said to satisfy the syntactic context of the substructure if the parsing of the string $C(t_0 \ldots t_n)$ continues without any syntactic errors resulting during the reduction

$$\text{substructure} \Rightarrow * \; C(t_0 t_1 \ldots t_n) \;,$$

and at the end of reductions the contents of the pushdown list indicates the correct parsing of a substructure.

The LL and LR parsers can access all the contents of the pushdown list. Hence, the parsing up to the delimiter t_n without any syntactic error is sufficient to ensure the proper configuration of the pushdown list. However, this is not the case with precedence parsers. For example, consider the incorrect statement (in XPL) if B the X = B ;

("the" is a mispunch for "then".) One of the correction strings is

if B the: X = B ;

A parser using a (1,1) precedence relation will parse this without noticing the syntactic error. The top of the push-down list will have the form:

<expression>
<replace>
<variable>
<label definition>
<expression>
if

One more reduction will reduce it to

<statement list>
<expression>
if

Since the parser checks only the top symbol, it will not notice the presence of "if" and <expression>. Therefore, special checking is necessary to ensure the correct configuration of the pushdown list.

The compiler designer, at the time of implementing our algorithm forms a list of symbols which can appear on the parse stack after the complete parsing of a string satisfying the syntax of "statement" (substructure). For XPL this is:

<statement> , <statement list> ,
<basic statement> , <if statement>,
<group head> , <procedure head> and ";"

If the parsing of a correction string $C(t_0...t_n)$ continues without any syntactic errors then it undergoes the special checking which consists of examining the parse stack for any symbols not listed above. If a symbol not listed above is found on the parse stack the correction string $C(t_0...t_n)$ is considered to have failed the syntactic context of the substructure and is rejected.

After a correction string satisfies the syntactic context in the substructure, the next step is to check its semantic compatibility with the part of the program already compiled.

Semantic compatibility is checked by calling the semantic analyzer. The semantic analyzer may find that the semantic information of the correction string $C(t_0t_1...t_n)$ creates conflicts with the semantic table constructed from the semantic analysis of the part of the program already compiled. In this case the correction string $C(t_0...t_n)$ is rejected. For example, consider the incorrect statement (in XPL)

A : B (C + 1) + 1 ;

A correction string for this is

A : B (C + 1) = 1 ;

This satisfies the syntactic context, but if A was declared to be any identifier type other than label, the semantic analyzer announces error and the correction string is rejected. In such a case the alternate correction string

$$A = B\ (C + 1) + 1\ ;$$

is accepted.

3.25 Correction Decision

Each generated correction string that satisfies the syntactic context and the semantic context is saved for the final selection. After all the correction strings are generated and tested the number of correction strings that are saved is computed. This number is used to make the decision about the next step to be taken in the error correction process. The following are the possibilities:

1. several correction strings saved,
2. exactly one correction string saved,
3. no correction strings saved.

We shall discuss how the decisions are made in these cases taking them one by one.

1. Several correction strings: When there is more than one correction string satisfying the syntactic and semantic context a scheme is needed to find the "best" one. The best criteria for such a scheme are language dependent. The following is a scheme used in our implementation for XPL which can be modified for other languages by including additional language-dependent criteria.

a. If the correction string $t_0 \ldots t_{k-1} t_k \ldots t_n$ obtained by interchanging t_k and t_{k+1} is one of the correction strings saved, choose it.

b. Otherwise, if the correction string $t_0 \ldots t_{k-1} t_{k+1} \ldots t_n$ (obtained by deleting t_k) is one of the correction strings saved, choose it.

c. Otherwise, if there is any correction string of the form $t_0 \ldots t_{k-1} s' t_k \ldots t_n$ (obtained by inserting a terminal s' in front of t_k), choose it.

d. Otherwise, choose the first one.

The justification for the above criteria is intuition and our experience with error correction for XPL. If the interchange of two consecutive symbols is one of the corrections saved, then most likely it is the "best" correction. The choice among b, c and d, however, was not so definite. There were examples suggesting the highest priority for the selection criterion b, but there were also examples that suggested the contrary.

2. <u>Exactly one correction string</u>: In this case the programmer is informed about the correction action taken, the error flag is reset and the parser enters the standard mode.

In either of the above two cases the correction string finally selected is very unlikely to be an undesirable one since it undergoes stringent tests.

3. <u>No correction strings</u>: This can happen for two reasons. First, the point of error may be left of the point of error detection. In this case, the parser is moved one step backwards (to be explained in the next section) and the error correction process is tried one symbol left of the position where error correction was being attempted previously.

This is repeated until one or more correction strings are found, or the backward move reaches the left end of the substructure. Second, if the left end of the substructure is reached the error correction algorithm has failed to correct the error(s) found in the substructure. A recovery action is taken, viz. delete the string

$$t_0 t_1 \ldots t_k \ldots t_n$$

comprising the substructure where the error was being corrected. This includes the terminals $t_0, \ldots, t_k$ that were scanned before the error was detected but after the delimiter for the previous substructure was scanned. Also, it includes the terminals $t_{k+1}, \ldots, t_n$ that were scanned in advance where t_n is the next delimiter.

3.26 Backing Up the Parser

We shall explain the backward move of the parser by an example. Consider the language L given by LL(1) grammar G:

1. S ::= $ V R $
2. R ::= =E
3. E ::= T E'
4. E' ::= e
5. E' ::= + T E'
6. T ::= F T'
7. T' ::= e
8. T' ::= * F T'
9. F ::= V

A. F ::= (E)

B. V ::= I V'

C. V' ::= e

D. V' ::= (E)

The language given in Section 2.43 is a subset of L. The Grammar G given above contains a few more productions in addition to the productions of the grammar of Section 2.43.

Suppose that we are parsing the string (scanning it from left to right)

(1) $ I (I + I $
↑

One method of correcting the error is the following. Top-down parsing is goal-oriented in the sense that the parser, on its pushdown list, has information about what it is expecting rather than what it has already parsed. With the information on the pushdown list and the parsing tables one can decide (for details see Irons, 1963) what strings of terminals are expected at the point of error. Hence, at the point of error the string of unwanted symbols can be replaced by the (shortest) correct string. With this method, a correction for the above string is

$ I (I + I) = I $
string inserted

As far as the parser is concerned, this is a "natural" correction. However, it is not very likely that a human (assuming he knows the language well enough) omits three symbols at once. It is more probable that a human omits or misplaces one symbol rather than a string of several symbols. Therefore, we consider

$ I = I + I $

as a "natural" correction of string (1).

In order to find this correction, the parser will have to move backwards. In the next chapter the backward move is illustrated by an example using bottom-up parsing. Therefore, to avoid duplication, an example in top-down parsing is used here.

The parsing of the string $ I (+ I $ is given by the following sequence of moves.*

M1	[I(I+I$, VR$, 1]	$\vdash$	[I(I+I$, IV'R$, 1B]
M2		$\overset{p}{\vdash}$	[(I+I$, V'R$, 1B]
M3		$\vdash$	[(I+I$, (E)R$, 1BD]
M4		$\overset{p}{\vdash}$	[I+I$, E)R$, 1BD]
M5		$\vdash$	[I+I$, TE')R$, 1BD3]
M6		$\vdash$	[I+I$, FT'E')R$, 1BD36]
M7		$\vdash$	[I+I$, VT'E')R$, 1BD369]
M8		$\vdash$	[I+I$, IV'T'E')R$, 1BD369B]
M9		$\overset{p}{\vdash}$	[+I$, V'T'E')R$, 1BD369B]
M10		$\vdash$	[+I$, T'E')R$, 1BD369BC]
M11		$\vdash$	[+I$, E')R$, 1BD369BC7]
M12		$\vdash$	[+I$, +TE')R$, 1BD369BC75]
M13		$\overset{p}{\vdash}$	[I$, TE')R$, 1BD369BC75]
M14		$\vdash$	[I$, FT'E')R$, 1BD369BC756]

---

* A configuration of the parser is $[w, \alpha, \pi]$ where w is the remainder of the input string, α is the contents of the pushdown list, and π is the sequence of production numbers used in the parsing. A move in which a terminal t is popped is indicated by

$$[tw,\ t\alpha, \pi] \overset{p}{\vdash} [w, \alpha, \pi].$$

An e-move is indicated by

$$[w,\ Z\alpha, \pi] \vdash [w,\ \gamma, \pi p]$$

where Z is a non-terminal, and γ is a string of terminals and non-terminals.

M15	$\vdash$	[I$, VT'E')R$, 1BD369BC7569]
M16	$\vdash$	[I$, IV'T'E')R$, 1BD369BC7569B]
M17	$\vdash^{P}$	[$, V'T'E')R$, 1BD369BC7569B]
M18	$\vdash$	[$,)R$, 1BD369BC7569BC74]

At this point the parser announces an action error because the terminal $ on the input does not match the symbol) on the pushdown list.

An attempt to

a) insert a terminal in front of the symbol $
b) replace the symbol $ by a terminal
c) delete the symbol $

will not correct the error.

Therefore, we back up the parser to the step when it was about to scan the previous terminal, viz. I, in the input string. M13 is the required step. At M13, the parser observes the input symbol I the first time, and moves accordingly. Instead, we

a) insert a terminal before I
b) replace I by another terminal,
c) delete I,
d) interchange I and $,

and allow the parser to continue. None of these attempts allow the parser to continue without an error. Therefore, we back up the parser again. This time, we set the parser back at M9. Again, our attempts at correction fail. The process of backing up continues and finally we arrive at move M2. Here, the correction

replace the input symbol (by the symbol =

succeeds and the parser continues as given below.

```
M2'   [I = I+I$, IV'R$, 1B] ⊢p [=I+I, V'R$, 1B]
M3'                         ⊢  [=I+I$, R$, 1BC]
M4'                         ⊢  [=I+I$, =E$, 1BC2]
M5'                         ⊢p [I+I$, E$, 1BC2]
M6'                         ⊢  [I+I$, TE'$, 1 BC23]
M7'                         ⊢  [I+I$, FT'E'$, 1BC236]
M8'                         ⊢  [I+I$, VT'E'4, 1BC2369]
M9'                         ⊢  [I+I$, IV'T'E'$, 1BC2369B]
M10'                        ⊢p [+I$, V'T'E'$, 1BC2369B]
M11'                        ⊢  [+I$, T'E'$, 1BC2369BC]
M12'                        ⊢  [+I$, E'$, 1BC2369BC]
M13'                        ⊢  [+I$, +TE'$, 1BC2369BC5]
M14'                        ⊢  [I$, TE'$, 1BC2369BC5]
M15'                        ⊢  [I$, FT'E'$, 1BC2369BC56]
M16'                        ⊢  [I$, VT'E'$, 1BC2369BC569]
M17'                        ⊢  [I$, IV'T'E'$, 1BC2369BC569B]
M18'                        ⊢  [$, V'T'E'$, 1BC2369BC569B]
M19'                        ⊢  [$, $, 1BC2369BC569BC74]
```

CHAPTER FOUR
IMPLEMENTATION OF THE ALGORITHM

This chapter describes an implementation of the error correction altorithm given in the last chapter. The basic principle of the algorithm can be implemented in almost any kind of deterministic parser. After the parser detects the existence of an error a string of terminals around the position where the existence of the error was detected is isolated. The string of terminals isolated presumably corresponds to a substructure (e.g. "statement") in the language. The algorithm requires that there be at most one error in this string. Also, for the algorithm to successfully correct the input string, it must differ from the intended string by one terminal symbol or by a single permutation of two adjacent terminals. However, the success of the algorithm does not depend on how soon the error is detected, as long as it is detected in the same substructure as it appears. In Chapter Two it was explained why precedence parsers do not, in general, detect errors as early as the LR and LL parsers. Since our algorithm does not rely on the early error detection capability of the parser, it can be implemented with a precedence parser without difficulty. In fact, the compiler used in our implementation parses bottom-up using precedence relations.

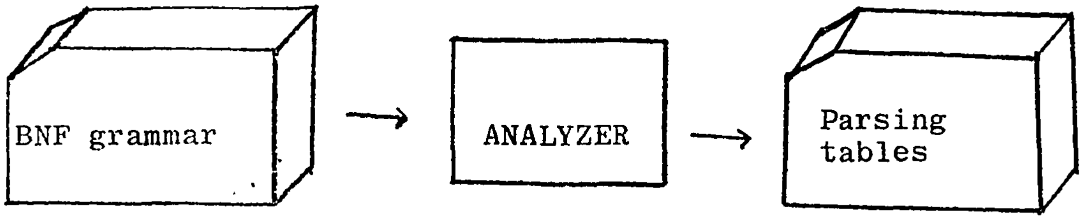
The language chosen for implementing our algorithm is XPL. XPL is a subset of PL/1 introduced by McKeeman et al. (1970). An XPL compiler for the implementation was built from SKELETON. SKELETON is a protocompiler* (ibid.) which is briefly described in the next section. Dection and location of errors are discussed in Section 4.2. Section 4.3 describes generation of the correction strings. Testing of these correction strings is given in Section 4.4. After testing all the correction strings a decision is made about the conclusion of the correction process for the particular error; Section 4.5 considers such correction decisions. If none of the correction strings generated passes the tests (of Section 4.4) the parser is backed up (unless the process of backing up has reached the left end of the substructure in which case the algorithm announces its failure to correct the error detected in that substructure) and the correction process is repeated. Section 4.6 explains the process of backing up the parser. Section 4.8 includes some language-dependent heuristics to aid error correction.

4.1 SKELETON, A PROTOCOMPILER

SKELETON of the XPL system was used in building the compiler of our implementation. A detailed description of the XPL system is given in McKeeman et al. (1970). Only a brief account will be included here.

The first step in using SKELETON to generate a syntax-directed compiler is to give a BNF grammar of the language for which the compiler is to be written as input to a program called ANALYZER. The productions of the grammar are written in the usual BNF -- non-terminals enclosed in angular brackets (< , >) and terminals without the angular brackets.

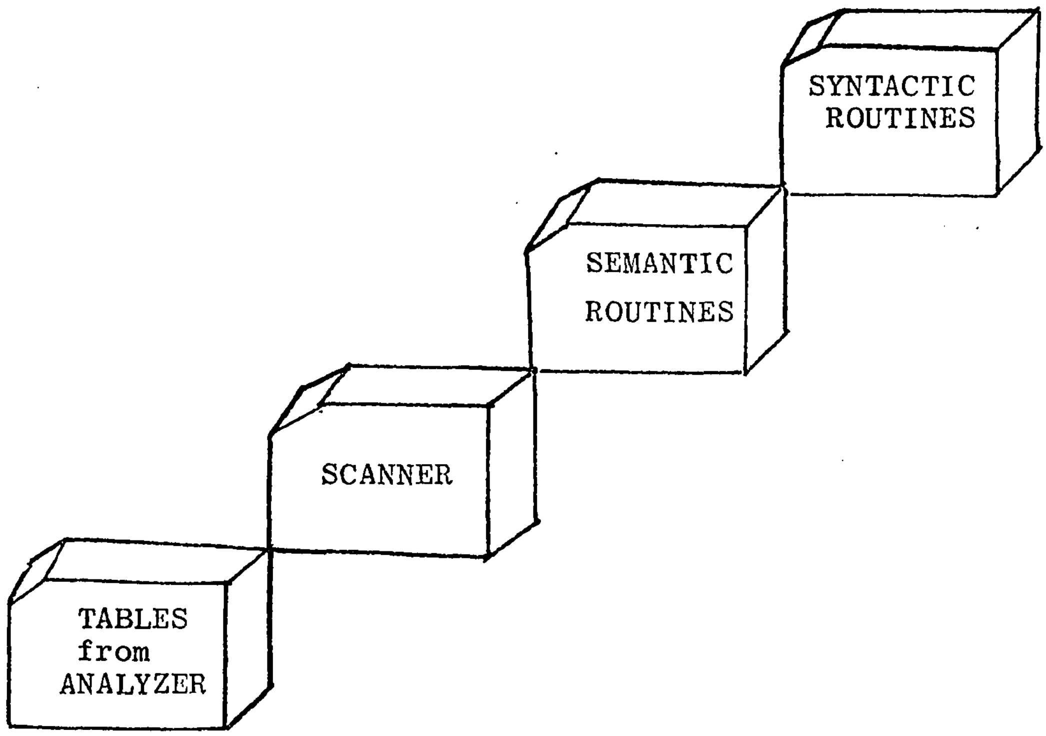
*A protocompiler is a model compiler on the basis of which one can build his own compiler.



After checking for ambiguities, ANALYZER computes the tables that constitute C1 and C2 decision functions (see Section 2.45 for definitions of C1 and C2 functions). C1 tables represent a (2,1) precedence matrix. For a grammar with N symbols, the (2,1) precedence matrix is of the size N x N x N. For grammars of practical programming languages, N >100, and the size of the precedence matrix becomes impractically large. ANALYZER uses the following scheme to economize the storage of the precedence matrix.

A (2,1) precedence matrix indicates which one of the three relations $\doteq\lessdot$, $\gtrdot$, or null exists between the string of two symbols from the top of the parse stack and the next input symbol. In other words, it determines the precedence relation from (2,1) context. Most of the time a (1,1) context is sufficient to decide which one of the three relations holds. Therefore, a 2-dimensional matrix will do for most of the cases. In a few cases, (1,1) context is not sufficient to decide if $\doteq\lessdot$ or $\gtrdot$ holds. For such cases, ANALYZER enters a conflict symbol, # , say, in the 2-dimensional matrix. Corresponding to each entry in the matrix there will be two sets of triples, one for the $\doteq\lessdot$ relation and the other for the $\gtrdot$ relation. ANALYZER produces only those triples for which $\doteq\lessdot$ holds. Absence of a triple indicates a $\gtrdot$ relation. The triples corresponding to all the # entries are listed in an array C1_TRIPLES.

After ANALYZER produces the parsing tables, the next step is to construct the remaining parts of the compiler. A compiler generated from SKELETON has the form:



The syntactic analysis routines are provided by SKELETON. The SCANNER and SEMANTIC ANALYZER are to be provided by the compiler writer.

4.2 DETECTION AND LOCATION OF ERRORS

Figure 4.1 is an overview of bottom-up parsing. Entrance to the boxes ERROR_1 or ERROR_2 indicates an error in the input string. In either case we say that the parser has detected the existence of an error. Entrance to the box ERROR_1 corresponds to having a null relation in the C1 matrix. The parser indicates this by printing out the message:

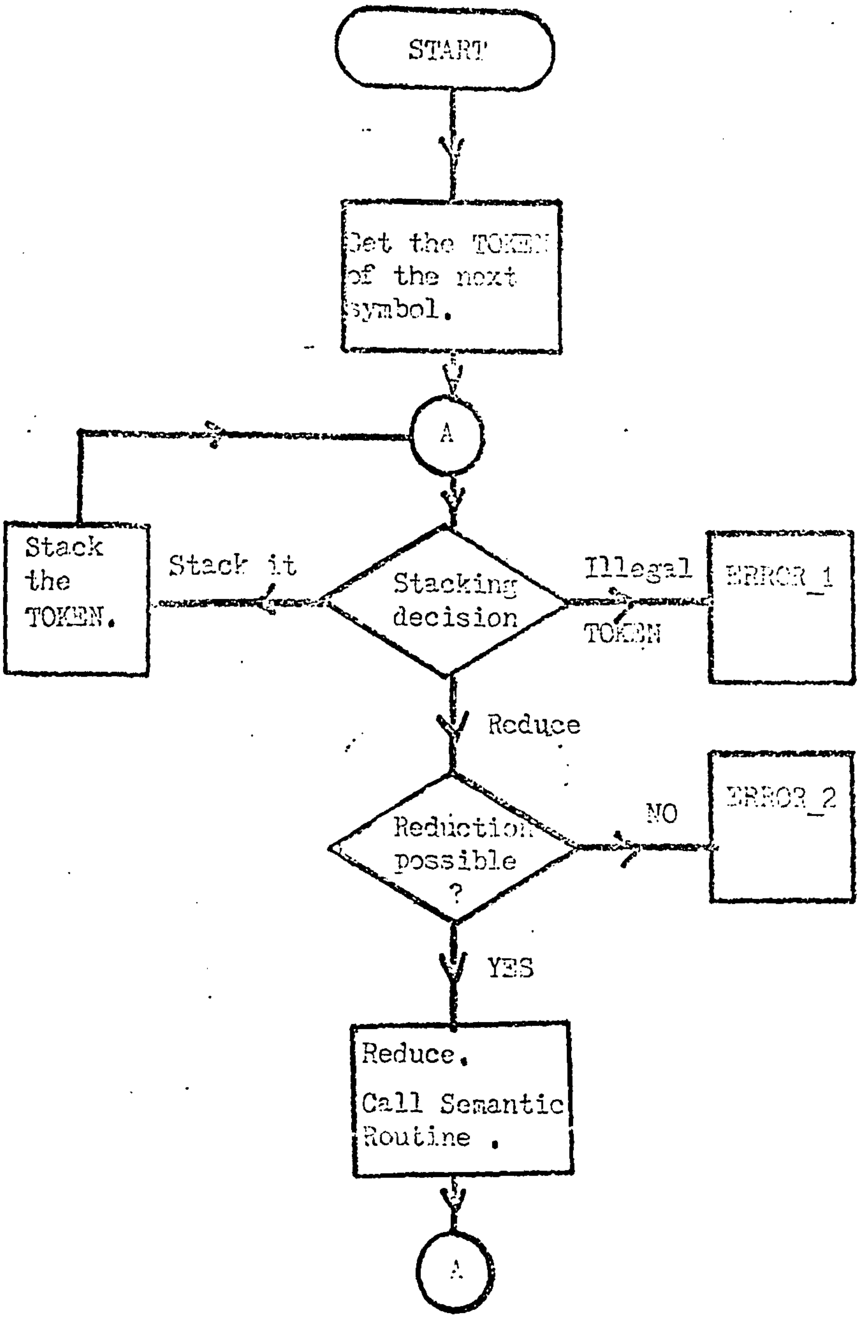


Figure 4.1 A precedence parser.

```
                         DECLARE (I,J,) FIXED ;
***ERROR, ILLEGAL SYMBOL PAIR: , ) .        ↑
```

In Chapter Three we referred to this type of error as action error; it corresponds to type 0 error in Leinius (1970).

Entrance to box ERROR_2 in Figure 4.1 indicates that the tail of the handle has been found because there is a "2" in the C1 matrix indicating a reduction needs to be done on a top portion of the parse stack, but no production can be found to satisfy the following conditions:

1) its right side matches a top portion of the parse stack,
2) it satisfies the left and the right context.

In Chapter Two this was called reduction error. Our reduction error corresponds to type 1, type 2 and type 3 errors of Leinius. The following is an example of reduction error.

```
                            A + AA(C + 1 ;
***ERROR, NO PRODUCTION APPLICABLE
PARTIAL PARSE TO THIS POINT IS:
      <statement list><variable><replace><subscript head>.
```

It is important to note that the place where the parser announces an error is not necessarily where the error-causing symbol is. Rather it is the place where the parser noticed for the first time the existence of an incorrect input string. The location of the error-causing symbol depends on how the incorrect string is interpreted. For example, for the statement

```
                          A : B(C + 1) + 1 ;
***ERROR, NO PRODUCTION APPLICABLE.
PARTIAL PARSE TO THIS POINT IS:
      <statement list> <label definition> <expression> .
```

The location of the error-causing symbol depends on which correction string is considered as the intended string. Syntactically the two corrections cannot be distinguished. However, the latter may change the semantics of the identifier A.

The detection of the existence of error is automatic but determining the location of the error is not. The error-causing symbol is not finally located until correction for the incorrect string is decided. At first, it may appear that the symbol that causes the parser to announce the error must be the error-causing symbol. Unfortunately, this is not so. The error-causing symbol may not be detected until many symbols beyond the error are scanned. The delay in the detection of error may be due to:

1) insufficient right context (misinterpretation of the string),
2) insufficient left context.

Since these two causes of delay have been discussed in the last chapter we shall not repeat them here.

As a result of the delay in detecting errors the parser quite often announces the error a few symbols after the appearance of the error-causing symbol. Therefore, to locate the error-causing symbol the parser may have to be backed up several symbols. Also, another thing must be noted. When the parser announces the error, it indicates if it is an action error or reduction error. However, the information about the type of error does not indicate anything regarding the location of the error. Hence the process of error correction does not distinguish the two types of errors. The type of error is indicated for the benefit of the programmer only.

4.3 GENERATING CORRECTION STRINGS

Once the parser recognizes the presence of an error, a marker is placed under the next input symbol and the programmer is notified of the error. For example,

```
                              DECLARE (I,J,) FIXED ;
                                            ↑
***ERROR, ILLEGAL SYMBOL PAIR , ) .
          LAST PREVIOUS ERROR WAS DETECTED ON LINE 0 ***
PARTIAL PARSE TO THIS POINT IS:
      <statement list> DECLARE <identifier list> <identifier> .
```

(Procedure ERROR is responsible for writing this message.)

At this point the tokens of "DECLARE", "(" , "I" , "," , "J" , and "," have been stored in a stack called BUFFOR, and its pointer, BUF_PTR has the value 5* pointing is to the top element on BUFFOR. There is another stack, BCD_BUF, to store the EBCDIC form of these terminals. After the error message is printed, the procedure STORE_INFORMATION is called to store the following information.

1. The card number (LINE_NO) where the error was recognized,
2. The parse stack (SAVE_STACK) and its pointer (SAVE_SP),
3. The token (SAVE_TOKEN) and the EBCDIC code (SAVE_BCD) of the next input symbol,
4. The token (TOKEN_IN_ADVANCE) and EBCDIC (BCD_IN_ADVANCE) of the symbols up to and including the symbol after the next delimiter (";" or "THEN") are read in advance. The pointer MARK points to the top of the stacks** TOKEN_IN_ADVANCE and BCD_IN_ADVANCE.

---

*The count starts at 0.

**TOKEN_IN_ADVANCE and BCD_IN_ADVANCE are actually so called decques. These elements will be added and deleted from the top and the bottom as well.

In other words, STORE_INFORMATION has isolated the string $t_0 t_1 \dots t_k t_n$. t_0 is the first symbol after the previous delimiter. After the completion of the parsing of symbols up to a delimiter the PARSEr reinitializes BUF_PTR to 0. Therefore, at the time of detection of error, BUF_PTR = k-1, and BUFFOR contains $t_0 \dots t_{k-1}$. t_k is saved in SAVE_TOKEN. TOKEN_IN_ADVANCE contains the tokens of $t_{k+1}, \dots, t_n$, t_{n+1} where t_{n+1} is the symbol after the delimiter t_n. Then the procedure TRY_AGAIN is called to generate the correction strings. Next, we shall describe how TRY_AGAIN controls the generation of the correction strings one by one.

The following sets of strings are to be generated:

(4.32)	$t_0 \dots t_{k-1} r t_k \dots t_n$,	(inserting r in front of t_k)
(4.33)	$t_0 \dots t_{k-1} r t_{k+1} \dots t_n$,	(replacing t_k by r)
(4.34)	$t_0 \dots t_{k-1} t_{k+1} \dots t_n$,	(deleting t_k)
(4.35)	$t_0 \dots t_{k-1} t_{k+1} t_k \dots t_n$	(permuting t_k and t_{k+1})

Using the notation of Rule (3.21), if string (4.31) is written as $w = w_1 w_2 w_3$ then the strings in (4.32), (4.33), (4.34), or (4.34) are given by $w_1 s_2 w_3$ where s_2 is

$$r t_k t_{k+1}, \quad r t_{k+1}, \quad t_{k+1} \quad \text{or} \quad t_{k+1} t_k$$

respectively. In other words, the string (of tokens) to be generated is

$$w_1 s_2 w_3 \ .$$

The tokens of w_1 are already scanned and parsed. That much of the correction string $w_1 s_2 w_3$ need not be generated. The remaining part, viz. $s_2 w_3$ is generated by presenting the tokens one by one. The token t_{k+1} and the tokens in w_3 are all in TOKEN_IN_ADVANCE, and the token t_k is saved in SAVE_TOKEN. TRY_AGAIN controls the generation of $s_2 w_3$ in the following way:

a) for $s_2 = rt_k t_{k+1}$,
it sets TOKEN = r and SUCCESS_PTR = 0 ;

b) for $s_2 = rt_{k+1}$
it sets TOKEN = r and SUCCESS_PTR = 1 ;

c) for $s_2 = t_{k+1}$,
it sets TOKEN = t_{k+1} and SUCCESS_PTR = 2 ;

d) for $s_2 = t_{k+1} t_k$,
it sets TOKEN = t_{k+1}, SUCCESS_PTR = 1 and interchanges the values of SAVE_TOKEN and TOKEN_IN_ADVANCE(0).

TOKEN is the current token presented for the construction of the correction string, and SUCCESS_PTR controls the generation of successive tokens by the algorithm (SCOOP_OR_SCAN):

```
if SUCCESS_PTR = 0 then
         TOKEN = SAVE_TOKEN;
else
         TOKEN = TOKEN_IN_ADVANCE (SUCCESS_PTR-1);
SUCCESS_PTR = SUCCESS_PTR + 1;
```

There are 42 terminals in XPL. For each terminal r, there is a string of the form (4.32). Hence, there will be 42 strings of the form(4.32). Not all of these are actually generated:

a) Among syntactically equivalent strings only one is generated,
b) those strings for which the terminal r has an invalid (1,1) context (which is indicated by 0 in the C1 matrix) need not be generated.

The same argument holds for the strings of the form (4.33). The procedure NEXT_LEGAL_TOKEN is responsible for skipping over the unwanted tokens.

4.4 TESTING THE CORRECTION STRING

In the last section, the correction string to be generated was denoted by $w_1 s_2 w_3$. The part w_1 is already scanned and parsed. The remaining string $s_2 w_3$ is parsed (by the procedure PARSE) as it is generated. While $s_2 w_3$ is being parsed one of two things may happen:

1. PARSE announces a syntactic error,
2. end of $s_2 w_3$ is reached without any error.

In the former case, the correction string $w_1 s_2 w_3$ is said to have failed the right-context check and is rejected. TRY_AGAIN is called to present the next correction string. In the latter case, the correction string $w_1 s_2 w_3$ has satisfied the right-context check and the procedure CONTEXT_CHECK is called to perform the next two tests:

a) left-context check,
b) semantics check on the string $w_1 s_2 w_3$.

First, CONTEXT_CHECK announces that the string $w_1 s_2 w_3$ has satisfied the right context. Consider the example

```
(4.40)      IF B THE X = 0 ; ...
***ERROR, ILLEGAL SYMBOL PAIR <identifier> <identifier> .
PARTIAL PARSE TO THIS POINT IS: <statement list> IF <identifier>.
```

One of the correction strings is IF B : X = 0 ; (obtained by replacing "THE" by ":"). This string parses without causing any syntactic error. Therefore, CONTEXT_CHECK announces:

```
.... : IS BEING CONSIDERED AS A REPLACEMENT FOR "THE".
```

We shall define formally what is meant by "a correction string fails the left-context check." Let $t_0 \ldots t_k \ldots t_n$ be an incorrect string, and let $w_1 w_2 w_3$ be a correction string where

$$w_1 = t_0 \ldots t_{k-1}, \quad w_3 = t_{k+1} \ldots t_n, \quad \text{and} \quad s_2 \quad \text{is}$$
$$rt_k t_{k+1}, \ rt_{k+1}, \ t_{k+1} \quad \text{or} \quad t_{k+1} t_k \ .$$

The correction string $w_1 s_2 w_3$ is said to <u>fail the left-context check</u>.

To see if the correction string $w_1 s_2 w_3$ satisfies the left-context check a (k,1) precedence matrix, where $k = w_1$, is required. Strictly speaking, C1 is just a (1,1) precedence matrix. Therefore, when $k > 1$ the C1 matrix cannot check the left context of a correction string. The following scheme is used to check the left context. The right context is checked automatically during the continued parsing of the part $s_2 w_3$. Therefore a correction string undergoes the right-context check first; then in case it does satisfy the right-context check it undergoes the left-context check. Satisfaction of the right-context check implies that the correction was completely parsed and the parser detected no error during this time. If the correction string satisfies the syntax of substructure, the top of the parse stack should reflect this. Therefore, the left-context check involves examining the portion of the parse stack that corresponds to the parsing of the most recent substructure. This raises the question: what is the syntax of substructure? We shall answer this by considering the following example.

A string between two consecutive delimiters* (including the right delimiter) corresponds to a substructure. In the following segment of an XPL program each line contains a substructure.

*See footnote on the next page.

```
(1)   DECLARE (A,B) FIXED ;
(2)   IF A = B THEN
(3)       A = 0 ;
(4)   ELSE B = 0 ;
(5)   DO I = 1 TO 5 ;
(6)       A = 1 ;
(7)   END ;
(8)   XX : PROCEDURE ;
(9)       RETURN ;
(10)      END XX ;
```

When string (1) is completely parsed it reduces to <statement list> . The last three reduction steps are:

<declaration statement> ⇒ <basic statement>
<statement>
<statement list> .

String (2) reduces to <if clause> . String (3) reduces to <basic statement> . However, in the right context of "ELSE" the non-terminal <basic statement> will not be reduced to <statement> . The partial parse stack to this point has the form:

<basic statement>
<if clause>
<statement list>

---

*The syntactic unit in XPL that naturally corresponds to the substructure described in Section 3.22 is <statement> . A <statement> is either a <basic statement> or an <if statement> . Either of these two are delimited by a ",". In an <if statement> what follows "THEN" is a <statement> in itself. By considering "THEN" also as a delimiter we can increase the number of errors that are correctable by our algorithm. Thus, we choose ";" and "THEN" as the two delimiters in XPL. In case the delimiter is absent, the presence of: "IF", "DO", "DECLARE" etc. indicates the beginning of the next substructure.

String (4) will first reduce to ELSE <statement> and then

<basic statement> ELSE <statement>
⟹ <true part><statement> .

Then,

<if clause><true part><statement>
⟹ <if statement>
⟹ <statement> .

Finally, the partial parse stack will have the form:

<statement list> .

String (5) reduces to <group head> . String (6) first reduces to <statement> then

<group head><statement> ⟹ <group head> .

String (7) first reduces to <ending> ; then,

<group head><ending> ; ⟹ <group> ;
⟹ <basic statement>
⟹ <statement> .

Once again, the partial parse stack becomes <statement list> . String (8) reduces to <procedure head> . String (9) reduces to <statement list> and the partial parse stack assumes the form:

<statement list>
<procedure head>
<statement list> .

The "END XX" of string (10) first reduces to <ending> then,

<procedure head><statement list> <ending>
=⇒ <procedure definition> .

Finally, <procedure definition> ;
=⇒ <basic statement>
=⇒ <statement>

and the partial parse stack becomes

<statement list> .

The only time a substructure does not reduce to <statement> and then to <statement list> is when it is a part of a compound statement. The following are the compound statements in XPL:

<if statement>
<group>
<procedure definition>;

Since the above example considers each of these compound statements we have exhausted the different ways a substructure can be reduced. Therefore we can conclude that a substructure reduces to one of the following non-terminals:

<statement list>, <basic statement> ,
<if clause> , <group head> or <procedure head> .

In other words, the syntax of substructure is given by

```
substructure ::= <statement list>
               | <basic statement>
               | <if clause>
               | <group head>
               | <procedure head>
```

Therefore, the testing of the so called left-context is performed by a check of the parse stack for any symbols not mentioned above. In case a symbol not mentioned above is found on the parse stack, the correction string has failed the left-context check and is therefore rejected. For example, the correction string (4.41) generated to correct the string (4.40)

```
              IF    B    :     X  =  0  ;
              └─────┘    ↑     └────────┘
(4.41)      left context |    right context
                   terminal for
                   replacement
```

causes the parse stack configuration to be

<statement list>

IF

<statement list>

...

Since "IF" is not one of the symbols listed above, string (4.41) fails left-context check.

4.41 Semantic Check

In case a correction string satisfies the left-context check it undergoes semantic analysis. The semantics for our XPL compiler was borrowed from the XCOM compiler for XPL (McKeeman et al., 1970), with the following differences:

1) Our compiler does not produce any machine code,
2) XCOM does not distinguish between subscripted and unsubscripted identifiers. The flag ARRAY TYPE was added to make this distinction possible.

The semantics of our compiler, like the semantics of compilers of most programming languages, requires the following:

1) Each label referenced must actually appear as the label of an appropriate statement in the program.
2) No identifier can be declared more than once.
3) All identifiers except labels must be declared before use.
4) The arguments of a function call must be compatible both in number and in attributes with the definition of the function.

Failure to comply with any of the requirements listed above results in a semantic error. While in the standard mode, the compiler attempts to resolve the semantic error. In other words, if a substructure contains one or more semantic errors but no syntactic errors, an attempt is made to resolve the error. For example,

1) if an identifier is used without declaration, the default type, viz. fixed type is assumed, or
2) if an identifier is declared twice, the second declaration is ignored.

A semantic error detected during error correction mode may be due to the programmer or it may be introduced by the correction string that is being tested for semantic context check. In either case, a semantic error detected during the error correction mode causes the value of the switch SEMANTICS to be false. If at the end of the semantic analysis of a correction string the switch SEMANTICS is false, then the correction string has failed the semantics check and therefore rejected. This, of course, implies that

> semantic errors committed in a substructure by the programmer in addition to a syntactic error in the same substructure will inhibit the correction of the syntactic error.

If a correction string satisfies the semantics check, it is saved (by SAVE_CORRECTION).

4.5 CORRECTION DECISION

After all the correction strings are tested for a) right-context check, b) left-context check, and c) semantics check a decision is made if the error correction process for the particular error being corrected should be concluded. The decision depends on the number of correction strings that where saved after all the tests. The following three are possible:

1. several correction strings were saved,
2. exactly one string was saved,
3. no strings were saved.

All these cases were treated in the last chapter. It was mentioned under case 3 that if no correction strings were saved and if the left end of the substructure was not reached the parser must be moved one position to the left and the process of error correction repeated. In the next section we shall illustrate the backward movement of the parser of our implementation.

4.6 BACKING UP THE PARSER

Consider the segment

(4.61) AA(1) = B(C + 1 ; A ...

(Assume that the statement DECLARE AA(4) FIXED, (A,B,C) FIXED has appeared previously.) The parser detects the error while scanning the symbol ";" and writes the message

***ERROR, NO PRODUCTION APPLICABLE.

PARTIAL PARSE TO THIS POINT IS:

<statement list> <variable> <replace> <subscript head> <expression>.

At this point BUFFOR contains the tokens of the following symbols:

(4.62) AA(1) = B(C + 1

and BUF_PTR = 9 (the count starts at 0). STORE_INFORMATION saves the token of ";" in SAVE_TOKEN and saves ";" itself in SAVE_BCD. Since the existence of the error was detected at the end of the substructure there is only one symbol, viz., "A" to be read in advance. Therefore, TOKEN_IN_ADVANCE(0) = token of <identifier> , and MARK = 0.

TRY_AGAIN is called to generate all the correction strings. The following corrections satisfy the right context:

(1)	AA(1) = B(C + 1) ;	(Insert ")")
(2)	AA(1) = B(C + 1 THEN ;	(Insert "THEN"
(3)	AA(1) = B(C + 1 THEN	(Replace ";" by "THEN"

Since B is not an array, correction string (1) does not satisfy the semantic context and is therefore rejected. (2) and (3) are also rejected because they both fail to satisfy the left context check. Thus, no correction strings are saved. The message:

> ...NO CORRECTIONS ON THE PRESENT POSITION WERE SUCCESSFUL
> THEREFORE, THE NEXT POSITION ON THE LEFT WILL BE TRIED.

is written to indicate that the backward movement of the parser is in process.

At this point the partial parse stack is:

(4.63)
<expression>
<subscript head>
<replace>
<variable>
<statement list>

If the top symbol of the parse stack were a terminal, backing up the parser by one step would simply mean taking the symbol on the top of the parse stack and adding it to the head of the input string (where it originally came from). However, the top symbol on the parse is not always a terminal. For example, in the present case, the symbol on the top of the parse stack is the non-terminal expression which is the reduction of the terminals

C + 1 .

Therefore backing up the parser implies undoing this reduction, or "unreducing". In other words, the partial parse stack should be changed to

(4.64)
1
+
C
<subscript head>
<replace>
<variable>
<statement list>

Now the top symbol on the parse stack is a terminal. The backward movement of the parser by one symbol is completed if the symbol on the top of the parse stack is moved to the head of the input string.

The most difficult part of moving the parser backwards is "unreducing" the non-terminal on the top of the parse stack. For a bottom-up left-to-right parser "unreducing" amounts to top-down right-to-left parsing. For example, in our case, the problem was to find how much of BUFFOR, viz.,

AA(1) = B (C + 1

from right-to-left was reduced to the non-terminal <expression> . Given <expression> , finding out that

<expression>==> * C + 1

by scanning BUFFOR) amounts to parsing top-down right-to-left. Therefore, "unreducing" would call for a new parser and a new parsing table. This can be avoided by using the following method.

Instead of asking how much on the top of BUFFOR corresponds to the symbol on the top of the parse stack, we ask how much on the top of the parse stack corresponds to all of BUFFOR. BUFFOR consists of the terminals since the end of the last substructure. In Section 4.5, it was mentioned that the end of the parsing of a substructure is indicated by the presence of one or more of the following symbols (and no others).

<statement list> , <basic statement> ,
<group head> , <procedure head> and <if clause>.

Therefore, all the symbols on the parse stack corresponding to the terminals in BUFFOR are identified they can be deleted and the parsing of the present substructure can be repeated by taking the terminals from (the bottom of) BUFFOR rather than the input string. Also, the parsing can be stopped when the top of BUFFOR is reached and the top terminal of BUFFOR can be added to the head of the input string. For example, in the stack configuration (4.63) the symbols

<expression> , <subscript head> , <replace> and <variable>

correspond to the terminal string (4.62) which is the present contents of BUFFOR. The procedure UNSTACK removes the above symbols from the parse stack leaving only

<statement list>.

Then UNSTACK sets the switch

RESTACKING = true,
BUF_PTR_LMT = BUF_PTR - 1, and
(In our present case, BUF_PTR = 9, therefore, BUF_PTR_LMT = 8)
BUF_PTR = 0.

Now, we proceed with reparsing of the segment (4.61) starting from the symbol "AA". As the parsing continues BUF_PTR is incremented by 1 and the next token is taken from BUFFOR. When

BUF_PTR = BUF_PTR_LMT (8, in the present case)

the process of restacking stops and the procedure RESTACK sets

RESTACKING = false.

By this time the partial parse stack has the form:

+
C
<subscript head>
<replace>
<variable>
<statement list>.

The terminal "1" is on the top of the stack BUFFOR. To complete the process of backing the parser the terminal "1" has to be pushed back on the input stack. In error correction mode TOKEN_IN_ADVANCE plays the role of the input stack. Therefore the terminal "1" is pushed on the stack TOKEN_IN_ADVANCE (by RESTACK).

4.7 THE PROCEDURES

The procedures constituting the error correction algorithm have been referrenced in Section 4.2 through 4.6. An alphabetic list of these procedures is given in table 4.1. A flow chart of PARSE, the procedure central to all the procedures, is given in Figure 4.2. Figure 4.3 gives an overview of the correction algorithm.

4.8 HEURISTICS

We shall conclude this chapter with a few heuristic aids for correcting errors in XPL language.

First, we shall consider the case of an unmatched

END ;

statement. There is no local error in the above statement. The global error is caused by one of the following two cases:

1. the <group head> of a <group> statement is missing or incorrectly specified,
2. the <procedure head> of a <procedure definition> statement is missing or incorrectly specified.

In either case, the best solution is to delete the statement "END;", since it is too late to go back and correct the <group head> or the <procedure head>. Our error correction algorithm would find the following corrections:

; END (interchanging "END" and ";")
; (deleting "END")
RETURN ; (replacing "END" by "RETURN")

Instead, in our implementation an ad hoc procedure, UNMATCHED_END, is given control and the standard error correction is bypassed.

The next case will be illustrated by examples. Consider the incorrect statement:

A B + 1 ;

The automatic correction algorithm will find the following corrections:

A = B + 1 ; (inserting "=" before B)
A = + 1 ; (replacing B by "=")

Obviously, the former correction is more "natural" than the latter. Also, for the incorrect statement:

A = B IF A < 0 ...

the corrections selected by the automatic correction algorithm are:

A = B ; IF A < 0 ... (inserting ";" before "IF")
A = B ; A < 0 ... (replacing "IF" by ";")

In this case, the latter correction will introduce an error in the next statement.

In our implementation the problem is solved in the following way. For an incorrect string

$$t_0 t_1 \ldots t_k \ldots t_n$$

if the correction string

$$t_0 t_1 \ldots s' t_k \ldots t_n$$

(obtained by inserting a token s' before t_k) is saved as one of the successful corrections, then the correction string

$$t_0 t_1 \ldots s' t_{k+1} \ldots t_n$$

(obtained by replacing t_k by the token s') is not generated.

TABLE 4.1

Procedure	Called by	Calls	Function
ABORT_STATEMENT	CORRECTION_DECISION		Aborts the offending statement.
CONTEXT_CHECK	PARSE	CORRECTION_DECISION TRY_AGAIN, REPORT_CORRECTION, SAVE_CORRECTION.	Checks: 1) right context, 2) left context and 3) semantic context. If a correction satisfies all three then calls SAVE_CORRECTION to save it.
CORRECTION_DECISION	PARSE, CONTEXT_CHECK	WRITE_MESSAGE, UNSTACK, ABORT_STATEMENT, GET_CARD.	1) If no corrections backs up the parser by one symbol. If the backing up has reached the beginning of the statement, declares the failure of the error correction scheme and deletes the symbols up to the next semicolon. 2) If any corrections reports them. In case of multiple corrections, indicates which correction was chosen.
ERROR[1]	PARSE, SYNTHESIZE	STACK_DUMP	Prints error message. For syntactic error writes out the contents of the parse stack.
GET_CARD[1]	SCAN, CHAR		Reads source cards. During standard mode lists the cards as they are read, but during error-correction mode, the listing is delayed until the correction algorithm has finished trying to correct the error.

[1] Modified from the procedure (with the same name) in XCOM.

Procedure	Called by	Calls	Function
NEXT_LEGAL TOKEN	TRY_AGAIN		Among syntactically equivalent terminals considers only one. Also, skips over the terminals with invalid context.
PARSE	MAIN_PROCEDURE	STACKING, REDUCE, ERROR, RESTACK CONTEXT_CHECK, CORRECTION_DECISION, STORE_INFORMATION, SCOOP_OR_SCAN, TRY_AGAIN.	Decides which move: stack, reduce, accept or error. If stack, stacks the token; if reduce, calls reduce; if accept returns to MAIN_PROC; if error, enters error correction made.
REDUCE[1]	PARSE	PR_OK, SYNTHESIZE	If no production applicable returns the value false. Otherwise, calls SYNTHESIZE to perform the semantic analysis for the production.
REPORT_CORRECTION	CONTEXT_CHECK		Prints out the current correction that has satisfied the right context.
RESTACK	PARSE	TRY_AGAIN	Restacks the parse stack to the position one symbol before the previous position of correction.
RESTORE_STACKS	TRY_AGAIN, ABORT_STATEMENT		Restores the parser as it was before the latest correction was tried.
SAVE_CORRECTION	CONTEXT_CHECK		If a correction satisfies the right context, and the semantic context SAVE_CORRECTION is called to save it.

[1]Modified from the procedure (with the same name) in XCOM.

Procedure	Called by	Calls	Function
SCOOP_OR SCAN	PARSE	SCAN	In standard mode calls SCAN. In error correction mode gets the next TOKEN_IN_ADVANCE.
STACKING[1]	PARSE		Returns the value Q, in case of illegal symbol pair; 1, if they symbol is to be stacked; 2, if a reduction is to be performed.
STORE_ INFORMATION	PARSE	SCAN	Sets the error flag. Reads the symbols in advance for right context.
SYNTHESIZE[1]	REDUCE	ENTER, ID_LOCKUP,	Performs semantic analysis (does not produce any code). In case of a semantic error, attempts to resolve it and in error correction mode turns the switch SEMANTICS to false.
TRY_AGAIN	PARSE, RESTACK CONTEXT_CHECK	NEXT_LEGAL_TOKEN, RESTORE_STACKS	Arranges to: 1) insert a legal token, 2) replace by a legal token, 3) delete, 4) interchange
UNMATCHED_ END	PARSE		Deletes (ad noc) unmatched "end;" statement.

[1]Modified from the procedure (with the same name) in XCOM.

Procedure	Called by	Calls	Function
UNSTACK	CORRECTION_DECISION		Peels off the parse stack to the place corresponding to the end of the previous statement. BUF_PTR_LMT, the pointer indicating the position to be corrected is set one position back.
WRITE_MESSAGE	CORRECTION_DECISION		The valid corrections, if any, are printed.

[1]Modified from the procedure (with the same name) in XCOM.

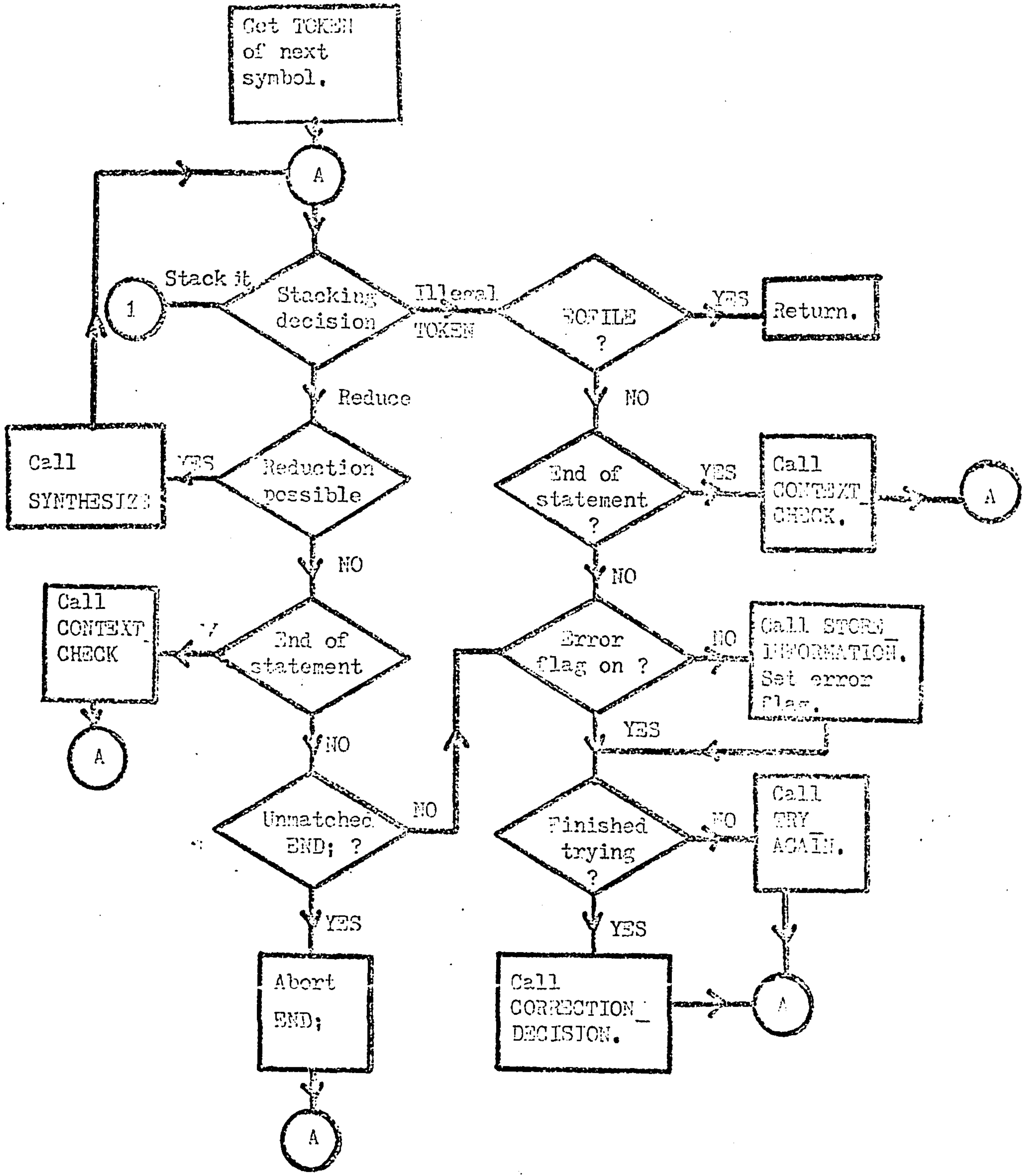


Figure 4.2 Flow Chart of PARSE.

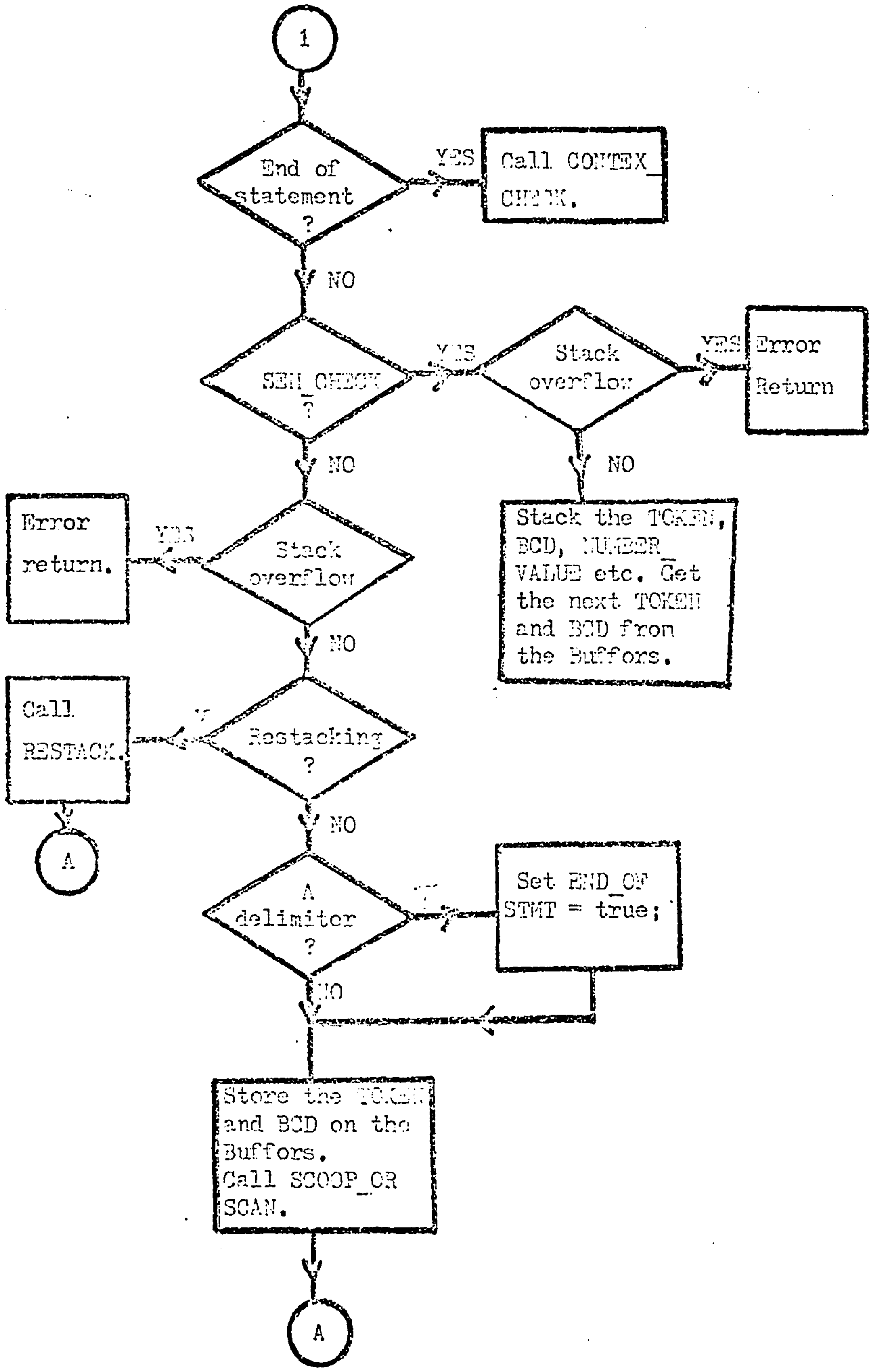
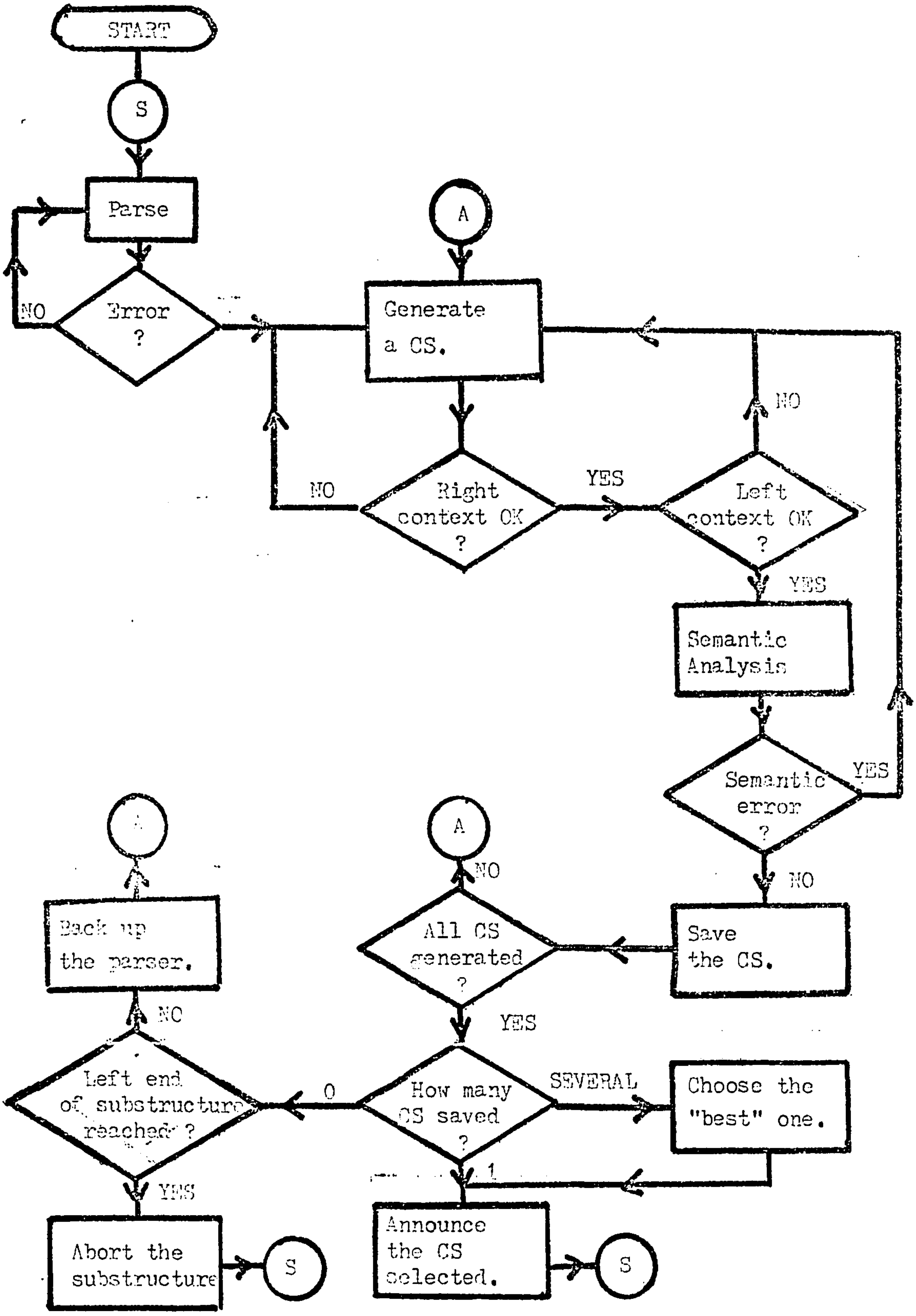


Figure 4.2 (continued)



CS = correction string

Figure 4.3

CHAPTER FIVE

CONCLUSION

5.1 SIGNIFICANCE OF THIS RESEARCH

In this thesis we have presented an algorithm for correcting syntactic errors. The significance of this research consists of the following results:

5.11 Locating the Error

High-quality error correction must not insist on making a correction where the existence of an error is detected but rather must solve the problem of actually finding the error. There are two reasons why the existence of error sometimes is not detected at the occurrence of the error: 1) the nature of the error, 2) the nature of the parser. There are parsers which are quite efficient in parsing correct programs but do not detect the existence of errors as early as certain other parsers. The error-correction algorithm to be used in such parsers will be of little use if it insists on making a correction where the existence of error was detected.

5.12 Testing the Corrections

A correction candidate must undergo through testing before it is selected as the final correction. A hasty and lenient test could select a candidate that may allow the parsing to continue correctly for a short time but later cause spurious errors. Therefore, a correction candidate must be tested in a context large enough to ensure the "goodness" of the candidate but not so large as to make the test very time-consuming. Most of the existing error-correction algorithms use only syntactic context for testing the correction candidates. In our algorithm the candidates are also tested in semantic context.

5.2 THE PERFORMANCE OF THE IMPLEMENTATION

The EXPL compiler is running with no major bugs. Many sample EPL programs have been run, typical examples of which are given in Appendix D. One program was run both under EXPL and the PL/1 F compilers to demonstrate how the PL/1 F compiler fails to correct errors which are not detected at the point of their occurrence. EXPL is capable of correcting such errors by moving to the left of the point where the error was detected.

5.21 Correction Speed

Since the vast variety of possible errors precludes definition of anything like a "typical" or "average" error, it is difficult to extract from our results answers to questions such as "how long does EXPL take to correct an error". The answers depend very much on the particular error. In order to get a rough estimate of the correction speed of EXPL, an XPL program with 30 statements was run several times, each time with a different number of "bugs". All times are relative to the IBM system 360/50.

With no errors the compilation time was 3.23 seconds. Figure 5.1 shows compilation time vs. number of errors per 30 cards.

The curve is almost linear and rises monotonically and not sharply.

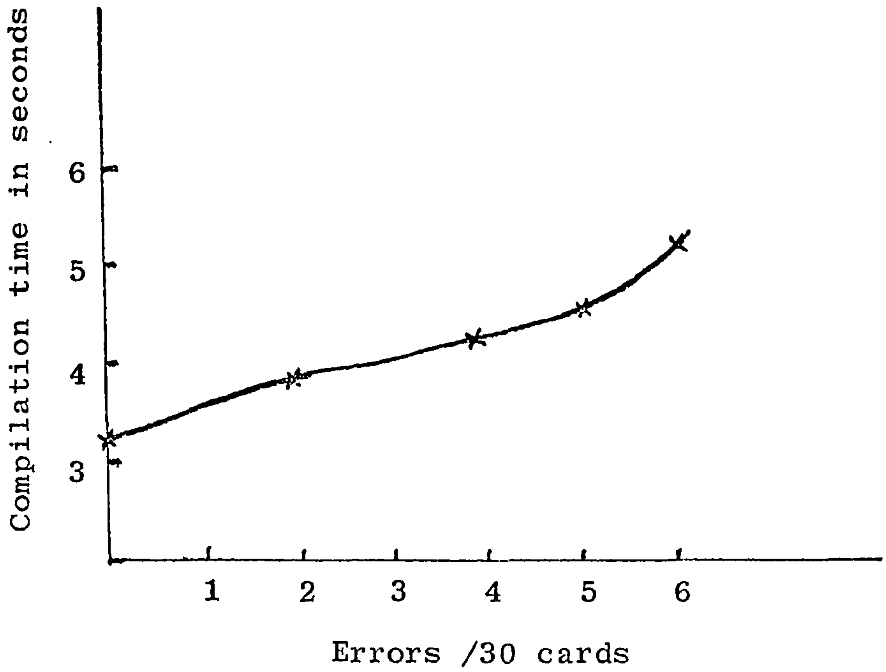


FIGURE 5.1

The correction speed can be improved by about 100 percent by the following modification. At present the algorithm does not stop after it finds a correction that satisfies all the tests. It finds all the corrections. If there are several alternates the algorithm is provided with a criterion by which it selects the "best" correction. For example, the criteria given in Section 3.22 select the "best" correction according to the priority:

1) interchange the error-causing symbol and the one to its right (highest priority),
2) delete the error-causing symbol,
3) replace the error-causing symbol by another symbol,
4) insert a symbol in front of the error-causing symbol.

Instead of finding all the corrections and then selecting the best one we can arrange the finding of the corrections according to the given criterion. With this modification the algorithm can stop after finding the first correction since it is the "best" one. This, on the average, will save about 50 percent of the time (which amounts to doubling the speed). The only disadvantage is that the user will not be able to see all the possible corrections.

5.3 EXTENSIONS

Our algorithm could be made more effective by adding several additional features. In our algorithm the scanner does not participate in correcting errors. The correction algorithm works on the tokens of the symbols rather than the symbols themselves. This makes the errors involving a missing space or a misplaced space immune to our algorithm. As indicated in Chapter Three, statements like

```
DEC LARE A FIXED ;
IFX = Y THEN Y = 0 ;
```

are not correctable by our algorithm. To be able to correct such errors we may need to:

1) Concatenate terminals to form one or more new terminals, e.g., DEC and LARE concatenated should give DECLARE.
2) A substring of a terminal may be recognized as a differnet terminal e.g., in "IFX", "IF" is a terminal in itself.

This can be done by involving the scanner in the correction process. The question then arises: how can the scanner know when and where to make a correction? To provide the scanner with its own error detection facility would slow down the error correction process. Therefore the scanner should only convert terminal symbols into tokens as usual and retain approximately one record of the latest input in a buffer. The syntactic analyzer can carry out the error correction procedure with the following modification. At a given point in the substructure if no correction is available before the point to be corrected is moved to the left the scanner should be called to attempt to correct the error at that point. The scanner can take certain actions. For example:

1) concatenate the last two terminal symbols if the cancatenated symbol is a legal terminal in the language
2) Split the terminal symbol into two terminal symbols.

In case the scanner is successful in getting new terminal symbol(s) from the old one(s) the syntactic analyzer can repeat the error correction process with the new tokens. Otherwise, the point to be corrected will be moved to the left as usual.

In the following (PL/1) statement:

...IFX = Y THEN ...

the existence of error is detected at the symbol "THEN" by the syntactic analyzer. At this point no corrections are available. The actions of the scanner will also fail. Finally, when the point to be corrected is shifted to the symbol "IFX" the scanner will be able to correct the error.

We have demonstrated that EXPL can correct errors better than other compilers that were available for our testing. We do not, however, claim that our error correction algorithm is the best possible one. Also, our algorithm is not yet perfected; there is scope for improving our algorithm which we mentioned in this and in the last chapter.

BIBLIOGRAPHY

Aho, A.V. and J.D. (1972)
The Theory of Parsing, Translation, and Compiling.
Volume I & II
Prentice-Hall, Englewood Cliffs, N.J.

Aho, A.V., P.J. Denning, and J.D. Ullman (1972)
Weak and mixed strategy precedence parsing.
Journal of ACM Vol. 19, No. 2, pp 225-243

Abrahams, P.W. (1972)
A syntax-directed parser for Recalcitrant Grammars.
International Journal of Computer Mathematics
Vol. 3, No. 2, September '72.

Beals, A.J. (1969)
The generation of a deterministic parsing algorithm.
Report No. 304, Department of Computer Science,
University of Illinois, Urbana.

Beals, A.J., J.E. LaFrance, and R.S. Northcote (1969)
The automatic generation of Floyd production syntactic analyzers. Report No. 350. Department of Computer Science, University of Illinois, Urbana.

Bell, J.R. (1969)
A new method for determining linear precedence functions for precedence grammars.
Communications of ACM Vol. 12, No. 10, pp 316-333.

Burgess, C.J. (1972)
Compile-time error diagnostics in syntax-directed compilers.
The Computer Journal Vol. 15, No. 4, Nov. '72.

Brooker, R.A., and D. Morris (1963)
The compiler-compiler.
Annual Review in Automatic Programming, Vol. 3.
Pergamon, Elmsford, N.Y., pp 229-275

Cheatham, T.E. (1965)
The TGS-II translator-generator system.
Proc. IFIP Congress 65. Spartan, N.Y., pp 592-593

Cheatham, T.E. (1967)
The Theory and Construction of Compilers (2nd ed.)
Computer Associates, Inc., Wakefield, Mass.

Cheatham, T.E., and K. Sattley (1964)
Syntax-directed compiling.
Proc. AFIPS Spring Joint Computer Conference,
Vol. 25. Spartan, New York, pp 31-57

Cocke, J., and J.T. Schwartz (1970)
Programming Languages and Their Compilers (2nd ed.)
Courant Institute of Mathematical Sciences, New York
University, N.Y.

Conway, R.W., et al. (1970)
PL/C. A high performance subset of PL/I.
Technical Report 70-55. Department of Computer Science,
Cornell University, Ithaca, N.Y.

DeRemer, F.L. (1968)
On the generation of parsers for BNF grammars: an algorithm.
Report No. 276. Department of Computer Science,
University of Illinois, Urbana.

Earley, J. (1966)
Generating a recognizer for a BNF grammar.
Computation Center Report, Carnegie-Mellon University,
Pittsburgh.

Feldman, J.A. (1966)
A formal semantics for computer languages and its application
in a compiler-compiler.
Communications of ACM Vol. 9, No. 1, pp 3-9

Fischer, M.J. (1969)
Some properties of precedence languages.
Proc. ACM Symposium on Theory of Computing, pp 181-190

Feldman, J.A., and D. Gries (1968)
Translator writing systems.
Communications of ACM Vol. 11, No. 2, pp 77-113

Floyd, R.W. (1963)
Syntactic analysis and operator precedence.
Journal of ACM Vol. 10, 316-333

Floyd, R.W. (1964a)
Bounded context syntactic analysis.
Communications of ACM Vol. 7, No. 2, pp 62-67

Floyd, R.W. (1964b)
The syntax of programming languages --a survey.
IEEE Trans. on Electronic Computers EC-13, No. 4, pp 346-353

Floyd, R.W. (1967)
Assigning meanings to programs.
Proc. Symposia in Applied Mathematics, Vol 19.
American Mathematical Society, Providence.

Foster, P.M. (1970)
Automatic syntax analysis.
American Elsevier, New York.

Freeman, D.N. (1964)
Error correction in CORC, the Cornell computing language.
Proc. AFIPS Fall Joint Computer Converence, Vol. 26.
Spartan, New York, pp 15-34

Graham, S.L. (1970)
Extended precedence languages, bounded right context languages and deterministic languages.
IEEE Conference Record of 11th Annual Symposium on Swithching Theory and Automata Theory, pp 175-180.

Gray, J.N. (1969)
Single pass precedence analysis.
IEEE Conference Record of 10th Annual Symposium on Swithching Theory and Automata Theory, pp 106-117.

Gries, D. (1971)
Compiler Construction for Digital Computers.
Wiley, New York.

Griffiths, T.V., and S.R. Petrick (1965)
On the relative efficiencies of context-free grammar recognizers.
Communications of ACM Vol. 8, No. 5, pp 289-300

Hopcroft, J.E., and J.D. Ullman (1969)
Formal languages and their relations to automata.
Addison-Wiley, Reading, Mass.

Hopgood, F.R.A. (1969)
Compiling Techniques. American Alsevier, New York.

Irons, E.T. (1961)
A syntax-directed compiler for ALGOL 60.
Comm. ACM 4:1, pp 51-55.

Irons, E.T. (1963a)
An error correcting parse algorightm.
Communications ACM 6:11, ;; 669-673.

Irons, E.T. (1963b)
The structure and use of the syntax-directed compiler.
Annual Review in Automatic Programming, Vol. 3.
Pergamon, Elmsford, N.Y., pp. 207-227.

James, E.B. and D.P. Partridge (1973)
Adaptive correction of program statements.
Communications ACM 16:1

James, Lewis R. (1972)
Technical Report CSRG 13.
A syntax-directec error recovery method, University of Toronto

Knuth, D.E. (1965)
On the translation of languages from left to right.
Information and Control 8:6, pp 607-639.

Knuth, D.E. (1968b)
Semantics of context-free languages. Math. Systems Theory
2:2, ;; 127 - 146. Also see Math. Systems Theory 5:1,
pp 95-96.

LaFrance, J. (1970)
Optimization of error recovery in syntax-directed parsing
algorithms. ACM SIGPLAN Notices 5:12. pp 2-17.

Lalonde, W.R., E.S. Lee, and J.J. Horning (1971)
An LALR(k) parser generator.
Proc. IFIF Congress 71, TA-3. North-Holland, Amsterdam,
pp 153-157.

Lee, J.A.N. (1967)
Anatomy of a Compiler. Van Nostrand Reinhold, New York.

Lévy, J.P. (1971)
Automatic correction of syntax errors in pargramming
languages. Cornell University

Lewis, P.M., II, and D.J. Rosenkrantz (1971)
An ALGOL compiler designed using automata theory.
Proc. Symposium on Computers and Automata, Microwave
Research Institute Symposia Series, Vol. 21. Polytechnic
Institute of Brooklyn., New York, pp 75-88.

Lewis, P.M., II, and R.E. Stearns (1968)
Syntax-directed transduction. J. ACM 15:3, ;; 464-488.

Lucas, P., and K. Walk (1969)
On the formal description of PL/I.
Annual Review in Automatic Programming, Vol. 6, No. 3.
Pergamon, Elmsford, N.Y., pp 105-182.

McClure, R.M. (1965)
TMG-a syntax-directed compiler.
Proc. ACM National Conference, Vol. 20. pp 262-274.

McClure, R.M. (1972)
An appraisal of compiler technology
AFIPS conference proceedings, Vol. 40 pp.1-9.

McKeeman, W.M., J.J. Horning, and D.B. Wortman (1970)
A Compiler Generator. Prentice-Hall, Englewood Cliffs, N.J.

Morgan, J.L. (1970)
Spelling correction in systems programs.
Communications ACM 13:2, pp 90-93.

Peterson, Thomas G. (1972)
Syntax error detection, correction and recovery parsers.
Stevens Institute of Technology

Rich, L.V. (1972)
Error detection, analysis and recovery in XPL base compilers
Government Reports Announce, Vol. 72, No. 9, May 10,
pp 105, Navel Post Graduate School, Monterey, California

Rosenkrantz, D.J., and R.E. Stearns (1970)
Properties of deterministic top-down grammars.
Information and Control 17:3, ;; 226-256.

Rustin, R. ed. (1970)
Dubugging techniques in large systems
Courant computer science symposium 1.

Stearns, R.E., and P.M. Lewis, II (9169)
Property grammars and table machines.
Information and Control 14:6, pp 524-549.

Williams, J.H. (1970)
Technical Report No. 58
Bounded context parsable grammars. University of Wisconsin

Wirth, N. (1968)
PL 360-a programming language for the 360 computers. J. ACM 15:1, ;; 37-34.

Wirth, N., and H. Weber (1966)
EULER-a generalization of ALGOL and its formal definition, Parts 1 and 2. Communications ACM 9:1-2, pp 13-23, and 89-99.

APPENDIX A

A BNF GRAMMAR FOR XPL

```
 1. <program> ::= <statement list> EOF
 2. <statement list> ::= <statement>
 3.                    | <statement list><statement>
 4. <statement> ::= <basic statement>
 5.               | <if statement>
 6. <basic statement> ::= <assignment> ;
 7.                     | <group> ;
 8.                     | <procedure definition> ;
 9.                     | <return statement> ;
10.                     | <call statement> ;
11.                     | <go to statement> ;
12.                     | <declaration statement> ;
13.                     |   ;
14.                     |   label definition  basic statement
15. <if statement> ::= <if clause><statement>
16.                  | <if clause><true part><statement>
17.                  | <label definition><if statement>
18. <if clause> ::=  IF <expression> THEN
19. <true part> ::= <basic statement> ELSE
20. <group> ::= <group head><ending>
21. <group head> ::=  DO;
22.                 | DO <step definition>;
23.                 | DO <while clause>;
24.                 | DO <case selector>;
25.                 | <group head><statement>
26. <step definition>::=<variable><replace><expression><iteration
                                                        control>
27. <iteration control>::= TO <expression>
28.                      | TO <expression> BY <expression>
29. <while clause> ::=  WHILE <expression>
30. <case selector> ::=  CASE <expression>
31. <procedure definition>::=<procedure head><statement list><ending>
32. <procedure head> ::= <procedure name>;
33.                    | <procedure name><type>;
34.                    | <procedure name><parameter list>;
35.                    | <procedure name><parameter list><type>;
36. <procedure name> ::= <label definition> PROCEDURE
37. <parameter list> ::= <parameter head><identifier>)
38. <parameter head> ::=  (
39.                     | <parameter head><identifier>,
40. <ending> ::=  END
41.           | END <identifier>
42.           | <label definition><ending>
43. <label definition>::= <identifier>:
```

```
44. <return statement> ::= RETURN
45.                      | RETURN <expression>
46. <call statement> ::= CALL <variable>
47. <go to statement> ::= <go to><identifier>
48. <go to> ::= GO TO
49.           | GOTO
50. <declaration statement>::= DECLARE<declaration element>
51.                     | <declaration statement><declaration element>
52. <declaration element> ::= <type declaration>
53.                          | <identifier>LITERALLY<string>
54. <type declaration> ::= <identifier specification><type>
55.                      | <bound head><number> )<type>
56.                      | <type declaration><initial list>
57. <type> ::= FIXED
58.          | CHARACTER
59.          | LABEL
60.          | <bit head><number>)
61. <bit head> ::= BIT (
62. <bound head> ::= <identifier specification>(
63. <identifier specification> ::= <identifier>
64.                              | <identifier list><identifier>)
65. <identifier list> ::=  (
66.                     | <identifier list><identifier>,
67. <initial list> ::= <initial head><constant>)
68. <initial head> ::=  INITIAL
69.                   | <initial head><constant>,
70. <assignment> ::= <variable><replace><expression>
71.                | <left part><assignment>
72. <replace> ::=  =
73. <left part> ::= <variable>,
74. <expression> ::= <logical factor>
75.                | <expression>|<logical factor>
76. <logical factor> ::= <logical secondary>
77.                    | <logical factor> & <logical secondary>
78. <logical secondary> ::= <logical primary>
79.                       | ¬<logical primary>
80. <logical primary> ::= <string expression>
81.              | <string expression><relation><string expression>
82. <relation> ::=  =
83.             |  <
84.             |  >
85.             |  <=
86.             |  >=
87.             |  ¬<
88.             |  ¬>
89.             |  ¬=
90. <string expression> ::= <arithmetic expression>
91.                 | <string expression><arithmetic expression>
92. <arithmetic expression> ::= <term>
93.                           | <arithmetic expression> + <term>
94.                           | <arithmetic expression> - <term>
95.                           |  +<term>
96.                           |  -<term>
```

```
 97. <term> ::= <primary>
 98.          | <term> * <primary>
 99.          | <term> / <primary>
100.          | <term> MOD <primary>
101. <primary> ::= <constant>
102.             | <variable>
103.             | (<expression>)
104. <constant> ::= <string>
105.              | <number>
106. <variable> ::= <identifier>
107.              | <subscript head><expression>)
108. <subscript head> ::= <identifier> (
109.                    | <subscript head><expression>
```

APPENDIX B

LIST OF IMPORTANT PROCEDURES IN EXPL

```
/*****************************************************************/
/**********                                           **********/
/**********              RESTCRE_STACKS               **********/
/**********                                           **********/
/*****************************************************************/

RESTORE_STACKS: PROCEDURE;
  /* RESTORE THE STATUS OF THE PARSER AS IT WAS WHEN THE ERROR WAS
                                                         DETECTED */
    SP=SAVE_SP;
    TOKEN=SAVE_TOKEN;
    BCD=SAVE_BCD;
    BUF_PTR=SAVE_PTR;
    DO I=2 TO SP;
       PARSE_STACK(I)=SAVE_STACK(I);
    END;
    SUCCESS_PTR=0;
END RESTORE_STACKS;

/*****************************************************************/
/**********                                           **********/
/**********            NEXT_LEGAL_TOKEN               **********/
/**********                                           **********/
/*****************************************************************/

NEXT_LEGAL_TCKEN:     PROCEDURE FIXED;
  /*  DETERMINE  THE TOKEN OF THE NEXT SYMBCL TO BE USED IN  FOR
   INSERTION OR REPLACEMENT  */
  DO WHILE TRY_INDEX<NT-2;    /* AMOUNG SYNTACTIC EQUIVALENT TOKENS USE
                          THE REPRESENTAIIVE AND SKIP OVER THE REST */
     TRY_INDEX=TRY_INDEX+1;
     IF BUF_PTR_LMT<0 & TRY_INDEX=1 THEN
        TRY_INDEX=2;
     IF TRY_INDEX=21 THEN TRY_INDEX=23; ELSE
     IF LAST_INDEX=6 THEN TRY_INDEX=16; ELSE
     IF TRY_INCEX=EOFILE | TRY_INDEX=25 | TRY_INDEX=33 | TRY_INDEX=41
       THEN TRY_INDEX=TRY_INDEX+1;
     DO I=1 TO TRY_INDEX - LAST_INDEX;
        IF SHIFT=4 THEN
          DO;
            C1_LINE=SUBSTR(C1_LINE,1);
            BITE=BYTE(C1_LINE);
            SHIFT=0;
          END;
        DO;
          SHIFT=SHIFT+1;
          NIBLE=BITE & 192;
          BITE=SHL(BITE,2);
        END;
     END;
     LAST_INDEX=TRY_INDEX;
     IF NIBLE¬=0 THEN
        RETURN TRY_INDEX;
  END;  /* OF DO WHILE TRY_INDEX < 40  */
  RETURN NT;  /* TO INDICATE SEARCH EXHAUSTED */
END NEXT_LEGAL_TOKEN;
```

```
/***********************************************************************/
/**********                                                 **********/
/**********                    TRY_AGAIN                    **********/
/**********                                                 **********/
/***********************************************************************/

 TRY_AGAIN:
   PROCEDURE;    /* INITIATE THE GENERATION OF THE NEXT CORRECTION
                                                               STRING  */
IF STACK_MESSED_UP THEN
   DO;
      CALL RESTORE_STACKS;
      STACK_MESSED_UP=FALSE;
      SEMANTICS=TRUE;
   END;

DO FOREVER;

   IF ¬INSERTED THEN
      DO;
         IF NEXT_LEGAL_TOKEN<NT-1 THEN
         DO;   /*  MAKE THE INSERTION  */
            SUCCESS_PTR=0;
            TOKEN=TRY_INDEX;
            BCD=V(TRY_INDEX);
            RETURN;
         END;
         ELSE  /* RE-INTITIALIZE FOR  REPLACEMENT */
         DO;
            IF MARK<0 THEN
               DO;
                  FINISHED_TRYING=TRUE;
                  RETURN;
               END;
            INSERTED=TRUE;
            SHIFT=0;
            LAST_INDEX=-1;
            TRY_INDEX=-1;
            C1_LINE=C1(PARSE_STACK(SP));
            BITE=BYTE(C1_LINE);
         END;
      END;
   ELSE
      DO;
         IF ¬REPLACED THEN
            DO;
               IF NEXT_LEGAL_TOKEN<NT-1 THEN
                  DO;
                     IF INSERTIONS>0 THEN
                        DO;   /* IF A TOKEN WAS A SUCCESSFUL INSERTION
                                 DO NOT TRY IT FOR REPLACEMENT */
                           I=INSERTIONS-1;
                           DO WHILE TRY_INDEX¬=INSERT_TOKEN(I) & I>=0;
                             I=I-1;
                           END;
                        END;
                     ELSE I=-1;
                     IF I<0 THEN
```

```
                        DO;
                           SUCCESS_PTR=1;
                           TOKEN=TRY_INDEX;
                           BCD=V(TRY_INDEX);
                           RETURN;
                        END;
                   END;
                ELSE
                   DO;
                     REPLACED=TRUE;
                     SHIFT=0;
                     LAST_INDEX=-1;
                     TRY_INDEX=-1;
                     C1_LINE=C1(PARSE_STACK(SP));
                     BITE=BYTE(C1_LINE);
                   END;
             END;
          ELSE
             DO;
                IF ¬DELETED THEN
                   DO;
                      DELETED=TRUE;
                      SUCCESS_PTR=2;
                      TOKEN=TOKEN_IN_ADVANCE(0);
                      BCD=BCD_IN_ADVANCE(0);
                      RETURN;
                   END;
                ELSE
                   DO;
                      IF ¬INTERCHANGED THEN
                         DO;     /* INTERCHANGE THE SYMBOLS */
                            INTERCHANGED=TRUE;
                            SUCCESS_PTR=1;
                            TEMP=SAVE_TOKEN;
                            SAVE_TOKEN=TOKEN_IN_ADVANCE(0);
                            TOKEN_IN_ADVANCE(0)=TEMP;
                            TOKEN=SAVE_TOKEN;
                            TEMP_BCD=SAVE_BCD;
                            SAVE_BCD=BCD_IN_ADVANCE(0);
                            BCD_IN_ADVANCE(0)=TEMP_BCD;
                            BCD=SAVE_BCD;
                            RETURN;
                         END;
                      ELSE
                         DO;     /* CHANGE THEM BACK THE WAY THEY WERE */
                            TEMP=SAVE_TOKEN;
                            SAVE_TOKEN=TOKEN_IN_ADVANCE(0);
                            TOKEN_IN_ADVANCE(0)=TEMP;
                            FINISHED_TRYING=TRUE;
                            TEMP_BCD=SAVE_BCD;
                            SAVE_BCD=BCD_IN_ADVANCE(0);
                            BCD_IN_ADVANCE(0)=TEMP_BCD;
                            RETURN;
                         END;
                   END;
             END;
       END;
   END;  /* OF  DO FOREVER*/

   END TRY_AGAIN;
```

```
   /*****************************************************************/
   /**********                                            **********/
   /**********              SAVE_CORRECTION               **********/
   /**********                                            **********/
   /*****************************************************************/

SAVE_CORRECTION:
 PROCEDURE;

 /* AFTER A CORRECTION THAT ALLOWS THE REQUIRED RIGHT CONTEXT  WE STORE
 THE PARTIAL PARSE_STACK FOR THAT CORRECTION FOR LATER USE IN CASE WE
 DECIDE  THAT IT IS THE FINAL CORRECTION */

   IF ¬INSERTED THEN
     DO;  /* RECORD SUCCESSFUL INSERTION */
        INSERT_TOKEN(INSERTIONS)=TRY_INDEX;
        INSERTIONS=INSERTIONS+1;
     END;

   ELSE
     IF ¬REPLACED THEN
       DO;  /* RECORD THE SUCCESSFUL REPLACEMENT */
          REPLACE_TOKEN(REPLACEMENTS)=TRY_INDEX;
          REPLACEMENTS=REPLACEMENTS+1;
       END;
     ELSE
       IF ¬INTERCHANGED THEN
             /* RECORD THE SUCCESSFUL DELETION */
         DO;
            DELETION=1;
         END;
       ELSE
         IF ¬FINISHED_TRYING THEN
           /* RECORD SUCCESSFUL INTERCHANGE */
           DO;
              INTERCHANGES=1;
           END;

   STMT_PTR=BUF_PTR-1;
   DO I=0 TO STMT_PTR;
      CORR_STMT(I)=BCD_BUF(I);
   END;

END SAVE_CORRECTION;
```

```
/*********************************************************************/
/**********                                               **********/
/**********              WRITE_MESSAGE                    **********/
/**********                                               **********/
/*********************************************************************/

WRITE_MESSAGE:
  PROCEDURE;

   CORRECTIONS= REPLACEMENTS + DELETION + INSERTIONS + INTERCHANGES;

   IF CORRECTIONS=0 THEN
      RETURN;

   ELSE IF CORRECTIONS=1 THEN

    DO; /*** UNIQUE CORRECTION ***/
      IF INSERTIONS=1 THEN
         DO;
           DOUBLE_SPACE;
           OUTPUT='ACTION------: '||V(INSERT_TOKEN(0))||
           ' WAS INSERTED BEFORE '||SAVE_BCD||' IN LINE NO.'||LINE_NO;
           INSERTIONS=0;
         END;

      ELSE IF REPLACEMENTS=1 THEN
         DO;
           DOUBLE_SPACE;
           OUTPUT='ACTION------: '||SAVE_BCD     ||' WAS REPLACED BY '
           ||V(REPLACE_TOKEN(0))||' IN LINE NO. '||LINE_NO;
           REPLACEMENTS=0;
         END;

      ELSE IF DELETION=1 THEN
         DO;
           DOUBLE_SPACE;
           OUTPUT='ACTION------: '||SAVE_BCD     ||' WAS DELETED'||
           ' IN LINE NO. '||LINE_NO;
           DELETION=0;
         END;

      ELSE IF INTERCHANGES=1 THEN
         DO;
           DOUBLE_SPACE;
           OUTPUT='ACTION------: '||V(TEMP)||' AND '||SAVE_BCD||
           ' ARE INTERCHANGED '||' IN LINE NO.'||LINE_NO;
           INTERCHANGES=0;
         END;

    END;    /***  OF UNIQUE CORRECTION  ***/
```

```
            ELSE  /**** IF DELTA > 1  ******/
         DO;
              OUTPUT=' ************ NO UNIQUE CORRECTION';

              I=0;
                IF INTERCHANGES=1 THEN
                   DO;
                     I=I+1;
                     OUTPUT='CORRECTION NO. '||I||X4||SAVE_BCD||' AND '
                     ||V(TOKEN_IN_ADVANCE(0))||' INTERCHANGED';
                     INTERCHANGES=0;
                   END;
                IF DELETION=1 THEN
                   DO;
                     I=I+1;
                     OUTPUT='CORRECTION NO. '||I||X4||SAVE_BCD||
                            ' DELETED';
                     DELETION=0;
                   END;

                DO WHILE REPLACEMENTS > 0 ;
                 I=I+1;
                 REPLACEMENTS=REPLACEMENTS-1;
                 OUTPUT='CORRECTION NO. '||I||X4||SAVE_BCD||
                  ' REPLACED BY '||V(REPLACE_TOKEN(REPLACEMENTS));
                END;

                DO WHILE INSERTIONS>0;
                 I=I+1;
                 INSERTIONS=INSERTIONS-1;
                 OUTPUT='CORRECTION NO. '||I||X4||V(INSERT_TOKEN(
                 INSERTIONS))||' INSERTED BEFORE '||SAVE_BCD;
                END;

         END;  /* OF IF DELTA > 1  */

END WRITE_MESSAGE;
```

```
/*****************************************************************/
/**********                                            **********/
/**********              STORE_INFORMATION             **********/
/**********                                            **********/
/*****************************************************************/

STORE_INFORMATICN:  PROCEDURE;  /* SAVE THE PRESENT INPUT SYMBOL AND
 READ SYMBOLS UP TO AND ONE AFTER THE NEAREST DELIMITER */
   DECLARE I FIXED;
               ATTEMPTED=TRUE;  /* ERROR CCRRECTION MODE FLAG ON */
               LINE_NO=CARD_COUNT;
               SAVE_SP=SP;
               SAVE_PTR=BUF_PTR;
               SAVE_TOKEN=TOKEN;
               SAVE_BCD=BCD;

               DO I=0 TO BUF_PTR;
                 SAVE_BUF(I)=BUFFOR(I);
               END;

               DO I=2 TO SP;
               SAVE_STACK(I)=PARSE_STACK(I);
               END;

               I=0;

               IF STOPIT(TOKEN)&TOKEN¬=SEMICOLON THEN
                 DO; /* IN CASE NO SYMBOLS NEED TO BE READ IN ADVANCE */
                   MARK=-1;
                   REPLACED=TRUE;
                   DELETED=TRUE;
                 END;
               ELSE
                 DO;
                   DO WHILE ¬STOPIT(TOKEN);    /* READ UNTIL THE
                                          BEGINNING OF NEXT STATMENT */
                     CALL SCAN;
                     TOKEN_IN_ADVANCE(I)=TOKEN;
                     BCD_IN_ADVANCE(I)=BCD;
                     MARK=I;
                     I=I+1;
                   END;

                   IF TOKEN=SEMICOLCN | BCD='THEN' THEN
                     DO;
                       CALL SCAN;
                       TOKEN_IN_ADVANCE(I)=TCKEN;
                       BCD_IN_ADVANCE(I)=BCD;
                       MARK=I;
                     END;
                 END;

               TRY_INDEX=-1;
               LAST_INDEX=-1;
               SHIFT=0;
               C1_LINE=C1(PARSE_STACK(SP));
               BITE=BYTE(C1_LINE);
 END  STORE_INFORMATICN;
```

```
/*********************************************************************/
/**********                                                **********/
/**********                    UNSTACK                     **********/
/**********                                                **********/
/*********************************************************************/

UNSTACK:  PROCEDURE;
 /* REMOVE ALL THE SYMBOLS FROM THE PARSE STACK CORRESPONDING TO
    THE PARSING OF THE PRESENT STATEMENT  */
 IF CONTROL(BYTE('Q')) THEN
 DO;
   OUTPUT='    THERFORE, THE NEXT POSITION ON THE LEFT WILL BE TRIED.';
   OUTPUT='... NO CORRECTIONS ON THE PRESENT POSITION WERE SUCCESSFUL';
 END;
        BUF_PTR_LMT=SAVE_PTR-1;
        RESTACKING=TRUE;

        DO WHILE ¬STOP1(PARSE_STACK(SP)) ;
             SP=SP-1;
             END;

        TOKEN=BUFFOR(0);
        IF BUF_PTR_LMT<0 THEN
           BUF_PTR=-1;
        ELSE BUF_PTR=0;
END UNSTACK;

/*********************************************************************/
/**********                                                **********/
/**********                ABORT_STATEMENT                 **********/
/**********                                                **********/
/*********************************************************************/

ABORT_STATEMENT:
 PROCEDURE;   /* DELETE THE OFFENDING STATEMENT  */
     LINE='ACTION------  ';
     IF BUF_PTR_LMT > -2 THEN
        LINE=LINE||X1||SAVE_BCD;
     DO I=0 TO MARK-1;
        IF LENGTH(LINE)>80 THEN
              DO;
                 OUTPUT=LINE;
                 LINE=X4;
              END;
        LINE=LINE||X1||BCD_IN_ADVANCE(I);
     END;
     LINE=LINE||'  ------ ABORTED';
     OUTPUT=LINE;

     CALL RESTORE_STACKS;
     DO WHILE ¬STOP1(PARSE_STACK(SP));
        SP=SP-1;
     END;
     OUTPUT='PARSING RESUMES IN STANDARD MODE';
END ABORT_STATEMENT;
```

```
/*****************************************************************/
/**********                                           **********/
/**********          REPORT_CORRECTION                **********/
/**********                                           **********/
/*****************************************************************/

REPORT_CORRECTION:
PROCEDURE;  /*  REPORT THE CORRECTION WHICH HAS SATISFIED THE RIGHT
                    SYNTACTIC CONTEXT */
   IF ¬INSERTED THEN
      OUTPUT='....  '||V(TRY_INDEX)||' IS BEING TRIED FOR INSERTION '
      ||'BEFORE '||SAVE_BCD;
   ELSE
     IF ¬REPLACED THEN
        OUTPUT='....      '||V(TRY_INDEX)||' IS TRIED AS A REPLACEMENT'
        ||' FOR '||SAVE_BCD;
     ELSE
       IF ¬INTERCHANGED THEN
          OUTPUT='....     '||SAVE_BCD||' IS BEING DELETED';
       ELSE
         IF ¬FINISHED_TRYING THEN
            OUTPUT='....    '||TEMP_BCD||' AND '||SAVE_BCD||
            ' ARE BEING INTERCHANGED';
END REPORT_CORRECTION;


/*****************************************************************/
/**********                                           **********/
/**********          CORRECTION_DECISION              **********/
/**********                                           **********/
/*****************************************************************/

CORRECTION_DECISION:
 PROCEDURE;

  CALL WRITE_MESSAGE;
  IF CORRECTIONS=0 & BUF_PTR_LMT>=0 & MARK>=0 THEN
       CALL UNSTACK;
   ELSE
    DO;
      IF CORRECTIONS>0 THEN
        DO;
             /* RESTORE STACKS ACCORDING TO THE CORRECTION */
          SP=CORR_SP;
          LAST_SP=SP;
          DO I=2 TO SP;
            PARSE_STACK(I)=CORR_STACK(I);
            LAST_STACK(I)=CORR_STACK(I);
          END;
```

```
          /* WRITE THE CORRECTED STATEMENT  */
          LINE=' CORRECTED STATEMENT ---';
          DO I=0 TO STMT_PTR;
             IF LENGTH(LINE) > 80 THEN
                DO;
                   OUTPUT=LINE;
                   LINE=X4;
                END;
             LINE=LINE||X1||CORR_STMT(I);
          END;
          OUTPUT=LINE;

        END;
      IF CORRECTIONS=0 THEN
        CALL ABORT_STATEMENT;

       IF MARK<0 THEN
        DO;
          TOKEN=SAVE_TOKEN;
          BCD=SAVE_BCD;
        END;
       ELSE
        DO;
          BCD=BCD_IN_ADVANCE(MARK);
          TOKEN=TOKEN_IN_ADVANCE(MARK);
        END;

      ATTEMPTED=FALSE;
      SPIT_CARD=TRUE;
      CALL GET_CARD;
      SPIT_CARD=FALSE;
      BUF_PTR=-1;
      BUF_PTR_LMT=0;
    END;

     /* RESET THE FLAGS FOR NEXT TIME */
                SEM_ANAL=0;
                SEM_CHECK=FALSE;
                REPLACED=FALSE;
                DELETED=FALSE;
                INSERTED=FALSE;
                INTERCHANGED=FALSE;
                FINISHED_TRYING=FALSE;

END CORRECTION_DECISION;
```

```
/*************************************************************/
/**********                                        **********/
/**********              UNMATCHED_END              **********/
/**********                                        **********/
/*************************************************************/

UNMATCHED_END:
 PROCEDURE;
  /* AD HOC CORRECTION FOR UN MATCHED  END ; IS TO DELETETE END ; */
    ATTEMPTED=TRUE;
    CALL SCAN;
    BCD_IN_ADVANCE(0)='END';
    BCD_IN_ADVANCE(1)=';';
    BCD_IN_ADVANCE(2)=BCD;
    TOKEN_IN_ADVANCE(2)=TOKEN;
    MARK=2;
    BUF_PTR_LMT=-2;
    BUF_PTR=-1;
    CALL CORRECTION_DECISION;
 END UNMATCHED_END;


/*************************************************************/
/**********                                        **********/
/**********              CONTEXT_CHECK              **********/
/**********                                        **********/
/*************************************************************/

CONTEXT_CHECK:
 PROCEDURE;  /* CHECK IF PARSE STACK HAS PROPER FORM  */

IF FINISHED_TRYING THEN
   DO;
      CALL CORRECTION_DECISION;
      RETURN;
   END;

I=SP;

 /* CHECK THE LEFT CONTEXT */
DO WHILE STOP2(PARSE_STACK(I));
   I =I -1;
END;

IF I¬=0 THEN
   DO;    /* IN CASE LEFT CONTEXT IS NOT SATISFIED */
       IF CONTROL(BYTE('Q')) THEN
         DO;
           CALL REPORT_CORRECTION;
           OUTPUT=
              '......IMPROPER STACK CONFIGURATION- CORRECTION REJECTED';
           CALL STACK_DUMP;
         END;
       IF SEM_CHECK THEN
          DO;
             SEM_CHECK=FALSE;
             BUF_PTR=SAVE_PTR;
          END;
```

```
      IF ATTEMPTED THEN
         CALL TRY_AGAIN;
   END;
ELSE
   IF ¬SEM_CHECK THEN
      DO;  /* IF LEFT CONTEXT IS OK PREPARE FOR SEMANTIC CHECK */
         SP=LAST_SP;
         DO I=2 TO SP;
            PARSE_STACK(I)=LAST_STACK(I);
         END;

         IF ATTEMPTED & CONTROL(BYTE('Q')) THEN
            CALL REPORT_CORRECTION;
         BUF_LMT=BUF_PTR+1;
         BUFFOR(BUF_LMT)=TOKEN;
         BCD_BUF(BUF_LMT)=BCD;
         SEM_CHECK=TRUE;
         TOKEN=BUFFOR(0);
         BCD=BCD_BUF(0);
         BUF_PTR=0;
         RETURN;
      END;
   ELSE
      DO;  /* RETURNED FROM SEMANTIC CHECK        */
          SEM_ANAL=SEM_ANAL+1;
          SEM_CHECK=FALSE;
          IF ATTEMPTED THEN
            DO;   /* IN ERROR CORRECTION MODE */
               IF SEMANTICS THEN
                  DO;
                      CALL SAVE_CORRECTION;
                      CORR_SP=SP;
                      DO I=2 TO SP;
                         CORR_STACK(I)=PARSE_STACK(I);
                      END;
                  END;
               ELSE
                IF CONTROL(BYTE('Q')) THEN
                  OUTPUT=
                   '.... CORRECTION REJECTED FOR SEMANTIC REASONS';
               BUF_PTR=SAVE_PTR;
               CALL TRY_AGAIN;
               RETURN;
            END;
         ELSE
            DO;   /* IN STANDARD MODE  */
               SEM_ANAL=0;
               TOKEN=BUFFOR(BUF_LMT);
               BCD=BCD_BUF(BUF_LMT);
               BUF_PTR=-1;
               LAST_SP=SP;
               DO I=2 TO SP;
                  LAST_STACK(I)=PARSE_STACK(I);
               END;
            END;
      END;
  END CONTEXT_CHECK;
```

```
/*******************************************************************/
/**********                                              **********/
/**********                  RESTACK                     **********/
/**********                                              **********/
/*******************************************************************/

RESTACK:
 PROCEDURE;  /* RESTACK THE PARSE STACK TO ONE SYMBOL BEFORE LAST TIME */

  IF BUF_PTR_LMT>BUF_PTR THEN
     DO;
          BUF_PTR=BUF_PTR+1;
          TOKEN=BUFFOR(BUF_PTR);
          RETURN;
     END;



  SAVE_PTR=BUF_PTR;
  SAVE_SP=SP;
  DO I=2 TO SP;
    SAVE_STACK(I)=PARSE_STACK(I);
  END;

  MARK=MARK+1;

  DO I=0 TO MARK-1;
     TOKEN_IN_ADVANCE(MARK-I)=TOKEN_IN_ADVANCE(MARK-I-1);
     BCD_IN_ADVANCE(MARK-I)=BCD_IN_ADVANCE(MARK-I-1);
  END;

  TOKEN_IN_ADVANCE(0)=SAVE_TOKEN;
  BCD_IN_ADVANCE(0)=SAVE_BCD;
  SAVE_TOKEN=BUFFOR(BUF_PTR_LMT+1);
  SAVE_BCD=BCD_BUF(BUF_PTR_LMT+1);

  LAST_INDEX=-1;
  TRY_INDEX=-1;
  SHIFT=0;
  C1_LINE=C1(PARSE_STACK(SP));
  BITE=BYTE(C1_LINE);

  FINISHED_TRYING=FALSE;
  RESTACKING=FALSE;
  INSERTED=FALSE;
  REPLACED=FALSE;
  DELETED=FALSE;
  INTERCHANGED=FALSE;

  CALL TRY_AGAIN;

END RESTACK;
```

```
   /*****************************************************************/
   /**********                                           **********/
   /**********               SCOOP_OR_SCAN               **********/
   /**********                                           **********/
   /*****************************************************************/

SCOOP_OR_SCAN:
 PROCEDURE;

   /*  IN STANDARD MODE JUST SCAN NEXT SYMBOL */

   IF ¬ATTEMPTED THEN
     CALL SCAN;

   ELSE  /* IN ERROR CORRECTION MODE GET THE NEXT TOKEN THAT IS ALREADY
                                                 READ IN  */
     IF SUCCESS_PTR=0 THEN

        DO;
          SUCCESS_PTR=1;
          TOKEN=SAVE_TOKEN;
          BCD=SAVE_BCD;
        END;
     ELSE
        DO;
          BCD=BCD_IN_ADVANCE(SUCCESS_PTR-1);
          TOKEN=TOKEN_IN_ADVANCE(SUCCESS_PTR-1);
          SUCCESS_PTR=SUCCESS_PTR+1;
        END;

END SCOOP_OR_SCAN;
```

```
/*********************************************************************/
/**********                                               **********/
/**********                    PARSE                      **********/
/**********                                               **********/
/*********************************************************************/

PARSE:
 PROCEDURE;

DO WHILE COMPILING;
 DO CASE STACKING;

    /************* CASE  0  ****************************/

    IF TOKEN=EOFILE THEN
      DO;
        OUTPUT='******** EOF FOUND ';
        OUTPUT='ACTION -------- END OF PROGRAM ASSUMED AND '
        ||' COMPILATION ABORTED ----------';
        RETURN;
      END;
    ELSE
      DO;
        IF END_OF_STMT &  V(TRY_INDEX)¬=';' & V(TRY_INDEX)¬='THEN' THEN
          DO;
            END_OF_STMT=FALSE;
            CALL CONTEXT_CHECK;
          END;
        ELSE
          DO;
            IF ¬ATTEMPTED THEN
              DO;
                CALL ERROR('ILLEGAL SYMBOL PAIR: '||V(PARSE_STACK(SP))
                ||X1||V(TOKEN),1);
                CALL STORE_INFORMATION;
              END;
            IF ¬FINISHED_TRYING THEN
              CALL TRY_AGAIN;
            ELSE
              CALL CORRECTION_DECISION;
          END;
      END;
```

```
DO;  /*****  CASE 1   ********/

  IF ATTEMPTED & ¬SEM_CHECK & SUCCESS_PTR=MARK+2
     | ATTEMPTED & SEM_CHECK & BUF_PTR=BUF_LMT
     | ¬ATTEMPTED & END_OF_STMT  THEN
    DO;
     END_OF_STMT=FALSE;
     CALL CONTEXT_CHECK;
    END;
  ELSE
      DO;
         IF TOKEN=SEMICOLON | BCD='THEN' THEN
            END_OF_STMT=TRUE;
         IF SEM_CHECK THEN
           DO;
             SP=SP+1;
             IF SP=STACKSIZE THEN
               DO;
                 CALL ERROR('STACK OVERFLOW***COMPILATION ABORTED***'
                                                             ,2);
                 RETURN; /* THUS ABORTING COMPILATION. */
               END;
             PARSE_STACK(SP)=TOKEN;
             VAR(SP)=BCD;
             FIXV(SP)=NUMBER_VALUE;
             FIXL(SP)=CARDS-1;
             BUF_PTR=BUF_PTR+1;
             TOKEN=BUFFOR(BUF_PTR);
             BCD=BCD_BUF(BUF_PTR);
           END;
         ELSE
           DO;
             IF ¬(RESTACKING & BUF_PTR_LMT<0) THEN
               DO;
                 SP=SP+1;
                 IF SP=STACKSIZE THEN
                   DO;
                     CALL ERROR(
                     'STACK OVERFLOW***COMPILATION ABORTED***',2);
                     RETURN;     /* THUS ABORTING COMPILATION. */
                   END;
                 PARSE_STACK(SP)=TOKEN;
               END;
             IF ATTEMPTED THEN
               STACK_MESSED_UP=TRUE;
             IF RESTACKING THEN
               CALL RESTACK;
             ELSE
               DO;
                  BUF_PTR=BUF_PTR+1;
                  BUFFOR(BUF_PTR)=TOKEN;
                  BCD_BUF(BUF_PTR)=BCD;
                  CALL SCOOP_OR_SCAN;
               END;
           END;
      END;
 END;   /*   END OF  CASE 1   */
```

```
     /********    CASE 2    *********/

    IF ¬REDUCE THEN
      DO;
        IF END_OF_STMT &  V(TRY_INDEX)¬=';' & V(TRY_INDEX)¬='THEN' THEN
          DO;
            END_OF_STMT=FALSE;
            CALL CONTEXT_CHECK;
          END;
        ELSE
         DO;
            IF V(PARSE_STACK(SP))='<ENDING>' & TOKEN=SEMICOLON &
               ¬ATTEMPTED THEN /* AD HOC CORRECTION FOR UN MATCHED END; */
              CALL UNMATCHED_END;
            ELSE
             DO;
            IF ¬ATTEMPTED THEN
              DO;
                CALL ERROR('NO PRODUCTION APPLICABLE',1);
                CALL STORE_INFORMATION;
              END;
            IF ¬FINISHED_TRYING THEN
              CALL TRY_AGAIN;
            ELSE
              CALL CORRECTION_DECISION;
             END;
          END;
      END;
    ELSE
       DO;
          STACK_MESSED_UP=TRUE;
       END;

   END; /********  OF CASE STACKING  ***********/
 END; /********    OF DO WHILE COMPILING   *******/
 END PARSE;
```

```
MAIN_PROCEDURE:
  PROCEDURE;
     CALL INITIALIZATION;
     CLOCK(1) = TIME;
     COMPILING=TRUE;
     SEMANTICS=TRUE;
     BUF_PTR=-1;
     CALL PARSE;
     IF ATTEMPTED THEN
        OUTPUT=' LAST ERROR WAS NOT CORRECTED';
     ELSE
        OUTPUT=' DONE WITH COMPILING';
     CLOCK(2) = TIME;
     /* CLOCK(3) GETS SET IN PRINT_SUMMARY */
     CALL PRINT_SUMMARY;
  END MAIN_PROCEDURE;


CALL MAIN_PROCEDURE;
     CLOCK(0) = TIME;  /* KEEP TRACK OF TIME IN EXECUTION */
 RETURN SEVERE_ERRORS;
OF EOF EOF
```

APPENDIX C

SAMPLE EXPL RUNS

```
     ************************************************************************
     *                                                                      *
     *                                                                      *
     *            X P L   SYNTAX  ANALYSIS  AND  ERROR CORRECTION           *
     *                                                                      *
     *                                                                      *
     ************************************************************************


TODAY IS AUGUST 7, 1973.  CLOCK TIME = 15:48:5.50.


   1 |   DECLARE A FIXED;
   2 |  DECLARE CARDS(100)  CHARACTER, (I,J)K) FIXED, TEMP CHARACTER;
                                                |
*** ERROR, ILLEGAL SYMBOL PAIR: ) <IDENTIFIER>
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 0.  ***
PARTIAL PARSE TO THIS POINT IS:
     <STATEMENT LIST> <DECLARATION STATEMENT> , <IDENTIFIER LIST> <IDENTIFIER> )


ACTION------: ) WAS REPLACED BY , IN LINE NO. 2
 CORRECTED STATEMENT --- DECLARE CARDS ( 100 ) CHARACTER , ( I , J , K ) FIXED , TEMP
     CHARACTER ;
   3 |  DELCARE (P,Q) FIXED;
                          |
*** ERROR, ILLEGAL SYMBOL PAIR: <VARIABLE> FIXED
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 2.  ***
PARTIAL PARSE TO THIS POINT IS:
     <STATEMENT LIST> <VARIABLE>


ACTION------: DELCARE WAS REPLACED BY DECLARE IN LINE NO. 3
 CORRECTED STATEMENT --- DECLARE ( P , Q ) FIXED ;
   4 |  DECLARE AA(2) FIXED;
   5 |  DO I=1 , 5;  END;
                 |
*** ERROR, NO PRODUCTION APPLICABLE
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 3.  ***
PARTIAL PARSE TO THIS POINT IS:
     <STATEMENT LIST> DO <VARIABLE> <REPLACE> <EXPRESSION> ,


ACTION------: , WAS REPLACED BY TO IN LINE NO. 5
 CORRECTED STATEMENT --- DO I = 1 TO 5 ;
   6 |  A:   AA(2);
                  |
*** ERROR, NO PRODUCTION APPLICABLE
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 5.  ***
PARTIAL PARSE TO THIS POINT IS:
     <STATEMENT LIST> <LABEL DEFINITION> <EXPRESSION>


ACTION------: : WAS REPLACED BY = IN LINE NO. 6
 CORRECTED STATEMENT --- A = AA ( 2 ) ;
   7 |  RETURN;
   8 |EOF OEF
 DONE WITH COMPILING
END OF CHECKING AUGUST 7, 1973.  CLOCK TIME = 15:48:10.61.

8 CARDS WERE CHECKED.
```

```
4 ERRORS (4 SEVERE) WERE DETECTED.
THE LAST DETECTED ERROR WAS ON LINE 6.


SYMBOL  TABLE  DUMP


A                : FIXED     DECLARED ON LINE 1 AND REFERENCED 1 TIMES.
AA               : ARRAY     DECLARED ON LINE 4 AND REFERENCED 3 TIMES.
CARDS            : ARRAY     DECLARED ON LINE 2 AND REFERENCED 0 TIMES.
I                : FIXED     DECLARED ON LINE 2 AND REFERENCED 1 TIMES.
J                : FIXED     DECLARED ON LINE 2 AND REFERENCED 0 TIMES.
K                : FIXED     DECLARED ON LINE 2 AND REFERENCED 0 TIMES.
P                : FIXED     DECLARED ON LINE 3 AND REFERENCED 0 TIMES.
Q                : FIXED     DECLARED ON LINE 3 AND REFERENCED 0 TIMES.
TEMP             : CHARACTER DECLARED ON LINE 2 AND REFERENCED 0 TIMES.


TOTAL TIME IN CHECKER    15:48:10.80.
SET UP TIME              15:48:5.65.
ACTUAL CHECKING TIME     0:0:4.96.
CLEAN-UP TIME AT END     0:0:0.19.
CHECKING RATE: 96 CARDS PER MINUTE.
```

```
        TEST: PROCEDURE OPTIONS(MAIN);


STMT LEVEL NEST
  1                   TEST: PROCEDURE OPTIONS(MAIN);
  2      1              DECLARE A FIXED;
  3      1             DECLARE CARDS(100)  CHARACTER, (I,J)K) FIXED,
                                   TEMP CHARACTER;
  4      1             DELCARE (P,Q) FIXED;
  5      1             DECLARE AA(2) FIXED;
  6      1             DO I=1  , 5;  END;
  8      1             A:   AA(2);
  9      1             RETURN;
 10      1              END TEST;
```

```
        TEST: PRCCEDURE OPTIONS(MAIN);

   COMPILER DIAGNOSTICS.


 SEVERE ERRORS.

    IEM0673I      4   INVALID USE OF FUNCTION NAME ON LEFT HAND SIDE CF

                      OPTION IN STATEMENT NUMBER 4

    IEM0725I      4   STATEMENT NUMBER 4 HAS BEEN DELETED DUE TC A SEVER

    IEM0124I      3   INVALID ATTRIEUTE IN CECLARE OR ALLOCATE STATEMENT

    IEM0031I      8   OPERAND MISSING IN CR FCLLOWING STATEMENT NUMBER 8

    IEM0128I      3   LENGTH OF BIT OR CHARACTER STRING MISSING IN STATE

    IEM0152I      3   TEXT BEGINNING 'K)FIXEC' IN STATEMENT NUMBER 3 HAS

    IEM0128I      3   LENGTH OF BIT OR CHARACTER STRING MISSING IN STATE


 ERRORS.

    IEM0080I      4   EQUAL SYMBOL HAS BEEN INSERTED IN ASSIGNMENT STATE

    IEM0080I      8   EQUAL SYMBOL HAS BEEN INSERTEC IN ASSIGNMENT STATE

    IEM0557I      2   THE MULTIPLE DECLARATICN CF IDENTIFIER 'A' IN STAT


 WARNINGS.

    IEM0764I          ONE OR MORE FIXED BINARY ITEMS CF PRECISICN 15 CR

                      ARE FLAGGED '*********' IN THE XREF/ATR LIST.
```

NAME ON LEFT HAND SIDE CF ECUAL SYMBOL, OR IN REPLY KEYTO OR STRING

BER 4

BEEN DELETED DUE TC A SEVERE ERROR NOTED ELSEWHERE.

CLARE OR ALLOCATE STATEMENT NUMBER 3 . ATTRIBUTE TEXT DELETED.

CLLOWING STATEMENT NUMBER 8 . DUMMY OPERAND INSERTED.

TER STRING MISSING IN STATEMENT NUMBER 3 . LENGTH 1 INSERTED.

:' IN STATEMENT NUMBER 3 HAS BEEN DELETED.

TER STRING MISSING IN STATEMENT NUMBER 3 . LENGTH 1 INSERTED.

NSERTED IN ASSIGNMENT STATEMENT NUMBER 4

INSERTED IN ASSIGNMENT STATEMENT NUMBER 8

:N CF IDENTIFIER 'A' IN STATEMENT NUMBER 2 HAS BEEN IGNORED.

RY ITEMS CF PRECISICN 15 CR LESS HAVE BEEN GIVEN HALFWORD STORAGE. THEY

' IN THE XREF/ATR LIST.

```
*****************************************************************************
*                                                                           *
*                                                                           *
*            X P L   SYNTAX  ANALYSIS  AND  ERROR CORRECTION                *
*                                                                           *
*                                                                           *
*****************************************************************************


TODAY IS SEPTEMBER 5, 1973.  CLOCK TIME = 16:31:43.00.


    1 | /* THIS PROGRAM READS N CARDS (N=10), SORTS THEM IN ALPHABETICAL
    2 |    (COLLATING) ORDER, AND PRINTS THEM. */
    3 |  DECLARE N LITERALLY '10';
    4 | DECLARE CARDS (N) CHARACTER, (I,L,K) FIXED, TEMP CHARACTER;
    5 |
    6 |  OUTPUT = 'INPUT CARDS:';
    7 |    DO  I=1 , N;
                    |
*** ERROR, NO PRODUCTION APPLICABLE
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 0.  ***
PARTIAL PARSE TO THIS POINT IS:
     <STATEMENT LIST> DO <VARIABLE> <REPLACE> <EXPRESSION> ,


ACTION------: , WAS REPLACED BY TO IN LINE NO. 7
 CORRECTED STATEMENT --- DO I = 1 TO 10 ;
    8 |    OUTPUT, CARDS(I) = INPUT;   /* READ AND LIST */
    9 |  DECLARE  Y FIXD;
                      |
*** ERROR, ILLEGAL SYMBOL PAIR: <IDENTIFIER> <IDENTIFIER>
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 7.  ***
PARTIAL PARSE TO THIS POINT IS:
     <STATEMENT LIST> <GROUP HEAD> DECLARE <IDENTIFIER>


ACTION------: FIXD WAS REPLACED BY FIXED IN LINE NO. 9
 CORRECTED STATEMENT --- DECLARE Y FIXED ;
   10 |  END;
   11 |
   12 |  K,L= N;
   13 | DO WHILE K << L;
                   |
*** ERROR, ILLEGAL SYMBOL PAIR: < <
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 9.  ***
PARTIAL PARSE TO THIS POINT IS:
     <STATEMENT LIST> DO WHILE <STRING EXPRESSION> <
 *********** NO UNIQUE CORRECTION
CORRECTION NO. 1     < DELETED
CORRECTION NO. 2     < REPLACED BY =
 CORRECTED STATEMENT --- DO WHILE K < L ;
   14 |     L= -N;
   15 |     DO I=1 TO K;
   16 |       L=I-1;
   17 |         IF CARDS(L) > CARDS(I) THEN
   18 |           DO;
   19 |             TEMP=CARDS(L);
   20 |             CARDS(L)=CARDS(I);
   21 |             CARDS(I)=TEMP;
   22 |             K=L;
```

```
  23 |          END;
  24 |       END;
  25 |   END;                    /* OF SORT LOOP */
  26 |    IF   ;
              |
*** ERROR, ILLEGAL SYMBOL PAIR: IF ;
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 13.   ***
PARTIAL PARSE TO THIS POINT IS:
     <STATEMENT LIST> IF
 ************ NO UNIQUE CORRECTION
CORRECTION NO. 1     IF DELETED
CORRECTION NO. 2     IF REPLACED BY RETURN
CORRECTION NO. 3     IF REPLACED BY DO
 CORRECTED STATEMENT --- ;
  27 |
  28 |   OUTPUT= 'SORTED CARDS:';
  29 |   DO I=1 TO N;
  30 |     OUTPUT =CARDS(I);
  31 |   END;
  32 |EOF  EOF
 DONE WITH COMPILING
END OF CHECKING SEPTEMBER 5, 1973.  CLOCK TIME = 16:31:47.68.

32 CARDS WERE CHECKED.
4 ERRORS (4 SEVERE) WERE DETECTED.
THE LAST DETECTED ERROR WAS ON LINE 26.


SYMBOL  TABLE  DUMP


CARDS          : ARRAY     DECLARED ON LINE 5 AND REFERENCED 8 TIMES.
I              : FIXED     DECLARED ON LINE 5 AND REFERENCED 9 TIMES.
K              : FIXED     DECLARED ON LINE 5 AND REFERENCED 4 TIMES.
L              : FIXED     DECLARED ON LINE 5 AND REFERENCED 8 TIMES.
TEMP           : CHARACTER DECLARED ON LINE 5 AND REFERENCED 2 TIMES.
Y              : FIXED     DECLARED ON LINE 9 AND REFERENCED 0 TIMES.


TOTAL TIME IN CHECKER      16:31:47.79.
SET UP TIME                16:31:43.35.
ACTUAL CHECKING TIME       0:0:4.33.
CLEAN-UP TIME AT END       0:0:0.11.
CHECKING RATE: 443 CARDS PER MINUTE.
```

```
*************************************************************************************
*                                                                                   *
*                                                                                   *
*              X P L   SYNTAX  ANALYSIS  AND  ERROR CORRECTION                      *
*                                                                                   *
*                                                                                   *
*************************************************************************************

TODAY IS AUGUST 7, 1973.  CLOCK TIME = 15:46:16.16.

   1 |  DECLARE (A,B,C,) FIXED;
                        |
*** ERROR, ILLEGAL SYMBOL PAIR: , )
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 0.  ***
PARTIAL PARSE TO THIS POINT IS:
     DECLARE <IDENTIFIER LIST> <IDENTIFIER> ,


ACTION------: , WAS DELETED IN LINE NO. 1
 CORRECTED STATEMENT --- DECLARE ( A , B , C ) FIXED ;
   2 |   A=/ B +1;
           |
*** ERROR, ILLEGAL SYMBOL PAIR: = /
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 1.  ***
PARTIAL PARSE TO THIS POINT IS:
     <STATEMENT LIST> <VARIABLE> =
 ************ NO UNIQUE CORRECTION
CORRECTION NO. 1     / DELETED
CORRECTION NO. 2     <NUMBER> INSERTED BEFORE /
CORRECTION NO. 3     <STRING> INSERTED BEFORE /
 CORRECTED STATEMENT --- A = B + 1 ;
   3 |  DECLARE I FIXD;
                  |
*** ERROR, ILLEGAL SYMBOL PAIR: <IDENTIFIER> <IDENTIFIER>
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 2.  ***
PARTIAL PARSE TO THIS POINT IS:
     <STATEMENT LIST> DECLARE <IDENTIFIER>


ACTION------: FIXD WAS REPLACED BY FIXED IN LINE NO. 3
 CORRECTED STATEMENT --- DECLARE I FIXED ;
   4 |  IFF 1 THEN ;
            |
*** ERROR, ILLEGAL SYMBOL PAIR: <IDENTIFIER> <NUMBER>
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 3.  ***
PARTIAL PARSE TO THIS POINT IS:
     <STATEMENT LIST> <IDENTIFIER>


ACTION------: IFF WAS REPLACED BY IF IN LINE NO. 4
 CORRECTED STATEMENT --- IF 1 THEN
   5 |  IF B,C THEN ; ELSE  B;
            |
*** ERROR, NO PRODUCTION APPLICABLE
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 4.  ***
PARTIAL PARSE TO THIS POINT IS:
     <STATEMENT LIST> IF <EXPRESSION> ,
```

```
ACTION------: , WAS REPLACED BY = IN LINE NO. 5
 CORRECTED STATEMENT --- IF B = C THEN
                                 |
*** ERROR, NO PRODUCTION APPLICABLE
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 5.  ***
PARTIAL PARSE TO THIS POINT IS:
      <STATEMENT LIST> <IF CLAUSE> <TRUE PART> <EXPRESSION>
 ************ NO UNIQUE CORRECTION
CORRECTION NO. 1     B DELETED
CORRECTION NO. 2     B REPLACED BY DO
CORRECTION NO. 3     B REPLACED BY ;
CORRECTION NO. 4     RETURN INSERTED BEFORE B
CORRECTION NO. 5     CALL INSERTED BEFORE B
 CORRECTED STATEMENT --- ELSE
   6 |    DECLARE F FIXD, K FIXED;
                       |
*** ERROR, ILLEGAL SYMBOL PAIR: <IDENTIFIER> <IDENTIFIER>
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 5.  ***
PARTIAL PARSE TO THIS POINT IS:
      <STATEMENT LIST> DECLARE <IDENTIFIER>


ACTION------: FIXD WAS REPLACED BY FIXED IN LINE NO. 6
 CORRECTED STATEMENT --- DECLARE F FIXED , K FIXED ;
   7 |    A A , ,;
            |
*** ERROR, ILLEGAL SYMBOL PAIR: <IDENTIFIER> <IDENTIFIER>
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 6.  ***
PARTIAL PARSE TO THIS POINT IS:
      <STATEMENT LIST> <IDENTIFIER>
ACTION------   A A , , ;  ------ ABORTED
PARSING RESUMES IN STANDARD MODE
   8 |  PROCEDURE:     AA;
                |
*** ERROR, ILLEGAL SYMBOL PAIR: <STATEMENT LIST> PROCEDURE
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 7.  ***
PARTIAL PARSE TO THIS POINT IS:
      <STATEMENT LIST>
ACTION------   PROCEDURE : AA ;  ------ ABORTED
PARSING RESUMES IN STANDARD MODE
   9 |   DECLARE AA(4) FIXED;
  10 |  AA(1 =)  2;
             |
*** ERROR, ILLEGAL SYMBOL PAIR: = )
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 8.  ***
PARTIAL PARSE TO THIS POINT IS:
      <STATEMENT LIST> <SUBSCRIPT HEAD> <STRING EXPRESSION> =


ACTION------: ) AND = ARE INTERCHANGED  IN LINE NO.10
 CORRECTED STATEMENT --- AA ( 1 ) = 2 ;
  11 |  END;
ACTION------   END ;  ------ ABORTED
PARSING RESUMES IN STANDARD MODE
  12 |   A  AA =(1);
             |
*** ERROR, ILLEGAL SYMBOL PAIR: <IDENTIFIER> <IDENTIFIER>
    LAST PREVIOUS ERROR WAS DETECTED ON LINE 10.  ***
PARTIAL PARSE TO THIS POINT IS:
      <STATEMENT LIST> <IDENTIFIER>
 ************ NO UNIQUE CORRECTION
CORRECTION NO. 1     AA AND = INTERCHANGED
CORRECTION NO. 2     AA DELETED
 CORRECTED STATEMENT --- A = AA ( 1 ) ;
  13 |  RETURN;
  14 |EOF  EOF
```

```
DONE WITH COMPILING
END OF CHECKING AUGUST 7, 1973.  CLOCK TIME = 15:46:21.48.

14 CARDS WERE CHECKED.
11 ERRORS (11 SEVERE) WERE DETECTED.
THE LAST DETECTED ERROR WAS ON LINE 12.


SYMBOL  TABLE  DUMP


A               : FIXED      DECLARED ON LINE 1 AND REFERENCED 2 TIMES.
AA              : ARRAY      DECLARED ON LINE 9 AND REFERENCED 2 TIMES.
B               : FIXED      DECLARED ON LINE 1 AND REFERENCED 3 TIMES.
C               : FIXED      DECLARED ON LINE 1 AND REFERENCED 1 TIMES.
F               : FIXED      DECLARED ON LINE 6 AND REFERENCED 0 TIMES.
I               : FIXED      DECLARED ON LINE 3 AND REFERENCED 0 TIMES.
K               : FIXED      DECLARED ON LINE 6 AND REFERENCED 0 TIMES.

TOTAL TIME IN CHECKER     15:46:21.58.
SET UP TIME               15:46:16.31.
ACTUAL CHECKING TIME      0:0:5.17.
CLEAN-UP TIME AT END      0:0:0.10.
CHECKING RATE: 162 CARDS PER MINUTE.
```