ERROR RECOVERING PARSERS

THE GENERATION OF ERROR RECOVERING SIMPLE PRECEDENCE PARSERS

by

CLIVE B. JOHNS, B.Sc., M.Sc.

A Project

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

July 1974

MASTER OF SCIENCE (1974)
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: The Generation of Error Recovering Simple Precedence Parsers.

AUTHOR: Clive B. Johns, B.Sc. (Leeds), M.Sc. (Southampton)

SUPERVISOR: Professor D. Wood

NUMBER OF PAGES: viii, 282

ABSTRACT

A survey of error detection and error recovery in simple precedence parsers is given, and in particular the error detection of Leinius, and the error recovery of Rhodes are described. An implementation of a system which generates error recovering simple precedence parsers is presented. The system divides into two subsystems, a constructor, and a skeletal parser. The grammar of the simple precedence language (L) for which a parser is required is used as input to the constructor. The constructor processes the grammar, and produces data used to prime the skeletal parser such that it can parse a sentence of L (or, an L program). The skeletal parser includes error detection based on the work of Leinius, and error recovery based on the work of Rhodes.

ACKNOWLEDGEMENTS

I wish to thank my supervisor Dr. D. Wood for his guidance and assistance during the work on this project. For advice during the implementation I would also like to thank C. Bryce, Dr. N. Solntseff, and A. Walker. An informal acknowledgement is due to R. P. Leinius and S. P. Rhodes. The error detection and recovery in this project is based on their work.

Finally, I would like to thank Mrs. Margaret Belec for her excellent typing.

TABLE OF CONTENTS

L I S T O F F I G U R E S

L I S T O F T A B L E S

CHAPTER 1

INTRODUCTION

1.1. Overview of the Project

The objectives of the project are to:

i) Investigate simple precedence parsing techniques.

ii) Investigate error detection and recovery for simple precedence parsers.

iii) Implement a system to generate error recovering simple precedence parsers.

This project is part of a major investigation of compiler techniques. Topics for future projects are for example (1) error detection/recovery in LR(k) and other kinds of parsers; (2) program optimisation; (3) register allocation.

Error detection and recovery in simple precedence parsers has been described by Leinius (1970) and Rhodes (1973). A summary of Leinius' error recovering system, and a detailed survey of Rhodes' error recovering system are given in Chapter 3. Implementation, described in Chapter 4, uses the concept of a constructor described in Gries (1971). The implementation of the basic simple precedence parsing algorithm and error detection is based on Leinius (1970) and the implementation of error recovery is based on the work of Rhodes (1973).

1.2. Overview of Error Detection and Recovery

Good error detection and recovery in a compiler are important. Most of the programs (or "sentences") submitted to a compiler will be incorrect, many of these will be syntactically incorrect.

The parser designer is faced with the problem of building a parser that can (a) detect a syntactic error; (b) do something about it, preferably so that parsing can be continued (we want to discover as many errors as possible each run). The programmer is faced with only one problem -- how to correct his program. For this he requires:

(1) No spurious error messages

(2) Meaningful error messages.

The parser cannot be designed to detect and recover from semantic or logical errors.

We use the term "recovery" and not "correction". There are differing opinions concerning the meaning of these two terms, Rhodes (1973). We favour the following based on Gries (1971):

> Error recovery is action designed to keep the parser going. After recovery the program is usually not executed. Descriptions of error recovering systems can be found in Leinius (1970), LaFrance (1971), James (1972), and Rhodes (1973).

Error correction is an attempt to produce a correct program. This may be done using a minimum distance approach (from badly formed sentence to "closest" well formed sentence). Descriptions of error correcting systems can be found in Hopcroft and Ullman (1966), Levy (1971), Aho and Peterson (1972), and Lyon (1973).

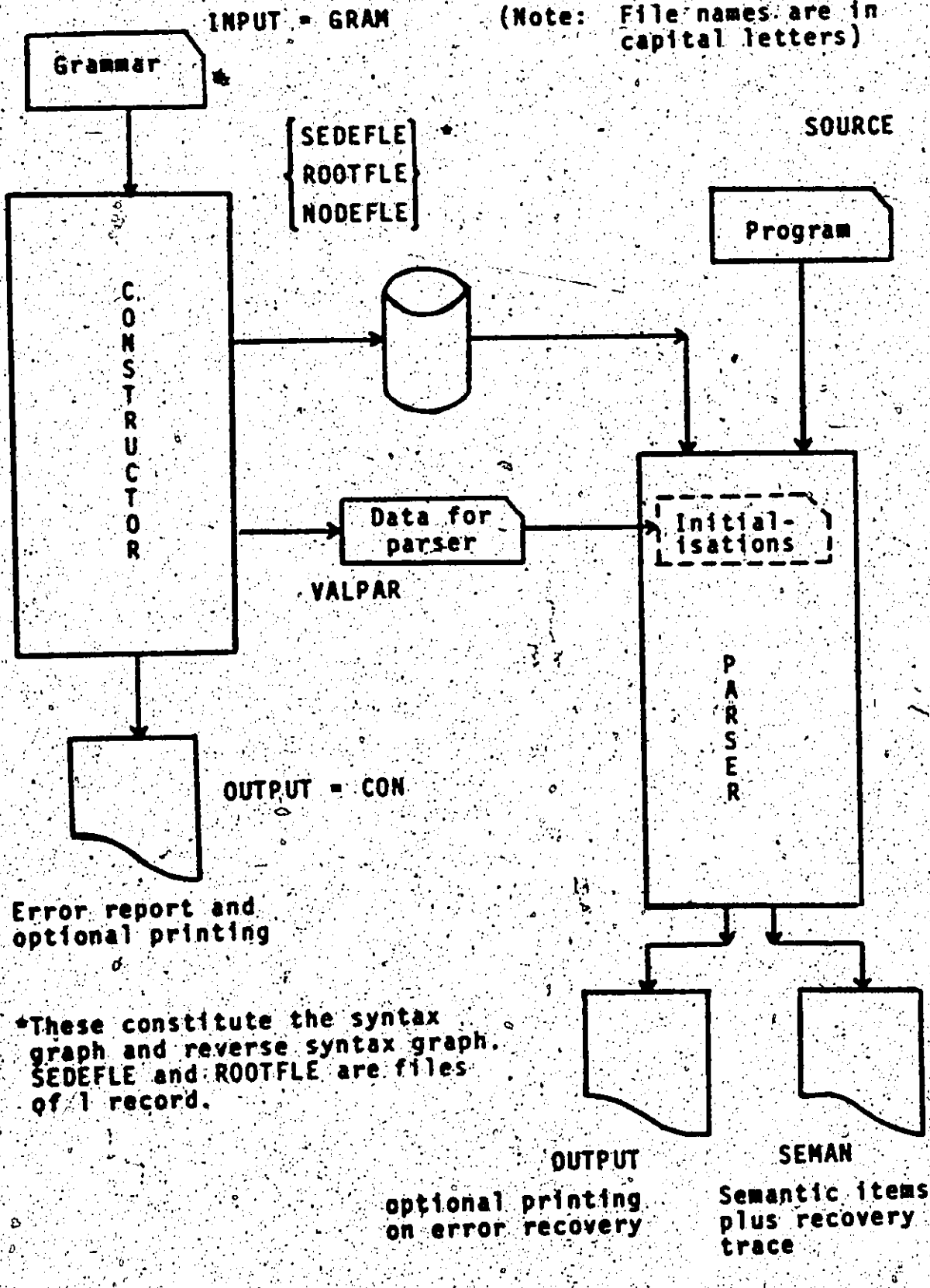
Recovery in the systems described by Rhodes and Leinius is effected through stack manipulation such that parsing can continue. A minimum distance approach may be taken when the stack is being manipulated. The minimum distance is between "sentential forms" not sentences, Rhodes (1973). It is not useful to consider this as correction.

1.3. Implementation Overview

The implementation consists of building a constructor, Gries (1971), and a parser -- see Figure 1.1.. The programming language for which we require the parser is specified in terms of a simple precedence grammar (for definition see section 2.3.) The constructor takes the grammar as input, checks that it is uniquely invertible, free of simple precedence conflicts, and has no e-rules, and outputs data (tables on cards, graphs on sequential files) that are used to prime the skeletal parser. The

FIGURE 1.1

IMPLEMENTATION OVERVIEW



skeletal parser incorporates routines used for error detection and recovery. Barnes (1973) may be of interest to the reader.

The programs are written in Pascal for a CDC6400 machine, Wirth (1971).

1.4. Notation and Abbreviations

The notation follows that in the book by Aho and Ullman (1972).

Symbols:	a, b, c, d	terminal
	0, 1 ... 9	
	A, B, C, D, S	nonterminal S is the start symbol
	U, V ... Z	terminal or nonterminal
Words:	α, β, γ ...	may contain either terminal or nonterminal symbols.
	u, v, w ... z	just terminal symbols

The following abbreviations are used:

CFG	Context free grammar
LSC	Locally syntactically correct
LHS	Left hand side
RHS	Right hand side
NT	Nonterminal

CHAPTER 2

THEORETICAL BASE

This chapter is based on the book by Aho and Ullman (1972) and to a lesser extent on the book by Gries (1971). The string operators in section 2.4. are taken from Rhodes (1973).

2.1. Generators and Recognisers

There are two principal methods for defining languages -- the generator and the recogniser.

Define a language L as a set of words over some finite alphabet Σ. If there are a finite number of words, and each word is of finite length, then we can specify L by listing all the words. However, for many languages it is not possible to put an upper bound on the length of the longest word in the language. These languages will contain an arbitrary number of words. Such a language cannot be specified by enumeration of its words; we have to look for an alternative method. We require our specification of the language to be finite.

One method is to use a generative system called a grammar. Each sentence in the language can be constructed by well defined methods using the rules (or productions) of the grammar. A formal treatment of grammars is given in

section 2.2.

A second method is to use a procedure which when presented with an arbitrary input word will halt and answer "yes" after a finite amount of computation if that word is in the language. This device is called a recogniser. We do not give a formal treatment of a recogniser.

The constructor (section 1.3) is a way of using the grammar for a language to build a recogniser for sentences of the language. (More strictly we build a transducer, which is a recogniser with an output stream.)

2.2. Grammars and Languages

A context free grammar (CFG) is a 4-tuple $G = (N,\Sigma,P,S)$ where:

(1) N is a finite set of nonterminal symbols (sometimes called syntactic entities).

(2) Σ is a finite set of terminal symbols disjoint from N.

(3) P is a finite set of formation rules or productions. It is a finite subset of $N\times(N\cup\Sigma)^*$. An element (A,β) in P will be written $A\to\beta$, and called a rule. A is called the left hand side (LHS), and β the right hand side (RHS).

(4) S is the distinguished symbol in N called the sentence or start symbol.

The vocabulary, V, of a grammar G is defined as NUΣ. A rule whose RHS is the empty word is called an e-rule.

The sentential forms of a grammar G are defined recursively as follows:

(1) S is a sentential form.

(2) If αAβ is a sentential form, and A→γ is in P, then αγβ is also a sentential form.

A sentential form of G containing no nonterminal symbols is said to be a sentence. The language generated by a grammar G, denoted L(G), is the set of sentences generated by G.

For the grammar G we define a relation $\xrightarrow[G]{}$ on V^* as follows:

If αAβ is a string in V^*, and A→γ is a rule in P, then $\alpha A\beta \xrightarrow[G]{} \alpha\gamma\beta$.

Read $\xrightarrow[G]{}$ as "directly derives". When it is clear to which grammar a relation applies, the subscript G will be dropped.

We use the notation $\xrightarrow{n}$ to denote the n-fold product of the relation →. That is to say, $\alpha \xrightarrow{n} \beta$ if there is a sequence $\alpha_0, \alpha_1 \ldots \alpha_n$ of n+1 words (not necessarily distinct) such that

$\alpha = \alpha_0$

$\alpha_{i-1} \Rightarrow \alpha_i, \quad 1 \leq i \leq n$

$\alpha_n = \beta$

This sequence of words is called a derivation of length n of β from α in G.

We use $\overset{+}{\Rightarrow}$ (read "derives in a nontrivial way") to denote the transitive closure of $\Rightarrow$, and $\overset{*}{\Rightarrow}$ (read "derives") to denote the reflexive transitive closure of $\Rightarrow$.

A direct derivation $\alpha A\beta \Rightarrow \alpha\gamma\beta$ is rightmost iff β contains only nonterminals. We say $\alpha \overset{+}{\Rightarrow} \beta$ is a rightmost derivation iff every direct derivation in it is rightmost. A grammar G is ambiguous if some sentence of G has more than one rightmost derivation. A rightmost derivation is also called a canonical derivation.

Let $\delta = \alpha\beta\gamma$ be a sentential form. Then β is called a phrase of the sentential form δ for a nonterminal A iff

$S \overset{*}{\Rightarrow} \alpha A \gamma$ and $A \overset{+}{\Rightarrow} \beta$

β is called a simple phrase iff

$S \overset{*}{\Rightarrow} \alpha A \gamma$ and $A \Rightarrow \beta$.

A handle of any sentential form is a leftmost simple phrase.

In certain cases a CFG may contain useless symbols and productions. These symbols and productions can be removed from G without affecting L(G). We say that a nonterminal A in N is useless in G (remember when we talk about G we assume it is context free) if there does not exist a derivation of the form $S \overset{*}{\Rightarrow} wAy \overset{*}{\Rightarrow} way$. Note that w, a, and y are in Σ^*. We say that a symbol X in V is inaccessible in G if X does not appear in any sentential form. A grammar

is reduced if N contains no useless nonterminals, and V contains no inaccessible symbols.

2.3. Simple Precedence Grammars and Languages

We must first define the Wirth-Weber precedence relations and describe the grammatical properties of being uniquely invertible and cycle free.

(a) The Wirth-Weber precedence relations $\lessdot$, $\doteq$ and $\gtrdot$ for a grammar G are defined on V as follows:

(1) We say that $X \lessdot Y$ if there exists $A \rightarrow \alpha X B \beta$ in P such that $B \overset{+}{\Rightarrow} Y\gamma$.

(2) We say that $X \doteq Y$ if there exists $A \rightarrow \alpha XY\beta$ in P.

(3) $\gtrdot$ is defined on $V \times \Sigma$ since the symbol to the right of a handle in a right sentential form is always a terminal. We say that $X \gtrdot a$ if $\alpha B Y \beta$ is in P, $B \overset{+}{\Rightarrow} \gamma X$ and $Y \overset{*}{\Rightarrow} a\delta$.

In precedence parsing it is convenient to put the word between endmarkers (we use \$). We assume that $\$ \lessdot X$ and $X \gtrdot \$$ for all X in V.

Precedence relations are conveniently expressed in a precedence matrix.

If more than one relation exists between two symbols, then we say there is a precedence conflict between the two symbols.

(b) A grammar is uniquely invertible if no two distinct rules have the same right hand side.

(c) A grammar is said to be cycle free if there is no derivation of the form $A \Rightarrow^{+} A$ for any A in N.

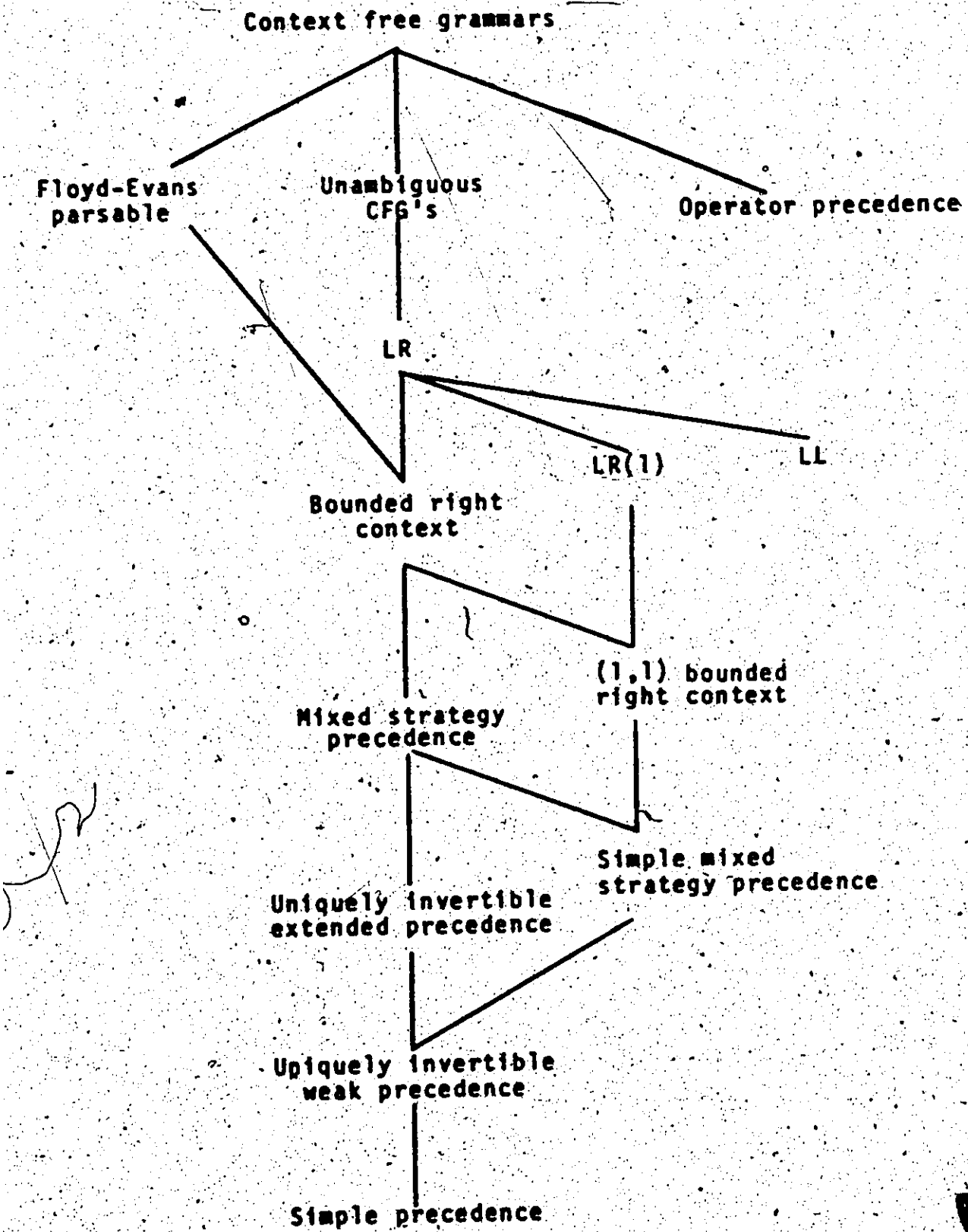
A CFG which is reduced, cycle free, contains no e-rules, and in which at most one Wirth-Weber precedence relation exists between any pair of symbols in V is called a precedence grammar. A precedence grammar which is uniquely invertible is called a simple precedence grammar. The language generated by a (simple) precedence grammar is called a (simple) precedence language. It can be shown that precedence grammars are unambiguous.

For some simple precedence grammars the precedence relations can be expressed in terms of two functions, with a saving in storage space required. If this is done we no longer know which pairs of symbols have no relationship. In the error detection and recovery employed we cannot afford to lose this information, therefore we do not use these functions.

In Figure 2.1. the relationship of (simple) precedence grammars to other restricted CFG's is displayed.

FIGURE 2.1.

Hierarchy of Grammars (from Aho and Ullman [1972])



2.4. Parsing

2.4.1. Introduction

Parsing has been defined by many people. We give a selection:

(a) Younger (1967):

"A parse of a sentence in a language L(G) is a description of how the sentence is generated by G, this description is usually in the form of a generation tree."

(b) Aho and Ullman (1972):

i) "The term parsing (or syntactic analysis) is given to the process of finding the syntactic structure associated with an input sentence."

ii) "We say that a sentence w in L(G) for some CFG G has been parsed when we know one (or perhaps all) of its derivation trees."

They define derivation tree as follows:

"A labelled ordered tree D is a derivation tree for a CFG $G(S) = (N,\Sigma,P,S)$ if

1) The root of D is labelled S.
2) If $D_1 \ldots D_k$ are the subtrees of the direct descendants of the root, and the root of D_i is labelled X_i, then $S \rightarrow X_1 \ldots X_k$ is a rule in P. D_i must be a derivation tree for $G(X_i) = (N,\Sigma,P,X_i)$ if X_i is a nonterminal and D_i is a single node labelled X_i if X_i

is a terminal.
3) Alternatively if D_1 is the only subtree of the root of D, and the root of D_1 is labelled e, then S→e is a rule in P."

Note: This definition of a derivation tree is not specific to any sentence in L(G).

(c) Graham (1971):

"The process of constructing a derivation of a sentence starting from the sentence and working back to the distinguished symbol is called parsing, and the derivation so obtained is called the parse of the sentence."

(This is in fact "bottom up" parsing.)

There are many techniques available for parsing a sentence (see Figure 2.2). We are interested in bottom up no backtrack parsing. With no backtrack parsing the sentence to be parsed is scanned just once. In bottom up parsing we start with the sentence and try to reduce it to the distinguished symbol. At each step of the parse a handle of the current sentential form is reduced. We use the rules of the grammar backwards. Two problems remain:

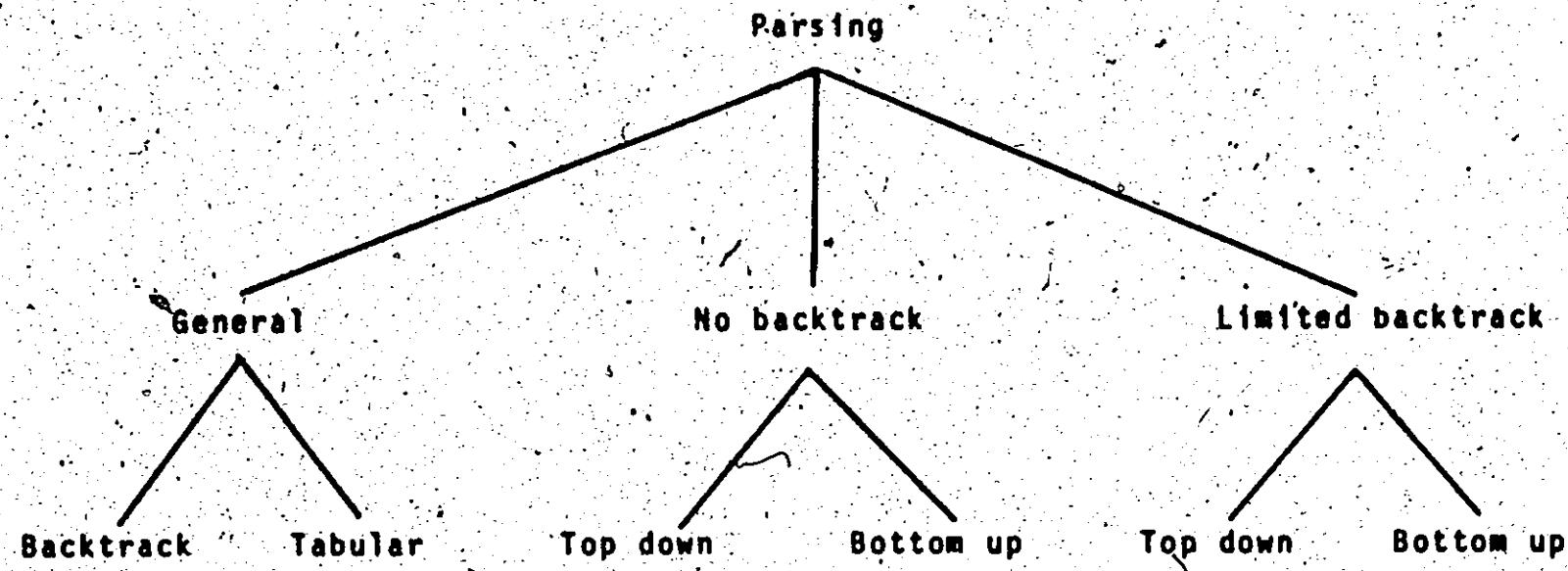
1) How do we identify the handle?

2) To what (i.e. which nonterminal) do we reduce it?

(c.f. top down where we start with the distinguished symbol and by using the rules of the grammar attempt to reach the sentence.)

FIGURE 2.2.

Parsing Techniques



2.4.2. Simple Precedence Parsing

Simple precedence grammars can be parsed by the "shift-reduce parsing algorithm". (Note that we abbreviate "sentences of the language generated by a simple precedence grammar" to just "simple precedence grammars".) Aho and Ullman (1972) give a rigorous treatment of shift-reduce parsing. We need not go to such depths. The shift-reduce parsing algorithm traces out a rightmost derivation. For simple precedence grammars, the shift-reduce algorithm is deterministic. It uses one pushdown list (called the "stack"), and employs a left to right input scan. We can view the action of a shift-reduce parsing algorithm in terms of configurations which are triples of the following form: $(\$X_1 \ldots X_m, a_1 \ldots a_n\$, p_1 \ldots p_r)$
where

1) $\$X_1 \ldots X_m$ represents a pushdown list with X_m on top. Each X_i is in V, and \$ acts as a bottom of the pushdown list marker.
2) $a_1 \ldots a_n$ is the remaining portion of the original input. a_1 is the current input symbol, and \$ acts as a right endmarker for the input.
3) $p_1 \ldots p_r$ is a string of semantic items. A rule may have a semantic item associated with it. When the rule is used in a reduction, the semantic item associated with it is output.

A configuration is changed by either shifting a symbol from the input to the stack, reducing a handle at the top of the stack, locating an error, or accepting the sentence.

The shift-reduce algorithm acts in two stages. The precedence relation between the top of the stack and the input symbol is obtained. If the relation is $\lessdot$ or $\doteq$ then a SHIFT is performed. If the relation is $\gtrdot$ then a REDUCE is performed since the right hand end of a possible handle has been located. Travel down the stack looking for a $\lessdot$ between two adjacent symbols. This marks the left hand end of the possible handle. The rules of the grammar are then used to reduce the handle to a nonterminal. (Since the grammar is uniquely invertible there is at most one rule that can be used to reduce the handle.) If there is no rule corresponding to the possible handle then an ERROR has been found. If there is no relation between the symbol at the top of the stack and the input symbol there is an ERROR. A sentence is ACCEPTED when a stack configuration of the form ($S, $, $p_1 \ldots p_r$) is reached. Immediately after carrying out a reduction the resulting configuration is checked to see if it is an acceptance configuration.

For a more complete description of error detection see Chapter 3.

The action of the shift-reduce parsing algorithm can be described by two relations $\vdash^{s}$ (for a shift) and $\vdash^{r}$ (for a reduce) on configurations.

Consider the following simple precedence grammar G.

$G = (\{S,A,B\}, \{a,b,c\}, P, S)$

where $P = \{S \to A|B,\ A \to c|cAb,\ B \to ab|aBb\}$

Parse the sentence aabb (neglecting any generated items).

$(\$, aabb\$) \vdash^{s} (\$a, abb\$)$

$\vdash^{s} (\$aa, bb\$)$

$\vdash^{s} (\$aab, b\$)$

$\vdash^{r} (\$aB, b\$)$

$\vdash^{s} (\$aBb, \$)$

$\vdash^{r} (\$B, \$)$

$\vdash^{r} (\$S, \$)$

$\vdash$ ACCEPT

Hence the sentence aabb is a sentence of L(G).

The three string operators FIRST, LAST and TOP are useful when we discuss error recovery in the simple precedence parser.

Consider a string α.

$\alpha = X_1 X_2 \ldots X_n$, α is in $(VU\$)^*$

X_i is in $VU\$$ for $i = 1 \ldots n$.

FIRST $(\alpha) = X_1$

LAST $(\alpha) = X_n$

Let i be the largest positive integer less than n such that $X_i \neq X_{i+1}$. If no such integer exists, then let i be 0.

TOP $(\alpha) = X_{i+1} \ldots . X_n$.

CHAPTER 3

ERROR DETECTION AND RECOVERY IN SIMPLE PRECEDENCE PARSERS -- A SURVEY

This chapter is based on the work by Leinius (1970) and Rhodes (1973).

3.1. Error Detection

3.1.1. Introduction

If a sentence is badly formed error(s) will be detected by the parser. Leinius proposes the following design objectives for error detection in the parser:

1) The analysis of syntactically correct programs (valid sentences in L(G)) should not be adversely affected by the incorporation of error detection procedures into the parser.
2) The parser should provide the "earliest" possible detection of errors.

Let A and B be parser configurations and A $\vdash$ B. If an error is detected in configuration A the error detection is earlier than if no error had been detected in configuration A, but one had been found in configuration B.

Some errors are detected naturally by the parser.

Specific error checks will be introduced into the parser to enable errors to be detected earlier than they would normally be in natural error detection.

3.1.2. Simplest Detection

The following rules are used for the parser:

Let the contents of the stack be γ.

Let the input symbol be a.

1) Shift symbols until LAST $(\gamma) \diamond a$.
2) Look down stack for a $\diamond$ relation between two symbols, and then attempt a reduction.

Any errors are located when reductions are attempted.

(Error detection in this parser can be simply improved by checking for no relationship between LAST (γ) and a. These errors may be considered to be detected naturally.)

3.1.3. Leinius Error Detection

Leinius describes four error conditions.

Let the contents of the stack be γ.

Let the input symbol be a.

1) Character -- pair (type 0)

There is no precedence relation between LAST (γ) and a.

2) Type 1 phrase error

TOP (γ) does not occur on the right hand side of any rule of the grammar.

3) Type 2 phrase error

TOP (γ) is reduced to A. No precedence relation or $\doteq$ exists between the symbol beneath TOP (γ) and A[†]. A is said to be Leinius unstackable on the left.

4) Type 3 phrase error

TOP (γ) is reduced to A. No precedence relation exists between A and a. A is said to be unstackable on the right.

3.2. Error Recovery

3.2.1. Introduction

The parser designer is faced with a number of choices. No longer is naïve error recovery (e.g. /read and discard until next ";") considered adequate. More sophisticated methods must be examined. The first decision

---

[†]If $\doteq$ were allowed a rightmost derivation could not be obtained.

is whether to write specific correction/recovery action for the particular language. For example specific error recovery could be given for type 0 and type 1 errors. For the type 0 errors special instructions could be inserted into the precedence matrix. For the type 1 errors, "error productions" could be incorporated into the grammar. Automatic methods of correction/recovery will then have to be investigated. In this project, phrase based recovery systems are dealt with. These systems pose two main problems:

1) How to specify the bounds of the incorrect phrase (the incorrect phrase is called a "candidate").
2) What to do with the candidate. Can it be deleted, or would it be better to replace it with a nonterminal, and if so, which one? Local context is used to help make these decisions.

In the systems described by Leinius and Rhodes, the stack is manipulated, but the remaining input is not.

In the rest of this chapter, the recovery methods used by Leinius and Rhodes are outlined, and the method of Rhodes is treated in detail.

3.2.2. Outline of Leinius' Recovery System

(1) Select a candidate.

(2) Compute a "replacement set" for the candidate. A member of the replacement set is either a non-terminal or a deletion. The set is computed using -- "left context, the right context character (in the input register), and the set of productions", Leinius (1970, p. 72).

(3) If the number of elements in the replacement set is zero or greater than one, repeat from step 1, otherwise remove the candidate from the stack and insert the replacement (for deletion, ϵ, the empty word, is inserted).

The method used to select candidates is now described. Leinius considers sets of recovery points. A candidate is the word between a left recovery point and a right recovery point. The left recovery points are in the stack. Consider two adjacent symbols X_i and X_{i+1} in the stack. If $X_i \lessdot X_{i+1}$, then there is a left recovery point. When an error is detected, all left recovery points are defined. The first right recovery point is between the top of the stack, and the input symbol when the error occurred. When an error is detected, only one right recovery point is known. To obtain a new right recovery point the parse proper

is restarted. The next right recovery point is located when either a new error is detected, or the relation between the symbol at the top of the stack and the input symbol is ◦.

When a new candidate is required, the old candidate is extended by either taking a new left recovery point, or obtaining a new right recovery point. "In which direction should the extensions be made? As we will show later, information obtained from the unsuccessful attempt to reduce the old potential phrase may be used to indicate the direction in which the search for a new recovery point should be made", Leinius (1970, p. 51).

3.2.3. Outline of Rhodes Recovery System

(1) Identify a maximum of three candidates (there is either one or three).

(2) For each candidate compute a "replacement set". A member of a replacement set is either a nonterminal or a deletion. The set is computed using left context (one symbol) and right context (one symbol). Let candidate i(i=1,2,3) be γ. A nonterminal A will be in candidate i's replacement set if "A is locally syntactically correct (LSC) with respect to γ." Deletion will be in candidate i's replacement set if "Deletion is LSC with respect to γ." These

two terms are defined in section 3.2.4. The member A is called a locally syntactically correct nonterminal, and the member deletion is called a locally syntactically correct deletion. The R.H.S. of the rules of each LSC nonterminal are called LSC phrases.

(3) If the replacement set for each candidate is empty, then exit from the Rhodes recovery to a specific recovery action, otherwise using a minimum distance approach compute the distance from each candidate to all its LSC phrases and to ε (this distance is arbitrarily called "cost").

(4) For each LSC nonterminal record the smallest cost, and record the cost of deletion for each candidate. Each member of each candidate's replacement set now has a "least" cost associated with it. This is called the selection cost.

(5) Choose a member with the best selection cost. The best selection cost may occur more than once in a candidate's set, and may occur for different candidates. We arbitrarily select one of these members. The member is said to be chosen. Let it be in candidate i's set, where candidate i is γ.

(6) If the best selection cost is higher than some arbitrary value, then we exit from the Rhodes recovery to a specific recovery action, otherwise if the chosen member is deletion then the candidate i is deleted (i.e. γ is removed from the stack), otherwise candidate i is replaced by the chosen member† (i.e. γ is replaced by the chosen member -- a nonterminal).

In section 3.2.4. we look at how left context and right context are used to determine whether a nonterminal or a deletion is LSC, and in section 3.2.5. we explain how the candidates are identified.

---

†Rhodes talks about replacing a candidate by the phrase that approximates it best, Rhodes (1973, pp. 51, 56, 57, 62, 89). In our notation, instead of replacing candidate i by the chosen member (a nonterminal) it would be replaced by the chosen member's LSC phrase which yielded the least cost. We do not favour this. The LSC phrase itself has not been checked for stackability. In the Rhodes notation, Rhodes (1973, p. 56), a phrase α, where $A \rightarrow \alpha$ is in P, is defined to be LSC in the context $\gamma\alpha\beta$ in terms of A being stackable against LAST (γ) and FIRST (β). The phrase α may not itself be stackable against LAST (γ) and FIRST (β). If the phrase is to be inserted into the stack, it should be checked for stackability. Our method of replacement follows Leinius (1970, pp. 48 and 49).

3.2.4. Locally Syntactically Correct (LSC) Nonterminals and Deletions

Firstly we consider nonterminals.

The concept of locally syntactically correctness for a nonterminal is also required for the condensation stage of recovery (see section 3.2.5.).

Consider the word $\alpha\gamma\beta$. A nonterminal A is LSC with respect to γ if A is "right stackable" and "Rhodes left stackable". The criteria for right stackability are the same as for error detection (3.1.3.). We say A is right stackable if a precedence relation exists between A and FIRST (β). We say A is Rhodes left stackable if:

i) LAST (α) $\lessdot$ A

or ii) LAST (α) $\gtrdot$ A[†] and TOP (α) is a right hand side (RHS) of a rule

or iii) LAST (α) $\doteq$ A and TOP (αA) is a prefix of the RHS of a rule, or is the RHS.

Note that in our definitions $A \to \gamma$ need not be a member of P. When we are "condensing" $A \to \gamma$ will be a member of P, and when we are computing the replacement sets it will

---

[†]Allowing this means that a rightmost derivation is not obtained. The derivation in any case is not a derivation of the original badly formed sentence, but of a well formed sentence that is "close" to it.

not be a member of P. In Rhodes thesis, Rhodes (1973), $A \rightarrow \gamma$ is always in P. Our definition differs from Rhodes' conceptually. He defines LSC phrases and does not formally define an LSC nonterminal. We define an LSC nonterminal, and then define an LSC phrase in terms of this.

Secondly we consider deletion. Given the word $\alpha\gamma\beta$, <u>deletion</u> is <u>LSC with respect to γ if</u>

i) LAST (α) $\lessdot$ FIRST (β)

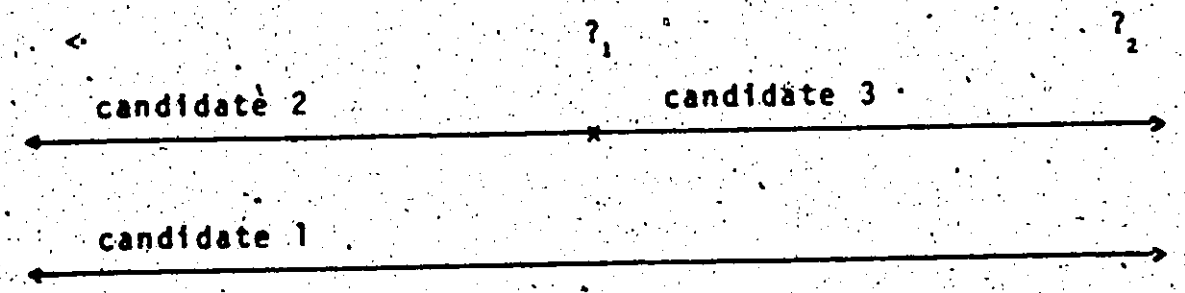
or ii) LAST (α) $\doteq$ FIRST (β) and TOP (α) is a RHS of a rule

or iii) LAST (α) $\doteq$ FIRST (β) and TOP (α FIRST (β)) is a prefix of a RHS of a rule, or is a RHS.

These criteria correspond to Rhodes left stackability.

3.2.5. <u>Selection of Candidates</u>

Rhodes defines three recovery points:



The leftmost recovery point is the first $\lessdot$ in the stack below the first error point. This is equivalent to the Leinius "first left recovery point" (3.2.2.). We want

to take as little from the stack as possible, so it seems a reasonable choice.

In certain situations $?_1$ is absorbed by $?_2$. This results in only one candidate.

Let us suppose that an error is detected for the first time. $?_1$ and $?_2$ are identified as follows:

1) Carry out first "backward move"

It is assumed that $\lessdot$ exists between the top of the stack and the input symbol. The backward move involves travelling down the stack searching for a $\doteq$ relationship ($\doteq$ can be found in the stack because of the criteria used to define Rhodes left stackability -- see section 3.2.4.). When a $\doteq$ is located, reductions (here called condensations) are attempted. The LHS of a potential condensation must satisfy some criteria for stackability. We favour using the same criteria used in computing the replacement sets, whereas Rhodes did not use a right stackability test, Rhodes (1973, p. 43). The traverse down the stack is repeated until no more condensations occur. The first backward move is then finished. $?_1$ is defined as the position between the top of the stack and the input symbol after the first backward move.

2) Return action to the regular parser

It is assumed that $\lessdot$ exists between the top of the stack and the input symbol. Before returning to the regular parser, the current input symbol is stacked, and the

next symbol read. The forward move† is then said to have started. The forward move finishes when either the sentence is accepted or an error is encountered. $?_1$ may have been absorbed in the forward move. This happens if a reduction occurred at or below the symbol beneath $?_1$.

3) Carry out second backward move

It is no longer assumed that a $\lessdot$ exists at $?_1$, but it is again assumed that $>$ exists between the symbol at the top of the stack, and the input symbol. The stack is traversed as in the first backward move. After all the possible condensations have been carried out, the second backward move is terminated. $?_2$ is then defined as the position between the symbol at the top of the stack, and the input symbol.

$?_1$ may be absorbed in the second backward move. This happens if a condensation occurs at or below the symbol beneath $?_1$.

After a backward move the stack configuration is checked for end-of-parse.

---

†If there is no more input, $?_2$ absorbs $?_1$, and there is no forward move and no second backward move.

CHAPTER 4

IMPLEMENTATION

A basic simple precedence parser, Leinius (1970), augmented with error detection, Leinius (1970), and error recovery, Rhodes (1973), is implemented. See Chapter 3 for a discussion of the system.

The Algol W subset (or ALGSUB) described by Rhodes (1973) is used for the implementation. The grammar given by Rhodes had to be modified to make it uniquely invertible and free of simple precedence conflicts. Barnes (1973) was found useful in removing the conflicts. Implementation was done on a CDC 6400.

4.1. Constructor

For ALGSUB the constructor requires 51K octal words of core, compiled in 21 seconds, and ran in 6 seconds (full output). See Appendix A for a program listing, output for ALGSUB, and the dayfile. The output for ALGSUB includes a listing of the grammar.

The program includes documentation on:

1) assumptions made about the grammar
2) summary of input/output files
3) output options
4) summary of error codes
5) procedure and function structure.

The meaning of constants and global variables is also described.

For a system flowchart see Appendix B.

4.1.1. Input

1) Format

The grammar must be specified in "inverse" Backus-Naur Form. A terminal symbol must start with a < and is delimited by a >. It must be written on one card. A nonterminal symbol must start with a letter or digit, and is delimited by a blank or EOL. A rule is not restricted to one card. Rules are terminated by a semi-colon (;).

e.g. A →B C D / B <CC>
/ <CC>B;

Either → or := or ::= may be used between the LHS and RHS of a rule.

Semantic items may be attached to rules. The item is enclosed between [and]. There is no restriction on the size of the semantic item. If a semantic item is to be continued on the next card a [is punched after the last character on the current card. (This is necessitated by the CDC SCOPE input procedures. It avoids unwanted insertion or deletion of blanks.) The semantic item is inserted either before the ; if the rule occurs last, or before the / if there are subsequent rules.

```
e.g.  A →B C D [A SEMANTIC ITEM]/B <CC>
          /<CC>B  [A VERY [
      LONG PIECE [
       OF SEMANTICS INDEED] ;
   column 1
```

The semantic item attached to A →<CC>B is "A VERY LONG PIECE OF SEMANTICS INDEED".

A nonterminal may occur any number of times on the LHS of rules. No check is made for duplication

e.g. A →B ;A →B ; is equivalent to A →B /B ;

The grammar is terminated by a period (.).

2) Errors

If an input error is detected, the input is discarded up to the next ; and processing continued.

Precedence grammars may not contain e-rules. If < > is found an error is reported unless the next character is > in which case it is assumed that ">" is a terminal symbol.

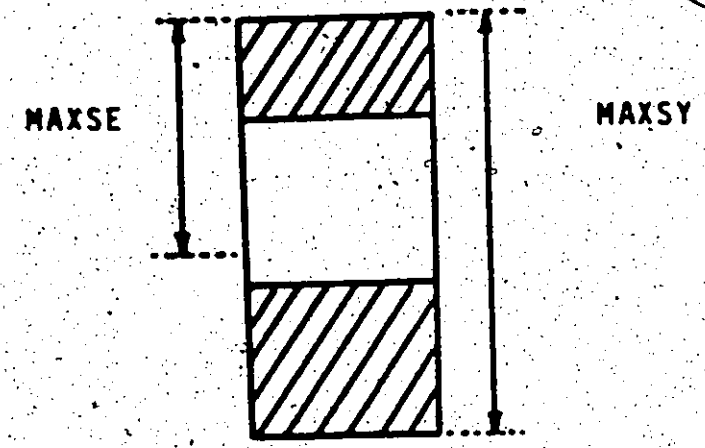
A lexicographic input error summary is given in Appendix C.

4.1.2. Data Structure

1) Symbols are stored in one, two, or three words (10, 20 or 30 characters).

2) There are two separate hash tables, one for terminal symbols, and one for nonterminal symbols.

3) There is one symbol table.



Nonterminal symbols are stored starting from the top, and terminal symbols are stored starting from the bottom. When all the symbols have been obtained, the terminal symbols are moved to close the gap.

4) The dimensions of the precedence matrix are 1..MAXSY1,1..MAXPREC. MAXSY1 is used to allow for $, the endmarker. A word is used to hold 10 precedence relations.

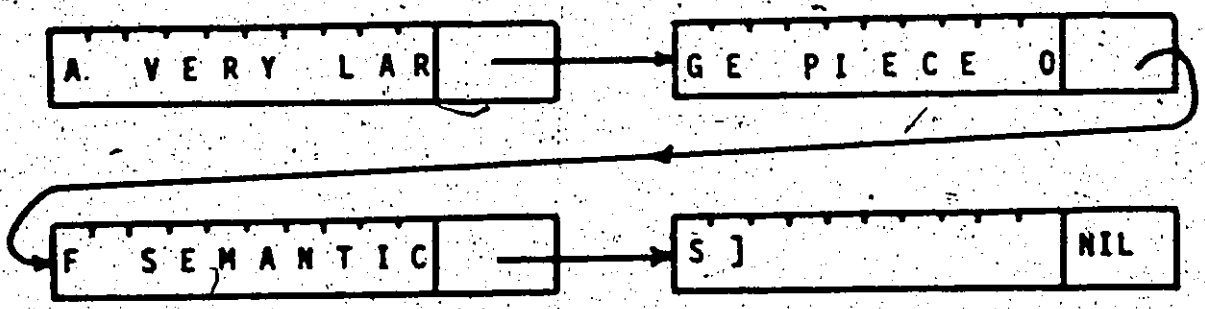
5) The left and right sets.

Each set is stored in a boolean array. At present each element in the array takes one word. However in the future it will be possible to "pack" them. (anticipated in the next version of PASCAL).

6) The syntax graph and reverse syntax graph. Semantic items are common to both graphs. The semantic node structure is:

Each node holds ten characters of semantic information. If more than ten characters is required, nodes are chained using PCONT. The semantic item is delimited in the last node by].

e.g. "A VERY LARGE PIECE OF SEMANTICS "



Symbols are stored in both graphs as numbers. When the meaning is clear, "symbol number" is abbreviated to "symbol".

a) The syntax graph.

For a description of a syntax graph see Barnes (1973).

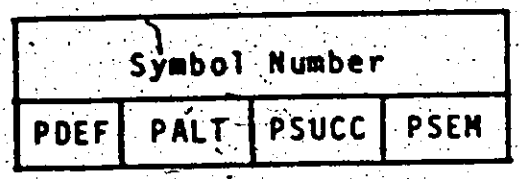
The graph consists of:

i) A key

ii) Symbol nodes

iii) Semantic nodes.

The key is an array of pointers, PSEDEF. PSEDEF[n] points to the rules for nonterminal symbol number n. The symbol node structure is:

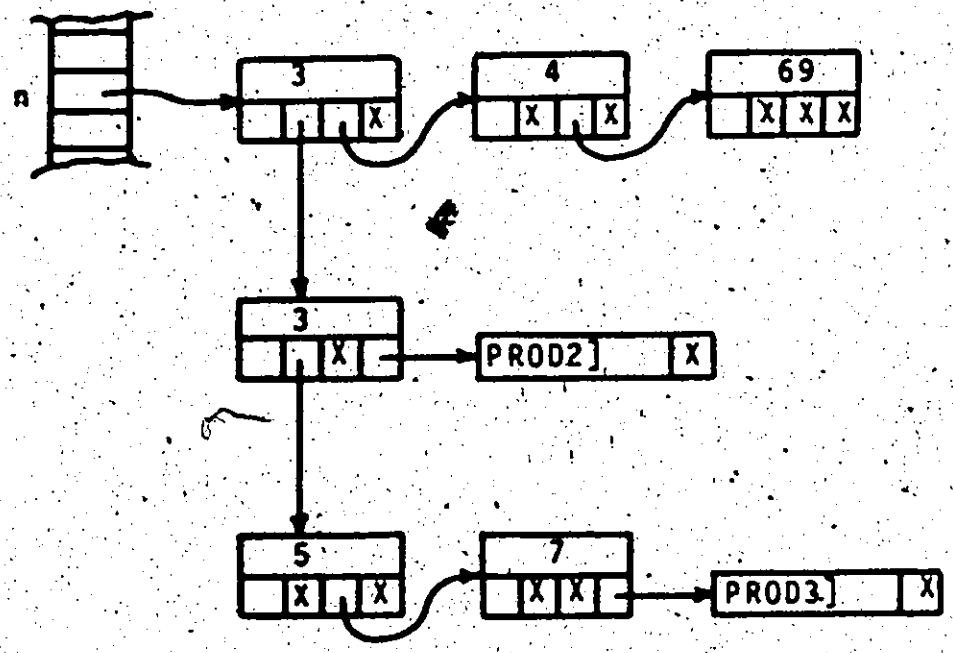


A symbol node contains the symbol number and four pointers.

Pointer	Use
PDEF	Shows where the symbol is defined (NIL for nonterminal).
PALT	Gives an alternative rule.
PSUCC	Gives next symbol for a rule
PSEM	Gives start of semantic item (NIL if rule does not have a semantic item).

example (excluding PDEF)

X ≡ NIL



Writing the rules in terms of symbol numbers, the above graph represents

n ←3 4 69 /3 [PROD2]/5 7 [PROD3];

b) The reverse syntax graph.

The reverse syntax graph consists of all the RHS of the rules. It is organised to facilitate a search for a given word.

The graph consists of:

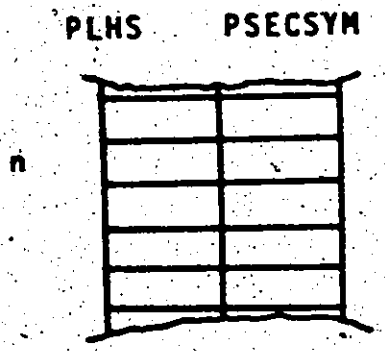
i) A root

ii) Reverse symbol nodes

iii) Left hand side symbol nodes

iv) Semantic nodes.

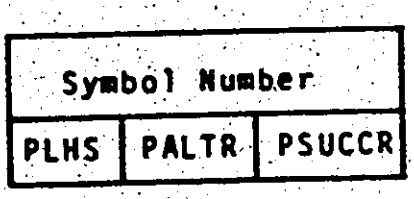
The root is an array of records; a record is two pointers, one points to the LHS of a rule, and the other points to the second symbol of the RHS of a rule.



Consider symbol number n. If symbol number n constitutes a RHS of a rule, then PPROD[n].PLHS points to the LHS of the rule. If symbol number n is a prefix of a RHS of a rule, then PPROD[n].PSECSYM points to the reverse symbol node holding the second symbol of the RHS of the rule.

Rules of the type AA →BB /BB CC ; may occur.

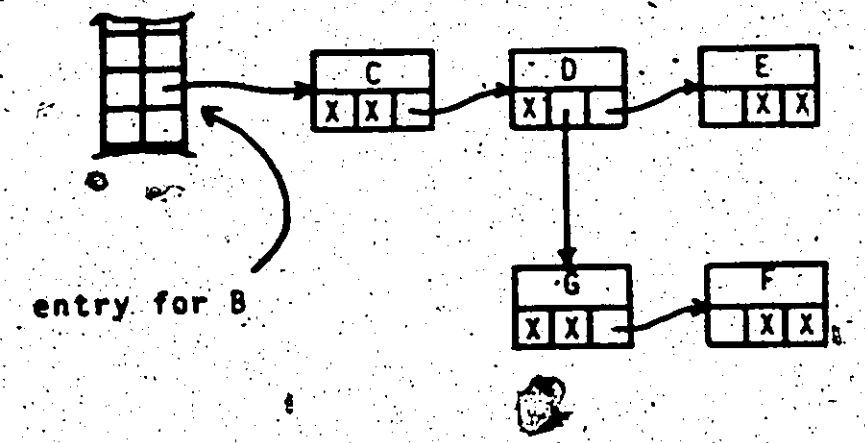
The reverse symbol node structure is:



Pointer	Use
PLHS	Points to a LHS symbol node.
PALTR	Points to an alternative reverse symbol node.

e.g. A →B C D E ;

P →B C G F ;



PSUCCR Gives next symbol in the rule.

The LHS symbol node structure is:

Symbol Number	PSEMR

PSEMR points to the semantic item (if there is one) attached to the rule.

Example

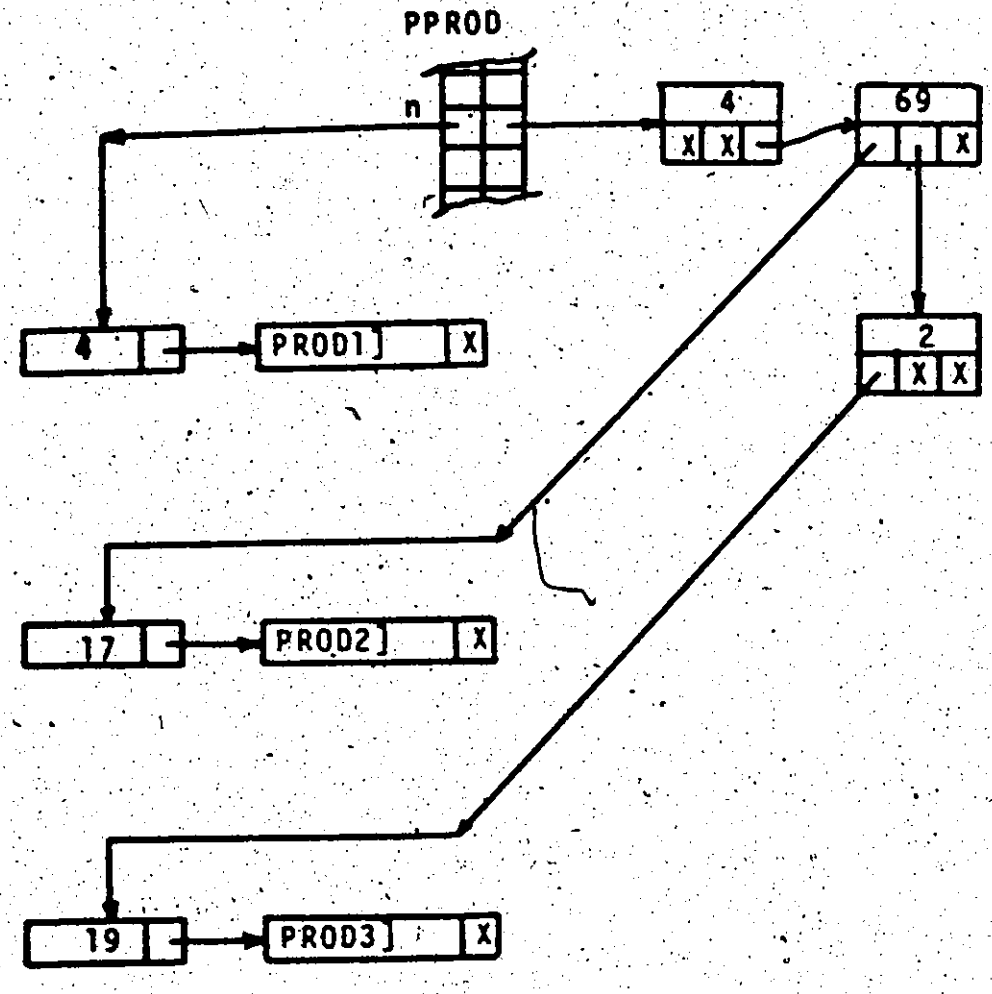
Consider the following rules (in symbol numbers, n is a symbol number).

4 →n [PROD1];

17 →n 4 69 [PROD2];

19 →n 4 2 [PROD3];

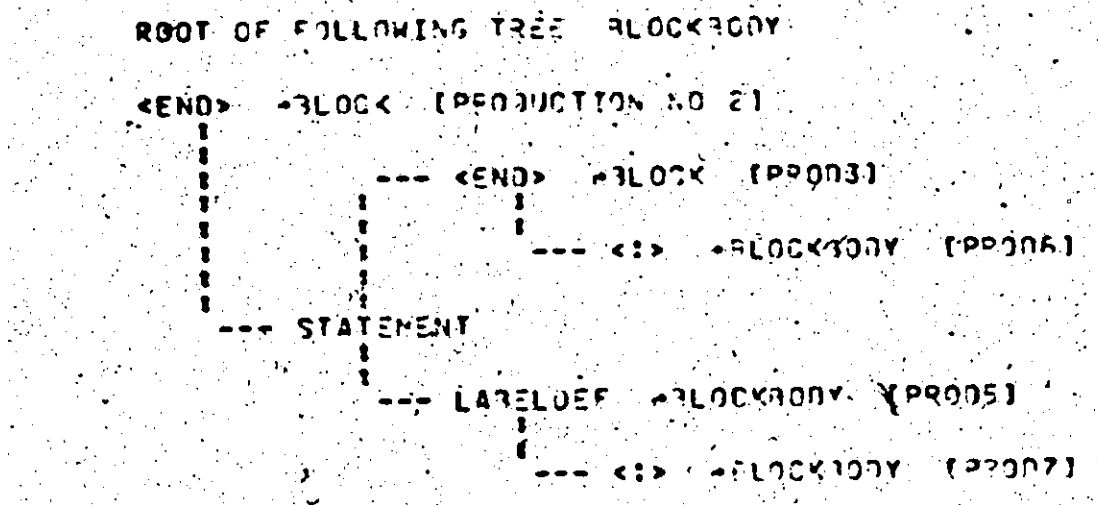
These will be represented in the reverse syntax graph as follows:



4.1.3. Printing of Reverse Syntax Graph

See Appendix A for a print of the reverse syntax graph for ALGSUB. Walker, Redish and Wood (1974) have developed a binary tree display algorithm which is

incorporated in the procedure used to print the reverse syntax graph.



Consider the extract (above) from the print of the reverse syntax graph for ALGSUB. This tree represents the RHS of rules of more than one symbol which start with BLOCKBODY. It is built from the following rules:

BLOCK →BLOCKBODY <END>[PRODUCTION NO 2];

BLOCK →BLOCKBODY STATEMENT <END>[PROD3];

BLOCKBODY →BLOCKBODY STATEMENT <;>[PROD6];

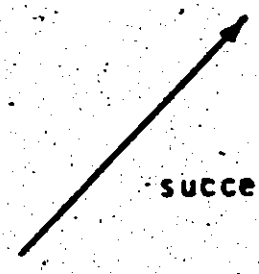
BLOCKBODY →BLOCKBODY LABELDEF [PROD5];

BLOCKBODY →BLOCKBODY <;>[PROD7];

Alternatives and successors are interpreted as follows:

alternatives

e.g. LABELDEF is an alternative to STATEMENT



successors

e.g. <END> (for PROD3) is a successor to STATEMENT

The → points to the LHS symbol of the rule. Semantics for the rule follow the LHS symbol.

4.2. Parser (including error detection and recovery)

For ALGSUB the parser requires 33K octal words of core, compiled in 16 seconds, and for our test data (see Chapter 5) ran in under 2 seconds. See Appendix D for a program listing, (ALGSUB), outputs (small correct sentence, and same sentence with one input error), and the dayfile.

The program listing includes documentation on:

1) summary of input/output files
2) summary of error codes (abort conditions)

3) procedure and function structure.

The meaning of constants and global variables is also described.

For a detailed flowchart of the basic simple precedence parsing algorithm see Appendix E, and for a system flowchart of the error recovery, see Appendix F.

4.2.1. Preparing the Parser for a Given Simple Precedence Language, L(G)

There are six stages:

1) Process the simple precedence grammar G using the constructor to obtain:

i) The syntax graph and reverse syntax graph on permanent file CONPAR.

ii) Punched output used to initialise the precedence matrix (PRM), symbol table (SYM), and terminal hash table (HSH)

2) Insert the punched output into the deck

Card code	to initialise
PRM	PRECMAT
SYM	SYMTAB
HSH	TTAB

3) ATTACH CONPAR, and peel off ROOTFLE, SEDEFLE and NODEFLE.

4) Insert a lexical scan (with appropriate data initialisations) for the language L(G). For ALGSUB, the lexical scan is called LEXALGSUB. The transition matrix method is used. The following data items are specific to the lexical scan for the language L(G):

i) TRANMAT -- transition matrix

ii) COLNO -- column number, used in conjunction with TRANMAT

iii) CHARNOT -- characters not appearing in terminal symbols (special attention is given to EOL)

iv) IDSTATES -- used to define reserved words.

5) Insert initializations for SYMCOST, the Rhodes recovery costs.

6) Check that the values of the following constants are the same in the constructor and the parser:

MAXSE

MAXSY

MAXSY1

MAXPREC

HSHT

MAXA

MAXC

Also check that the number used to define GRAPH is the same in the constructor and parser.

4.2.2. An Annotated Parser Run

The best way of working through the implementation is with an example. Before we begin, the following two notes may be useful. Firstly, we have a general note on output; two output files are produced, OUTPUT and SEMAN.

1) OUTPUT. The purpose of this file is to show the working of the basic simple precedence parser augmented with error detection and recovery. Four procedures are associated with this output file:

PUTSYM	prints a symbol
PUTSYMS	prints a word, with two spaces between each symbol
PUTREDN	prints the RHS of a rule
STCKSTAT	prints the current stack (in terms of symbol numbers).

2) SEMAN. This serves two purposes. It gives a brief trace of the parse, and it could be used, perhaps in a modified form, for generation of semantics. Every time the symbol configuration of the stack is changed, an item is output to SEMAN. Four items are possible:

SEMANTIC	normal reduction
C SEMANTIC	condensation
R SEMANTIC	replacement
D	deletion

SEMANTIC is the semantic item attached to a rule. It may be null.

Secondly, a note on the representation of $\lessdot$ in the stack. Leinius (1970) suggested that a special symbol be inserted into the stack to mark a $\lessdot$ relation between symbols. We adopted this idea, and used -1 as the special symbol.

e.g. a print of a stack produced by STCKSTAT

```
TOP OF STACK AT   11
 71   -1   70   -1    3   -1   20   52   -1   25   49
```

The stack contains symbol numbers or -1s, so $71 \lessdot 70$, $70 \lessdot 3$, $3 \lessdot 20$; $52 \lessdot 25$.

Consider the following example:

```
      %PHODES 14
      START
            BEGIN
              INTEGER AA,BB,CC ;
              %PHODES P44%
      (L)-----INTEGER ARRAY A(1..%,3..4) ;
              INTEGER ARRAY A1,A2,A3(...,1..5) ;
              %
              AA := ; ;
              A1(2,3) := A1(7,8) + AA*BB*CC ;
              %PHODES P52%
      (M)-----IF I >= 4
                THEN
                  GO TO L ;
              BB := C ;
              A1(2,3) := A1(7,8) + AA*BB*CC ;
              %PHODES P47%
              IF I = 1
                THEN
      (N),(P)---------C = 0
              GO TO LAB1 ;
            END
```

This sentence contains four errors described by Rhodes:

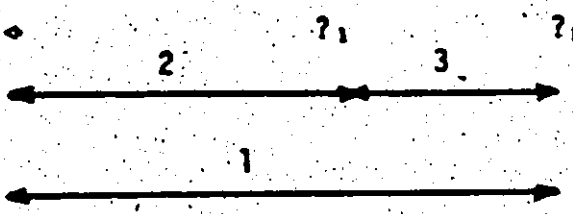
(i) missing comma (,) between 2 and 3.

(ii) >= instead of ≥

(iii) - instead of :=

(iv) missing semi-colon (;)

Points of interest in each of the four recoveries will be examined. In most recoveries, there is a first backward move, a forward move, and then a second backward move. (If an error occurs when the current input symbol is $ there is only one backward move, and no forward move, and if an error occurs when the next input symbol is $, then there is no forward move, just a backward move, stack current symbol, and another backward move.) Condensations are shown, these do not alway occur -- our criteria for allowing a condensation are stricter than those of Rhodes (see Chapter 3). Replacement sets are shown, followed by the costs of the LSC phrases and LSC deletions. One member of a set is chosen. We will see a candidate deleted, and a candidate replaced. Remember that the candidates are numbered:



The output to SEMAN is given in Figure 4.1.

Extracts of file OUTPUT are given.

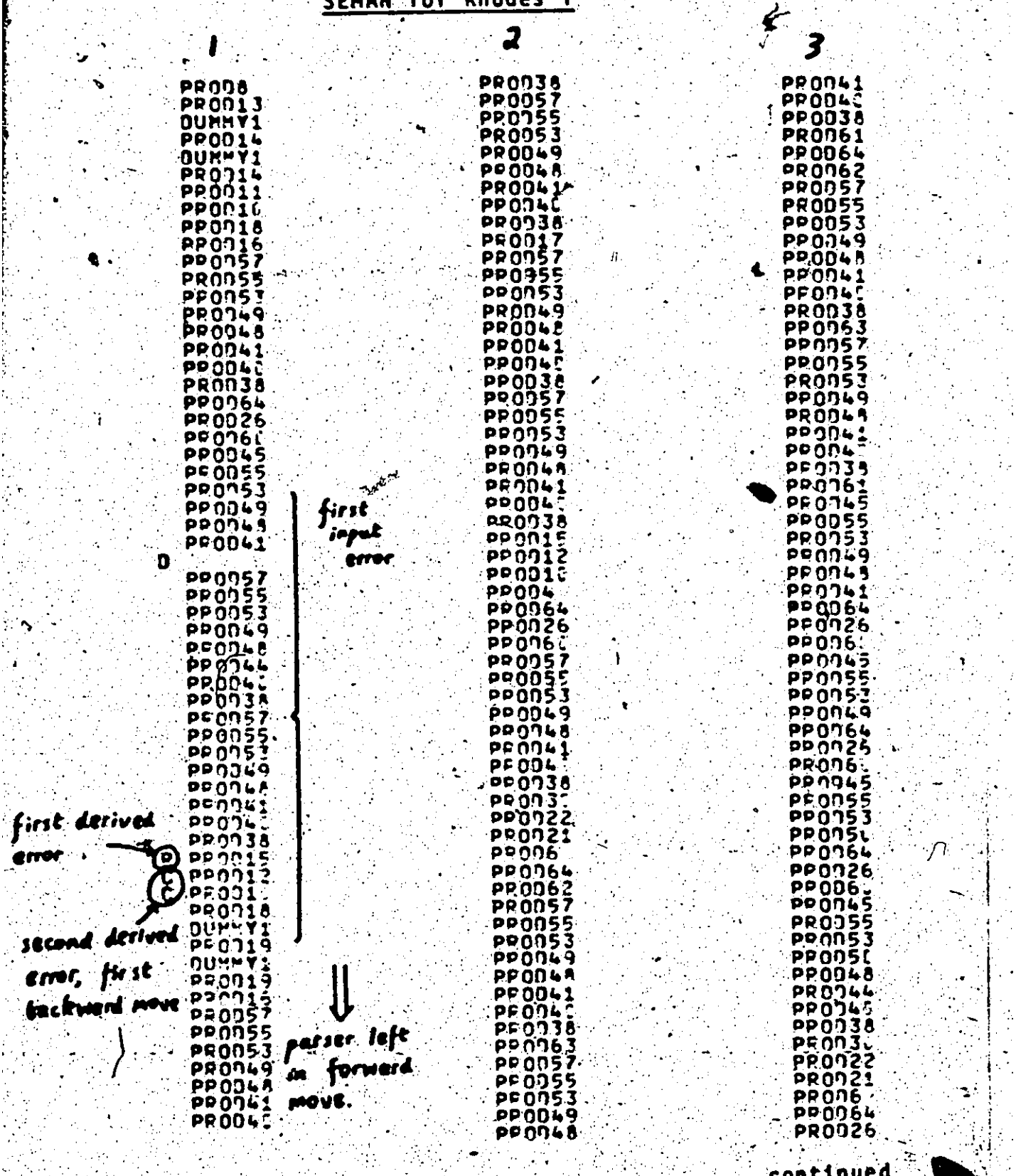
Output from the basic parsing algorithm is shown below.

```
SYMBOL  71  <START>

TOP OF STACK AT      1
  71

SYMBOL  66  <BEGIN>

TOP OF STACK AT      3
  71   -1   71

SYMBOL  65  <INTEGER>

TOP OF STACK AT      5
  71   -1   71   -1   66

LHS     5  BLOCKHEAD
RHS        <BEGIN>
SYMBOL  64  <IDENTIFIER>

TOP OF STACK AT      7
  71   -1   71   -1    5   -1   65
```

Input symbol with stack configuration

FIGURE 4.1.

SEMAN for Rhodes 1



continued

FIGURE 4.1. continued

SEMAN for Rhodes 1

```
SYMBOL   63   <,>

TOP OF STACK AT      8
  71    -1    7C    -1     5    -1    65    64

LHS      8   SIMPLEDEC
RHS          <INTEGER>  <IDENTIFIER>
SYMBOL   64   <IDENTIFIER>

TOP OF STACK AT      9
  71    -1    7?    -1     5    -1     ?    -1    6?

LHS     1?   AAAA?
RHS          <,>
SYMBOL   6?   <,>

TOP OF STACK AT      9
  71    -1    7?    -1     5    -1     8    1?    64

LHS      8   SIMPLEDEC
RHS          SIMPLEDEC  AAAA?  <IDENTIFIER>
SYMBOL   64   <IDENTIFIER>
```

Note how the semantics in SEMAN tie in with the reductions.

e.g. BLOCKHEAD →<BEGIN>[PROD8];

SIMPLEDEC →<INTEGER><IDENTIFIER>[PROD13];

AAAA0 →<,>[DUMMY1];

etc.

Parsing continues until the parser has the following configuration:

```
SYMBOL  48.  <+>

TOP OF STACK AT   11
 71   -1   7:   -1   5   -1   11   12   63   -1   64.

LHS     19   ARRAYID
RHS          <IDENTIFIER>
LHS     18   SIMPLEVAR
RHS          ARRAYID
LHS     20   VARIABLE
RHS          SIMPLEVAR
LHS     33   PRIMARY
RHS          VARIABLE
LHS     30   SECONDARY
RHS          PRIMARY
LHS     29   FACTOR
RHS          SECONDARY
LHS     28   TERM*
RHS          FACTOR
LHS     27   TERM
RHS          TERM*
LHS     25   EXPR**
RHS          TERM
SYMBOL  48.  <NUMBER>

TOP OF STACK AT   12
 71   -1   7:   -1   5   -1   11   12   63   -1   25   48

SYMBOL  48.  <NUMBER>

TOP OF STACK AT   14
 71   -1   7:   -1   5   -1   11   12   63   -1   25   48   -1   4:
```

The symbol at the top of the stack is <NUMBER>, and <NUMBER> is also the input symbol. A type 0 error (character pair) is detected, and the procedure ERROR is entered. The first backward move is carried out, see Figure 4.2.

FIGURE 4.2.

Output for First Input Error, First Backward Move, Forward Move, and Second Backward Move

```
ERROR
ENTER RACKMOVE  0
ENTER CONDENSE

TOP OF STACK AT    14
  71   -1   70   -1    5   -1   11   12   63   -1   25   48   -1   40

LHS   31   PRIMARY

TOP OF STACK AT    14
  71   -1   70   -1    5   -1   11   12   63   -1   25   48   -1   40

LEAVE CONDENSE
LEAVE RACKMOVE
ESYMBOL  63  <..>

TOP OF STACK AT    16
  71   -1   70   -1    5   -1   11   12   63   -1   25   48   -1   40   -1   40

LHS   31   PRIMARY


ERROR
ENTER RACKMOVE  1
ENTER CONDENSE

TOP OF STACK AT    15
  71   -1   70   -1    5   -1   11   12   63   -1   25   48   -1   40   40

LHS   -1

TOP OF STACK AT    15
  71   -1   70   -1    5   -1   11   12   63   -1   25   48   -1   40   40

LEAVE CONDENSE
LEAVE RACKMOVE
```

A ⋖ is assumed to exist between the top of the stack and the input symbol, and condensations are attempted. Symbol number 40, <NUMBER>, is found to be the RHS of the rule PRIMARY → <NUMBER>. PRIMARY is Rhodes left stackable against symbol number 48, <+>, (the relation ⋖ holds), but is not right stackable against <NUMBER>. The stack is traversed, and if a ⋖ is found between two symbols, condensations are attempted. No ⋖ are found, and as the stack remained unaltered for the traverse, the first backward move is finished. The current input symbol is stacked, and a new symbol is obtained (ESYMBOL), and the forward move is entered (control is returned to the basic parser). An error is detected straight away. <NUMBER> ⋖ <•••>, and <NUMBER> can be reduced to PRIMARY, but PRIMARY is not Leinius left stackable against <NUMBER>. The second backward move is entered, no condensations occur. The candidates are now defined:

```
CANDIDATES AT
BOTTOM   14
QUEST1   14
STACK    15
```

The replacement sets are now computed. This may involve calls to the procedure REDUCE for Rhodes left stackability.

```
CALLS TO REDUCE FOR RHODES LEFT STACKABILITY

LHS   31  PRIMARY
LHS   31  PRIMARY
LHS   25  EXPR**
LHS   25  EXPR**
```

```
REPLACEMENT SETS
NT.      CANDIDATE
          1     2     3
 0        F     T     T
 1        F     F     F
 2        F     F     F
 3        F     F     F
 4        F     F     F
 5        F     F     F
 6        F     F     F
 7        F     F     F
 8        F     F     F
 9        F     F     F
10        F     F     F
11        F     F     F
12        F     F     F
13        F     F     F
14        F     F     F
15        F     F     F
16        F     F     F
17        T     F     F
18        T     F     F
19        T     F     F
20        F     F     F
21        F     F     F
22        F     F     F
23        F     F     F
24        F     F     F
25        F     F     F
26        F     F     F
27        T     F     F
28        T     F     F
29        T     F     F
30        T     F     F
31        T     F     F
32        F     T     F
33        F     T     F
NON TERMINAL NUMBER 0 IS DELETION
```

Each LSC phrase and the LSC deletions are costed.

COST OF LSC PHRASES AND THE EMPTY WORD

CANDIDATE	NT	COST
1	18	13
1	19	13
1	26	13
1	27	13
1	28	13
1	29	13
1	29	13
1	30	13
1	31	5
1	31	13
1	31	22
2	0	5
2	32	11
2	33	14
2	33	15
3	0	5

A member is chosen (in the implementation the stage of finding the selection cost for each LSC nonterminal is not separate). In this case candidate 3 is deleted (Figure 4.3.).

```
INTEGER ARRAY A (1..N+2 3..8);
                  ↑
                deleted
```

In Figure 4.1. column 1, the "D" is the deletion just carried out. This recovery eventually gives rise to another error indicated by R PROD15. This error is called "the first derived error from the first input error". The replacement is shown in Figure 4.4. A further error

FIGURE 4.3

Output for First Input Error, Deletion

FIGURE 4.4.

Replacement for First Derived Error from the First Input Error

is caused straightaway. For this second derived error, two condensations are carried out in the first backward move (Figure 4.5.) QUEST1 is now at position 5. The forward move is then started. The parser has in fact now recovered from the error in the array declaration.

The second input error (IF I>=4) puts the parser into the second backward move (Figure 4.6.), in which nothing happens. The candidate(s) are now defined. There is only one candidate, as QUEST1 has been absorbed:

```
CANDIDATES AT
BOTTOM    11
QUEST1    [illegible]
STACK     11
```

The only LSC action is to delete candidate 1.

```
CHOSEN MEMBER
  COST      3
  SYMBOL    [illegible]
  CAND      1
CANDIDATE     <>>
DELETED

TOP OF STACK AT     11
  71   -1   71   -1   3   -1   57   -1   25   -1   34

TOP OF STACK AT     9
  71   -1   71   -1   3   -1   57   -1   25

RHODES RECOVERY OVER, PARSING CONTINUES.
```

FIGURE 4.5.

Condensations for Second Derived Error from the First Input Error

```
ERROR
ENTER BACKMOVE
ENTER CONDENSE

TOP OF STACK AT   8
  71   -1   7(   -1    5   -1    9   67

LHS   -1

TOP OF STACK AT   8
  71   -1   7(   -1    5   -1    9   67

LEAVE CONDENSE
ENTER CONDENSE

TOP OF STACK AT   8
  71   -1   7(   -1    5   -1    9   67

LHS    7  DECLARE
LHS    (
RHS       ARRAYDEC

TOP OF STACK AT   8
  71   -1   7(   -1    5   -1    3   67

TOP OF STACK AT   7
  71   -1   7(   -1    5    7   67

TOP OF STACK AT   7
  71   -1   7(   -1    5    7   67

LEAVE CONDENSE
ENTER CONDENSE

TOP OF STACK AT   7
  71   -1   7(   -1    5    7   67

LHS    5  BLOCKHEAD
RHS       BLOCKHEAD  DECLARE  <:>
LHS    (

TOP OF STACK AT   5
  71   -1   7(   -1    5

LEAVE CONDENSE
LEAVE BACKMOVE
```

1

2

FIGURE 4.6.

The Second Input Error

```
SYMBOL  34  <>>

TOP OF STACK AT  9
  71   -1   70   -1    3   -1   57   -1   64

LHS    19   ARRAYID
RHS         <IDENTIFIER>
LHS    18   SIMPLEVAR
RHS         ARRAYID
LHS    20   VARIABLE
RHS         SIMPLEVAR
LHS    31   PRIMARY
RHS         VARIABLE
LHS    30   SECONDARY
RHS         PRIMARY
LHS    29   FACTOR
RHS         SECONDARY
LHS    28   TERM*
RHS         FACTOR
LHS    27   TERM
RHS         TERM*
LHS    25   EXPR**
RHS         TERM
SYMBOL  37  <=>

TOP OF STACK AT  11
  71   -1   70   -1    3   -1   57   -1   25   -1   34

ERROR
ENTER BACKMOVE  1
ENTER CONDENSE

TOP OF STACK AT  11
  71   -1   70   -1    3   -1   57   -1   25   -1   34

LHS    26   RELATIONOP
LHS    <

TOP OF STACK AT  11
  71   -1   70   -1    3   -1   57   -1   25   -1   34

LEAVE CONDENSE
LEAVE BACKMOVE
```

(only 1 candidate)

Recovery for the third and fourth errors overlaps. The third input error causes two derived errors, and the fourth error, no derived errors. The fourth input error is "correctly" dealt with before the two derived errors are encountered (Figure 4.1.). The second derived error gives seven condensations (Figure 4.7.). Five condensations occur in the first backward move, four in the first traverse, and one in the subsequent traverse.

There is only one traverse of the stack in the second backward move.

The parser finds the final configuration at the end of the second backward move of the second derived error. (Note that there is no forward move because the next symbol is $.)

FIGURE 4.7.

Condensing in the Second Derived Error from the Third Input Error

```
ERROR
ENTER BACKMOVE .J
ENTER CONDENSE

TOP OF STACK AT 10
 71  -1  7?  -1   3  -1  15  -1  22  67

LHS   -1

TOP OF STACK AT 10
 71  -1  7?  -1   3  -1  15  -1  22  67

LEAVE CONDENSE
ENTER CONDENSE

TOP OF STACK AT 10
 71  -1  7?  -1   3  -1  15  -1  22  67

LHS   15  SIMPLESTAT
RHS       PROGNOPAR

TOP OF STACK AT 10
 71  -1  7?  -1   3  -1  15  -1  22  67

TOP OF STACK AT 10
 71  -1  7?  -1   3  -1  15  -1  15  67

LHS   14  STATEMENT*
LHS   14  STATEMENT*
RHS       SIMPLESTAT

TOP OF STACK AT 10
 71  -1  7?  -1   3  -1  15  -1  15  67

TOP OF STACK AT 9
 71  -1  7?  -1   3  -1  15  14  67

LHS   14  STATEMENT*
RHS       IFTHENCL  STATEMENT*

TOP OF STACK AT 9
 71  -1  7?  -1   3  -1  15  14  67
```

1 2 3

continued

FIGURE 4.7: continued — Condensing in the Second Derived Error from the Third Input Error

```
TOP OF STACK AT       8
  71    -1    70    -1    3    -1    14    67

LHS      4    STATEMENT
LHS      0
RHS           STATEMENT*

TOP OF STACK AT       8
  71    -1    70    -1    3    -1    14    67

TOP OF STACK AT       7
  71    -1    70    -1    3    4    67

TOP OF STACK AT       7
  71    -1    70    -1    3    4    67

LEAVE CONDENSE     end of first traverse
ENTER CONDENSE

TOP OF STACK AT       7
  71    -1    70    -1    3    4    67

LHS      3    BLOCKBODY
RHS           BLOCKBODY   STATEMENT   <:>
LHS      

TOP OF STACK AT       5
  71    -1    70    -1    3

LEAVE CONDENSE     end of second traverse
LEAVE BACKMOVE
CORRECT EOF
ESYMBOL   71
ENTER BACKMOVE   1
ENTER CONDENSE

TOP OF STACK AT       6
  71    -1    70    -1    3    69

LHS      2    BLOCK
LHS      1    PROGRAM
RHS           BLOCKBODY   <END>
LHS      2    PROGRAM
RHS           <START>   BLOCK

LHS     -1

TOP OF STACK AT       3
  71    -1    1

LEAVE CONDENSE
```

CHAPTER 5

RESULTS

5.1. Introduction

Six test programs were written. Five of the test programs are based on examples of recovery given by Rhodes (1973). We can compare the recovery Rhodes obtained with the recovery given by our implementation.† The sixth test program consists of errors that Rhodes did not consider in detail in his thesis. A summary of the six programs is given in Table 5.1.

TABLE 5.1.

Summary of the Six Programs

	Name of Test Program	Appendix	Number of Input Errors	How Parse Terminated
1	Rhodes 1	G	4	after 2nd backward move
2	Rhodes 2	H	2	EMPTY SET
3	Rhodes 2A	I	1	in forward move
4	Rhodes 3	J	3	in forward move
5	Rhodes 4	K	1	normally
6	Systems Test	L	5	LOOPING

†Our implementation differs from that of Rhodes in at least five aspects. The first four have been mentioned previously.

1) Criteria for condensing are different (section 3.2.5.).
2) Replacement of a candidate by a nonterminal or a deletion instead of a phrase or a deletion (section 3.2.3.).

Each Appendix consists of a listing of the test program, output from SEMAN for the incorrect sentence superimposed over output from SEMAN for the corrected sentence, and a truncated trace of the parse (on OUTPUT). The truncated printing on OUTPUT gives:

a) Parser configuration at initial error
b) Stack after first backward move
c) Parser configuration after forward move
d) Stack after second backward move
e) The position of the candidates
f) The chosen member
g) The stack before and after the candidate is replaced/deleted.

Items a to g with the information on condensations from SEMAN, describe the recovery action in sufficient detail for this chapter.

From the two outputs from SEMAN the effect of error recovery can be seen in relation to the rightmost derivation for the correct sentence. It is of interest to see the type of deviation from the rightmost derivation that

---

3) The ability of $?_2$ to absorb $?_1$ (section 3.2.5.)
4) Amendments on the Algol subset (Chapter 4).
5) Rhodes omitted some symbol costs, Rhodes (1973, pp. 67-69).

an input error causes. Interaction between input errors may occur. Input errors may give rise to derived errors -- which may result in continuous error recovery, or looping.

Additional information for the system test is given in Appendix M. Constructor output for ALGSUB is given in Appendix A. This is useful when interpreting the parser action. Note that the symbol number of $ is 71 (number of symbols + 1).

5.2. Rhodes 1 (Appendix G)

a) INTEGER ARRAY A (1..N+2 3..8);

Input error 1 missing comma (,) between 2 and 3.

When the error is detected the stack is in configuration (1):

$ <START> BLOCKHEAD BOUNDSLIST EXPRESSION

<...> -EXPR** <+> <NUMBER>

The incoming symbol is <NUMBER>, this corresponds to the 3. This agrees with Rhodes (1973, p. 48).

The stack configuration is not changed during the first backward move. In the Rhodes implementation, Rhodes (1973, p. 49), EXPR** <+> <NUMBER> is condensed to EXPRESSION. No condensations occur in our implementation due to the stricter criteria used. <NUMBER> can be reduced to PRIMARY, which is Rhodes left stackable against <+>, but is not right stackable against <NUMBER> (details are given in section 4.2.2.).

Recovery is effected in our implementation by deleting <NUMBER> (the 3). This generates two derived errors. Recovery is complete after two condensations in the first backward move of the second derived error. The parser is left in the forward move.

b) IF I > = 4

Input error 2 > = instead of ≥

A character pair error is detected between > and =. When this error is detected, the forward move of the second derived error from the first input error is over. ?₁ has been absorbed in the forward move. The second backward move is carried out, this results in no change. Recovery is effected by deleting <>>.

From the output SEMAN, it can be seen that this recovery is ideal. There are no derived errors, the parser is put back into its normal state, and the derivation sequence is correct.

c) IF I = 1

THEN

C = D

GO TO LAB1 ;

Input error 3 C = D instead of C := D

Input error 4 no semicolon (;) after D.

At the point of detection of the third input error, the stack configuration at (2) is:

$ <START> BLOCKBODY IFTHENCL VARIABLE

The input symbol is <=>. Rhodes (1973, p. 47) states that the error configuration is:

"<endmarker> <blockbody> <if-then-cl> ?, with <primary> as the incoming token".

This is incorrect, the token at least must be a terminal symbol. Recovery is effected by replacing <=> by RELATIONOP. This does not give a correct recovery, and two derived errors are generated. However, before the first derived error is encountered, the fourth input error is detected. Recovery for the fourth input error is to delete <GO> <TO> <IDENTIFIER>, which is reasonable -- it causes no derived errors. A derived error caused by the recovery for the third input error is then detected. The recovery changes VARIABLE RELATIONOP EXPR* to PROCNOPAR. It is equivalent in input terms to changing C = D to STOP. The resulting stack configuration α, at (3), is as follows:

$ <START> BLOCKBODY INTHENCL PROCNOPAR <;>

<END> $ is in the input stream.

α <END> $ is a rightmost derivation. However, another error (the second derived error) is detected. This is because the normal parser configuration for the right sentential form α <END> $ is γ on the pushdown stack, where

γ<;> = α, and <;> <END> $ in the input stream.

Recovery is effected by condensing to the\ACCEPT configuration. Five condensations are carried out in the first backward move (see section 4.2.2.), <END> is stacked, and two more condensations are carried out in the second backward move. Note that condensations carried out in the stack with nonterminals above the point of condensation will result in a nonrightmost derivation (of a well formed sentence that is "close" to our badly formed one). In this case, the terminal symbol <;> is above the point of condensation for the first five, and the second two occur at the top of the stack.

5.3. Rhodes 2 (Appendix H)

a) INTEGER ARRAY FAR (1..25;

Input error 1 (1..25; instead of (1,.25);

After the second backward move the stack is in the configuration at (1):

$ <START> BLOCKHEAD BOUNDSLIST EXPRESSION <··>
EXPRESSION <;>

The position of the candidates is also marked at (1).

The stack configuration and position of candidates agrees with Rhodes (1973, p. 54). The recovery is excellent, and agrees with Rhodes' prediction. However as in Rhodes 1 (c), condensations are needed to effect complete recovery.

The reason is again that the input stream has advanced too far. This seems to be unavoidable as the error is caused by a missing symbol. The parser is thus left in a forward move -- with undesirable effect as is seen in (b).

b) (I<1) ∨ (I>10)

Input error 2 missing IF.

A character pair error is detected between <;> and <(>.

When the error is detected, the parser is in the forward move of the first derived error from the first input error. ?, is absorbed in the forward move. There is no action in the second backward move. The replacement set for candidate 1 is calculated, and is found to be empty -- Rhodes recovery fails. To see if a similar error would result in failure if it had the benefit of a forward move, and two backward moves, we ran Rhodes 2A.

5.4. Rhodes 2A (Appendix I)

One input error only -- a missing IF (c.f. Rhodes 2 (b))

The stack configuration when the error is detected is at (1):

$ <START> BLOCKBODY STATEMENT <;>

<(> is the input symbol. No condensations are carried out in the first backward move. BLOCKBODY STATEMENT

<;> is reduceable to BLOCKBODY, and BLOCKBODY is Rhodes left stackable against <START>, but it not right stackable against <(>. In the Rhodes implementation, the condensation is carried out (p. 53); he does not do a right stackability test. An error is detected straight away in the forward move -- see configuration (2). <(> ⋖ <IDENTIFIER>, and <(> reduces to LEFTPAREN, which is not Leinius left stackable against <;>. Also LEFTPAREN is not Leinius left stackable against BLOCKBODY, a configuration Rhodes' implementation should have passed through (p. 53). After the two backward moves and the forward move, the Rhodes configuration is:

"<endmarker> <blockbody> $?_1$ <expression> THEN $?_2$"

Presumably in the Rhodes implementation $?_1$ is a ⋖ throughout the duration of the forward move, Rhodes (1973, p. 45). In our implementation, it is only assumed that the ⋖ exists between the two original symbols.

There is no change in our second backward move. Our recovery is to change <(> to ARRAYNAME. This is not satisfactory, and six derived errors are generated before the recovery is complete. The stack configuration for the first derived error is given at (3):

$ <START> BLOCKBODY STATEMENT <;> ARRAYNAME EXPRESSION <)>.

The input symbol is <v>.

Two condensations occur in the first backward move -- see output from SEMAN. <;> -> ARRAYNAME, and BLOCKBODY STATEMENT <;> reduces to BLOCKBODY, which is stackable left and right. BLOCKBODY < ARRAYNAME, and ARRAYNAME EXPRESSION <)> reduces to VARIABLE which is stackable left and right. The stack configuration (4) then looks like:

$ <START> BLOCKBODY VARIABLE

The final action is to delete <v> TERM (or related to the input, v(I>10)) which results in the stack at (5). Note this is the same as (4).

The second derived error is found when VARIABLE is reduced to PRIMARY, which is not Leinius left stackable against BLOCKBODY. The best recovery found for the second derived error is to do "nothing" (replace VARIABLE by VARIABLE).

The third derived error results in VARIABLE <THEN> being deleted. In effect the whole of (I<1) v (I>10) THEN has been deleted. The fourth derived error replaces <GO> <TO> <IDENTIFIER> with SIMPLESTAT. The stack at (6) then looks like:

$ <START> BLOCKBODY SIMPLESTAT <ELSE> SIMPLESTAT

The first SIMPLESTAT corresponds to C := 2*I.

The fifth derived error replaces SIMPLESTAT <ELSE> SIMPLESTAT by STATEMENT*. The sixth derived error is caused, as before, by the input stream being too far

advanced. Condensations are required. The parser is left in the forward move.

The recovery is not at all elegant. To summarise, the recovery in effect slices out the pieces of code attached to the "conditional part" of the conditional statement. It would have been preferable, as Rhodes (1973, p. 57) suggests, if IF had been inserted.

5.5. Rhodes 3 (Appendix J)

a) INTEGER TOP1,BOT1,TOP2,BOT2,CODE RES;

Input error 1 missing comma (,) between CODE and RES.

Recovery from this error is good. The second <IDENTIFIER>, RES , is deleted. Insertion of a comma (,) would of course have been ideal. No condensations occur. Rhodes (1973, p. 43) obtained two condensations. Consider the stack configuration at (1). SIMPLEDEC AAAAO <IDENTIFIER> reduces to SIMPLEDEC, which is Rhodes left stackable against BLOCKHEAD, but is not right stackable against <IDENTIFIER>. Rhodes, as mentioned previously, does not do a right stackability test.

b) A 3*(J+K);

Input error 2 missing :=

The recovery action is to delete <NUMBER>, the 3. This results in two derived errors. The first derived error gives the deletion of <*> FACTOR which is equivalent

to deleting *(J+K). The second derived error results in the deletion of VARIABLE, the A. The input has been reduced to ;. Insertion of a := would of course have been much better. Rhodes (1973, p. 44) obtains one condensation. <IDENTIFIER> is reduced to VARIABLE -- see stack configuration at (2). As Rhodes states BLOCKBODY <: VARIABLE, and VARIABLE is reduced to PRIMARY, but BLOCKBODY has no relation to PRIMARY. The second condensation is not allowed. The condensation was not allowed in our implementation because VARIABLE is not right stackable against <NUMBER>.

c) I := J-4

K := N/2 ;

Input error 3 missing ;

The recovery action is to delete <NUMBER>, the 4. This results in two derived errors. The first derived error, at configuration (4) is a Type 2 phrase error. VARIABLE <:=> EXPRESSION can be reduced to SIMPLESTAT which is not Leinius left stackable against <->. Recovery for the first derived error is to replace VARIABLE <:=> EXPRESSION by PRIMARY. The stack configuration α, at (5) is then:

$ <START> BLOCKBODY VARIABLE <:=> EXPR** <-> PRIMARY <;>

This recovery is excellent. As before, with a

normal parse γ would be in the stack, where $\gamma <;>^* \alpha$. The second derived error gives condensations under the <;>, and the parser finishes in the forward move. The overall recovery for the input error is reasonable. However, it would have been better if a ; had been inserted.

Rhodes (1973, p. 58) obtains the following configuration after the forward and backward moves:

"<endmarker> <- <blockbody> <statement> $?_1$

<statement> ; $?_2$"

A lot is done in his forward and backward moves. Our forward move and backward moves achieve nothing. His configuration for recovery is better than ours -- at (3):

$ <START> BLOCKBODY VARIABLE <:=> EXPR**

<-> <NUMBER> <IDENTIFIER>

In his first backward move VARIABLE <:=> EXPR** <-> <NUMBER> is condensed to STATEMENT. Condensations are prevented in our implementation because PRIMARY, to which <NUMBER> can be reduced, is not right stackable against <IDENTIFIER> (remember that Rhodes does not check for right stackability).

5.6. Rhodes 4 (Appendix K)

One input error only : extra THEN

The recovery is perfect, a <THEN> is deleted. After the second backward move the stack configuration at (1) is:

$ <START> BLOCKBODY <IF> EXPRESSION
<THEN> <THEN>

The Rhodes configuration (p. 65) is:

"<endmarker> <blockbody> <if-then-cl> $?_1$ THEN $?_2$"

In his first backward move <IF> EXPRESSION <THEN> is condensed to IFTHENCL. This is prevented in our implementation as IFTHENCL is not right stackable against <THEN>. In this example, both recovery systems give the same result -- deletion of a THEN.

5.7. Systems Test (Appendix L)

a) P = Q ;

Input error 1 = instead of :=

The recovery action is to change <=> to RELATIONOP. This results in one derived error. The recovery for the derived error is to in effect delete P RELATIONOP Q. This recovery is quite acceptable.

b) R = S ;

Input error 2 = instead of :=

Identical recovery to (a).

Note that although the first and second input errors are very close, there is no interference.

c) If A = 6 THEN A := A+1 ; ELSE
A := A/2 ;

Input error 3 ; before ELSE

The recovery action is to delete <ELSE>. It would have been preferable to delete <;>, but <;> is not a candidate -- see stack at (1). Deleting <ELSE> is as good as deleting <;> as far as the parse is concerned, but the meaning of the program is somewhat affected.

d) Missing END and GOTO LAB1 ;

Input error 4 missing END

Input error 5 GOTO instead of GO TO

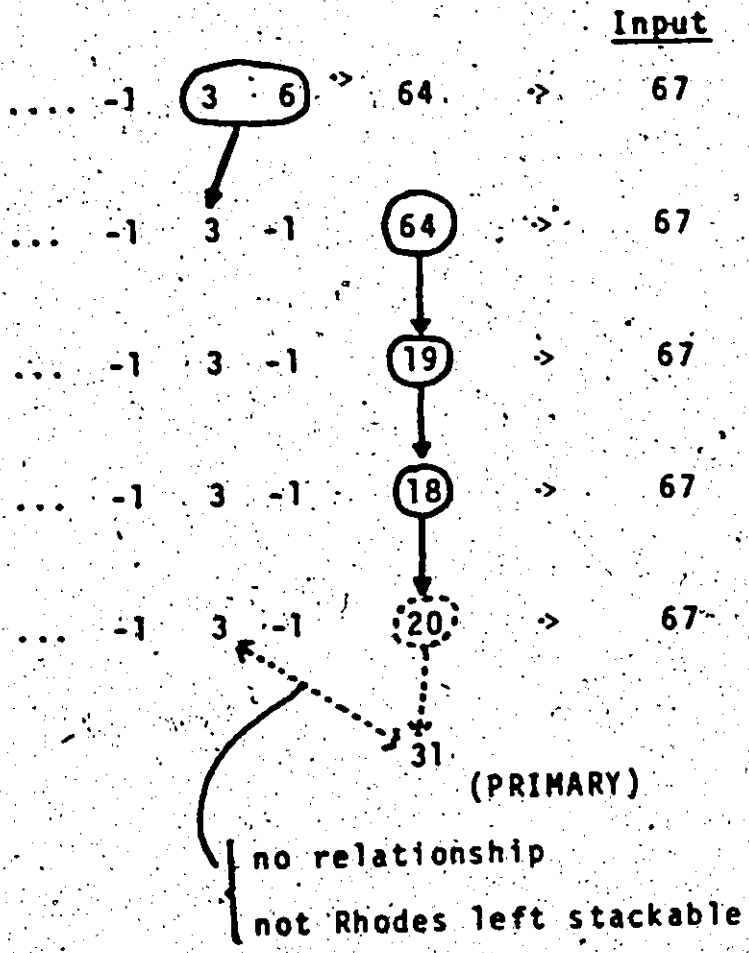
The fourth input error is not detected until the fifth input error has been detected, and recovery for it (the fifth) is complete.

i) GOTO LAB1

Recovery is effected by replacing <IDENTIFIER> (the GOTO) by LABELDEF. This results in one derived error. Four condensations occur in the first backward move -- see stack at (2). From the output for SEMAN, these condensations are:

PROD5 condense BLOCKBODY LABELDEF to BLOCKBODY
PROD64 condense <IDENTIFIER> to ARRAYID
PROD26 condense ARRAYID to SIMPLEVAR
PROD60 condense SIMPLEVAR to VARIABLE

In terms of symbol numbers:



The recovery action is to delete VARIABLE (equivalent to LAB1). The overall recovery is reasonable.

(ii) The missing END

For a full print of the parser action see Appendix M. The fourth input error is discovered when the input symbol is $ -- see configuration at (3). There is only one backward move, and ?1 is absorbed. The recovery action results in looping. Recovery is to replace BLOCKBODY

STATEMENT by BLOCKBODY. The LSC phrase giving the best selection cost is BLOCKBODY STATEMENT <;>. A <;> can be inserted at a cost of 1. Ideal recovery would replace BLOCKBODY STATEMENT by BLOCK. From Appendix M it can be seen that BLOCK (symbol number 2) is an LSC nonterminal (i.e. is a member of the only candidate's replacement set), with selection cost of 8 (from inserting <END>). As it is desirable to keep the cost of inserting <END> higher than the cost of inserting <;>, it is difficult to see how this recovery action can be avoided. A derived error is generated. The recovery action for the derived error is to replace BLOCKBODY with BLOCKBODY. This causes another derived error with the same parser configuration as the first derived error. Since recovery action does not change the parser configuration, LOOPING occurs. For the derived errors, it is difficult to see how to avoid the recovery action. Again the cost is 1. The LSC phrase giving the best selection cost is BLOCKBODY <;>. From Appendix M it can be seen that the selection cost for BLOCK is again 8 (from inserting <END>).

This situation is fairly serious as mis-matched ENDs are common.

5.8. Notes and Comments

An error summary is given in Table 5.2.

TABLE 5.2.

Error Summary

Test Data	Input Error Number	Number of Derived Errors	Number of Deletions	Number of Replacements	Number of Condensations
Rhodes 1	1	2	3	3	9
	2	-1†			
	3	2			
	4	0			
Rhodes 2	1	1	0	1	1
	2	-1†			
Rhodes 2A	1	6	2	4	4
Rhodes 3	1	0	5	1	11
	2	2			
	3	2			
Rhodes 4	1	0	1	0	0
Systems Test	1	-1	4	4*	4
	2	1			
	3	0			
	4	looping			
	5	1			
Totals	16 errors	16*	15	13	29

†recovery started in forward move

*ignoring the looping

NOTE: The total number of errors is 32, and the total number of recovery actions is 28. The difference is caused by four parses (Rhodes 1, 2, 2A and 3) terminating before a replacement or deletion could be carried out.

i) The only interaction between input errors is that sometimes after complete recovery from an input error, the parser was left in the forward move. This can have serious consequences (e.g. Rhodes 2), as the new input error is deprived of a forward and backward move. Also if $?_1$ is absorbed, the probability of Rhodes recovery failing increases (only one replacement set instead of three).

ii) The number of derived errors generated by an input error is "acceptable" -- the worst case is 6 for Rhodes 2A, see Table 5.2. The average number of derived errors for an input error is about 1. Looping was caused once -- in the systems test. Conditions for looping are outlined by Rhodes (1973, pp. 77, 78). For normal error recovery (1 forward move and two backward moves) the input stream is always advanced at least one symbol. The only situation where the input stream is not advanced during the process of error recovery is when the input symbol is $. We check for looping in this situation only. The looping occurred with a deficiency of ENDs. This is a common error and is a serious failing of the heuristic. We could see no easy way round it.

iii) Our criteria for condensing can be improved. The best combination of criteria is probably ours for condensations embedded in the stack, and Rhodes, for condensations at the top of the stack. The reason for this choice is that

errors must not be introduced into the stack. Using Rhodes criteria (i.e. no right stackability check) at the top of the stack is reasonable because often the input symbol is incorrect.

iv) The implementation seems heavy on deletions. This could be a reflection on the input data used.

v) Comparison with the Rhodes implementation is difficult due to the differences mentioned previously, but agreement, although not complete, was reasonable.

CHAPTER 6

CONCLUSIONS

In general the error recovery is adequate. LaFrance (1971, p. 9) has developed a statistic to measure the effectiveness of error recovery.

$$\text{effectiveness} = \frac{E + \frac{3}{4}G + \frac{1}{2}F + \frac{1}{4}P}{N+M} \cdot \frac{N}{N+X}$$

N -- number of input errors detected

M -- number of input errors missed

X -- number of derived errors.

The recoveries (N) are divided into excellent (E), good (G), fair (F) and poor (P). Effectiveness is "0 for a set of recoveries in which all errors were missed, and (the value) 1 for a set of recoveries in which all errors were detected with a rating of E, and no extras were created", LaFrance (1971, pp. 10, 12). It can be seen that the statistic is subjective. Our recovery gave an effectiveness of .36 (Table 6:1.). Omitting the worst test run (Rhodes 2A), effectiveness is .45. LaFrance (1971, p. 78, Table 3) shows recoveries with effectiveness from .093 to .891. To be able to draw any meaningful conclusions from such a statistic, a large number of test runs would have to be made for different languages, using both the Rhodes recovery and alternative recovery methods (e.g. the actual

TABLE 6.1.

Effectiveness of Recovery

Table 5.2. gives an error summary.

Test Program	Category†			
	E	G	F	P
Rhodes 1	2	0	2	0
Rhodes 2	1	1	0	0
Rhodes 2A	0	0	0	1
Rhodes 3	1	0	2	0
Rhodes 4	1	0	0	0
System test	1	3	0	1
Totals	6	4	4	2

Effectiveness for all Six Programs

N = 16
M = 0
X = 16

$$\text{effectiveness} = \frac{6 + \frac{1}{2}.4 + \frac{1}{4}.4 + \frac{1}{8}.2}{16+0} \cdot \frac{16}{16+16}$$

$$= \frac{11.5}{32} = .36$$

Effectiveness for Six Programs less Rhodes 2A

N = 15
M = 0
X = 10

$$\text{effectiveness} = \frac{6 + \frac{1}{2}.4 + \frac{1}{4}.4 + \frac{1}{8}.1}{15+0} \cdot \frac{15}{15+10}$$

$$= \frac{11\frac{1}{2}}{25} = .45$$

---

†based on number of derived errors. 0 derived errors -E; 1-G; 2-F; >2-P.

compiler for the language). Rhodes (1973, Chapter 8) submits specific syntax errors to the production PASCAL and PL/C compilers, and the compiler of James (1972).

The performance of the Rhodes recovery system could certainly be improved by experimenting with the criteria for condensing (section 5.8.). The Rhodes recovery system seems to be very stable. While developing the system, various mutations were tested. No matter what design criteria are used, the parser will continue to operate, and will eventually give reasonable recovery (with the notable exception of a deficiency of ENDs, which produced looping).

An obvious extension to the present system would be to provide error messages. Semantic interpretations of the parse could also be investigated. Optimal error recovery for practical recovery systems would perhaps be a fruitful area for further study.†

---

†In error correcting systems minimum distance is used as a criterion of optimality. However, as Rhodes (1973, p. 19), points out, minimum distance may be a poor metric for measurement of performance.

APPENDIX A

Constructor: i) program listing

ii) output (ALGSUB)

iii) dayfile ("job control")

DOCUMENTATION

CONSTRUCTOR FOR SIMPLE PRECEDENCE GRAMMAR

AUTHOR C.B.JOINS
DEPARTMENT OF APPLIED MATH
MC.MASTER UNIVERSITY
LANGUAGE PASCAL(DEC 1972 VERSION)
COMPUTER CDC 64
DATE JUNE 1974

ACKNOWLEDGEMENTS AND THANKS (IN ALPHABETICAL ORDER) TO :
C.BRYCE FOR ADVICE ON THE SCOPE OPERATING SYSTEM
DR.N.SOLNTSEFF FOR THE PROCEDURE WRITEINTEGER AND ADVICE ON PASCAL I/O
A.WALKER FOR THE BINARY TREE DISPLAY PACKAGE
DR.D.WOOD FOR GENERAL ADVICE ON SYSTEM DESIGN

ASSUMPTIONS MADE ABOUT GRAMMAR

1) REDUCED
2) CYCLE FREE

WE CHECK THAT THE GRAMMAR HAS NO E-PRODUCTIONS, IS FREE OF PRECEDENCE CONFLICTS, AND IS UNIQUELY INVERTIBLE

SUMMARY OF INPUT/OUTPUT FILES

	NAME	TYPE	DESCRIPTION
INPUT	INPUT	CODED	OUTPUT CONTROL(FIRST 9 CHARACTERS) AND SIMPLE PRECEDENCE GRAMMAR
OUTPUT	1) VALPAR	CODED	PRECEDENCE MATRIX AND/OR TERMINAL HASH TABLE AND/OR SYMBOL TABLE
	2) SEQFILE	BINARY	FILES 2,3, AND 4
	3) ROOTFILE	BINARY	CONSTITUTE THE SYNTAX GRAPH
	4) NODEFILE	BINARY	AND REVERSE SYNTAX GRAPH.
	5) OUTPUT	CODED	ERROR REPORTS AND OPTIONAL PRINTS

NOTE. VALPAR IS FORMATTED FOR THE CARD PUNCH.

OUTPUT OPTIONS

THE FIRST NINE CHARACTERS OF INPUT ARE USED TO CONTROL OUTPUT AS FOLLOWS :-

CHARACTER NUMBER	OUTPUT AFFECTED
1	SYMBOL TABLE (PRINT)
2	GRAMMAR (PRINT)

```
3                SETS (PRINT)
4                PRECEDENCE MATRIX (PRINT)
5                REVERSE SYNTAX GRAPH (PRINT)
6                GRAPHS (FOR PERMANENT FILE)
7                TERMINAL HASH TABLE (PUNCH)
8                PRECEDENCE MATRIX (PUNCH)
9                SYMBOL TABLE (PUNCH)

OUTPUT IS PRODUCED UNLESS THE CONTROL CHARACTER IS SET TO N
```

```
SUMMARY OF ERROR CODES

CODE NUMBER     MEANING
          1     TERMINAL DATA ERRORS
          2     UNDEFINED NO. TERMINAL
          3     HASH TABLE TOO SMALL
          4     GRAMMAR NOT UNIQUELY INVERTIBLE
          5     PRECEDENCE CONFLICTS
          6     SYMBOL TABLE TOO SMALL
          7     PARSE TOO SMALL
          8     NOT ENOUGH ROOM FOR NODES
```

```
PROCEDURE AND FUNCTION STRUCTURE

FIRST LEVEL          SECOND LEVEL          THIRD LEVEL

        ---SERVICE ROUTINES---
WRITEINTEGER
PNCHBLANKS
CARDEND
PNCHALF
NEWLINE
PUTBLANK
PUTCHAR
PUTSTRING
SPUTSTRING
PUTSEMAN
                        PUT2CHAR

        ---CONTROL---
CONTROL

        ---GRAMMAR INPUT---
INGRAMMAR
                        NEWNODES
                        HASHLO
                        ENDMARK
                        SETDEF
                        GETNBCHAR
                        GETSTRING
                                              TRUNCATE
                        GETCHAR
                        GETSEMAN
                                              GET2CHAR

        ---PRINT GRAMMAR---
PRINTGRAM
```

```
          ---PRINT SYMBOLS---
PRNTSY

          ---CREATE PRECEDENCE MATRIX---
CRPRMAT
                    LEFTRIGHT
                                        LSET
                                        RSET
                    CREATE
                                        ENTRY
                                        LTREL
                                        GTREL

          ---PRINT LEFT OR RIGHT SET---
PRNTLRSET
                    OUTMEM

          ---PRINT PRECEDENCE MATRIX---
PRNTMAT
                    RELOUT

          ---CREATE REVERSE SYNTAX GRAPH---
REVGRAPH
                    NEWNODERG
                    CREATELHS
                    BUILD
                    MATCHBLD

          ---PRINT REVERSE SYNTAX GRAPH---
PRNTREVGR
                    HEADOUT
                    DISPLAY
                                        SPACE
                                        PRNTBRANCH
                                        VISIT
                                        TRAVERSE

          ---PUNCH MATRIX OR SYMBOL TABLE---
PNCHSYMPREC

          ---PUNCH HASH TABLE---
PNCHALAYINT
                    PNCHPREC

          ---WRITE GRAPHS---
WRITEGRAPHS

          END OF DOCUMENTATION
```

GLOBALS

```
CONST
        MAXSE = 47 ;  MAXSY = 80 ;  MAXSY1 = 81 ;  MAXPREC = 9 ;
        HSHT = 77 ;
        HSHSE = 73 ;
        MAXNODE = 4.  ;
        MAXA = 2 ;  MAXC = 2[illegible] ;
```

```
        ALTERNATIVES FOR MAXA AND MAXC
        MAXA = 3 ;  MAXC = 3 ;
        MAXA = 5 ;  MAXC = 10 ;
        END OF ALTERNATIVES
    MAXSE     MAXIMUM NUMBER OF NON TERMINALS
    MAXSY     MAXIMUM NUMBER OF SYMBOLS
    MAXSY1    MAXSY + 1
    MAXPREC   MAXSY1 DIVIDED BY 10 AND ROUNDED UP
    HSHT      SIZE OF TERMINAL HASH TABLE
    HSHSE     SIZE OF NON TERMINAL HASH TABLE
    MAXNODE   MAXIMUM NUMBER OF NODES
    MAXC      MAXIMUM NUMBER OF CHARACTERS IN A SYMBOL
    MAXA      MAXC/3

    NOTE      THE NUMBER USED TO DEFINE GRAPH MUST BE SET TO
              THE VALUE OF MAXNODE

TYPE    POINT = ^GRAPH ;
        CHARARRAY = ARRAY[1..MAXC] OF CHAR ;
        ALFARRAY = ARRAY[1..MAXA] OF ALFA ;
        PSEDEFTYPE = ARRAY[1..MAXSE] OF POINT ;
        NODSDES = (TSYMBOL,TSEMANT,TSYMBOLR,TSYMBOLLHS) ;
        NODE = RECORD
                 CASE TAGFIELD : NODSDES OF
                   TSYMBOL    : (SYMBOL    : INTEGER ;
                                 PDEF      : POINT ;
                                 PALT      : POINT ;
                                 PSUCC     : POINT ;
                                 PSEM      : POINT ) ;
                   TSEMANT    : (SEMANT    : ALFA ;
                                 PCONT     : POINT ) ;
                   TSYMBOLR   : (SYMBOLR   : INTEGER ;
                                 PLHS      : POINT ;
                                 PALTR     : POINT ;
                                 PSUCCR    : POINT ) ;
                   TSYMBOLLHS : (SYMBOLLHS : INTEGER ;
                                 PSEMP     : POINT ) ;
                 END ;
        SETABTYPE = ARRAY[1..HSHSE] OF RECORD
                                         SE   : ALFARRAY ;
                                         POSN : INTEGER ;
                                       END ;
        TTABTYPE  = ARRAY[1..HSHT ] OF RECORD
                                         TERM : ALFARRAY ;
                                         POSN : INTEGER ;
                                       END ;
        PPPROTYPE = ARRAY[1..MAXSY] OF RECORD
                                         PSECSYM : POINT ;
                                         PLHS    : POINT ;
                                       END ;
        LRSETLIST = PACKED ARRAY[1..MAXSY] OF BOOLEAN ;
        LRSETTYPE = ARRAY[1..MAXSE] OF RECORD
                                         LSET : LRSETLIST ;
                                         RSET : LRSETLIST ;
                                       END ;
VAR     GRAPH       : CLASS 400 OF NODE ;
                      {SYNTAX GRAPH AND REV SYNTAX GRAPH}
        CHAY        : CHARARRAY ;
                      {FOR SYMBOL IN CHARACTERS}
        ALAY        : ALFARRAY ;
                      {FOR SYMBOL AFTER PACKING}
```

```
    WCHAR          : CHAR ;
                     ^WORKFIELD^
    PPRODST        : POINT ;
                     ^POINTER TO FIX START OF RULE^
    PWORK          : POINT ;
                     ^WORKFIELD^
    NODECT         : INTEGER ;
                     ^NODE COUNTER^
    INDEX          : INTEGER ;
                     ^WORKFIELD^
    SETABCT        : INTEGER ;
                     ^COUNTER FOR NON TERMINALS^
    TTABCT         : INTEGER ;
                     ^COUNTER FOR TERMINALS^
    ERRSW          : INTEGER ;
                     ^ERROR SWITCH^
    NOSYMS         : INTEGER ;
                     ^NUMBER OF SYMBOLS^
    NOTERM         : INTEGER ;
                     ^NUMBER OF TERMINALS^
    NONONTERM      : INTEGER ;
                     ^NUMBER OF NON TERMINALS^
    SETAB          : SETABTYPE ;
                     ^NON TERM HASH TABLE^
    TTAB           : TTABTYPE ;
                     ^TERM HASH TABLE^
    PPROD          : PPRODTYPE ;
                     ^ARRAY OF ROOTS FOR REV SYNTAX GRAPH^
    LRSET          : LRSETTYPE ;
                     ^FOR LEFT AND RIGHT SETS^
    PSEDEF         : PSEDEFTYPE ;
                     ^KEY TO SYNTAX GRAPH^
    CONCHRS        : ARRAY[1..9] OF CHAR ;
                     ^OUTPUT CONTROL CHARACTERS^
    PRECMAT        : ARRAY[1..MAXSY1,1..MAXPREC] OF ALFA ;
                     ^PRECEDENCE MATRIX^
    TEMCHARS       : ARRAY[1..11] OF CHAR ;
                     ^WORKARRAY^
    SYNTAB         : ARRAY[1..MAXSY] OF ALFAARRAY ;
                     ^SYMBOL TABLE^
    NODEPT         : ARRAY[1..MAXNODE] OF POINT ;
                     ^KEY FOR NODES OF GRAPH^
    VALPAR[OUT]    : TEXT ;
    SEDEFL[OUT]    : FILE OF PSEDEFTYPE ;
    ROOTFL[OUT]    : FILE OF PPRODTYPE ;
    NODEFLE[OUT]   : FILE OF NODE ;

          END GLOBALS
. . . . . .
          SERVICE ROUTINES

PROCEDURE WRITEINTEGER(X,BASE,L:INTEGER;VAR OUTF:TEXT);

{ THIS PROCEDURE WILL WRITE AN INTEGER AS A
  NUMBER WITH SPECIFIED BASE IN A FIELD OF
  WIDTH L. TRUNCATION WILL OCCUR FROM THE
  MOST SIGNIFICANT END IF THE ACTUAL LENGTH
  OF X PLUS ONE IS GT L!!.  THE BASE MUST BE
```

```
  < 17.  THE OUTPUT GOES TO FILE *OUTF* >

VAR
    I,J,S,T: INTEGER;
    DIGIT: ARRAY[0..23] OF INTEGER;
BEGIN
    S := ABS(X);
    I := 0;
    REPEAT
        T := S DIV BASE;
        DIGIT[I] := S - T*BASE;
        S := T;
        I := I + 1;
    UNTIL S = 0;
    IF X < 0
    THEN DIGIT[I] := 45
    ELSE DIGIT[I] := 43;
    FOR J := L-1 DOWNTO I+1 DO DIGIT[J] := 16;
    FOR J := L-1 DOWNTO 0 DO
        BEGIN
            S := DIGIT[J];
            IF (S < 15) & (S > 9) THEN S := S - 36;
            OUTF* := CHR(S+27); PUT(OUTF);
        END;
END;  { WRITEINTEGER }

. . . . . .

    PROCEDURE PNCHBLANKS(NO : INTEGER) ;
    BEGIN
      FOR INDEX := 1 TO NO DO
          BEGIN
              VALPAR* := ' ' ;
              PUT(VALPAR) ;
          END ;
    END ;

. . . . . .

    PROCEDURE CARDEND(BLANKCT : INTEGER ;
                      VAR CARDCT : INTEGER ;
                      CHAR1,CHAR2,CHAR3 : CHAR) ;
    BEGIN
      CARDCT := CARDCT + 1 ;
      {OUTPUT BLANKS TO BRING US TO COL 73}
      PNCHBLANKS(BLANKCT) ;
      VALPAR* := CHAR1 ;
      PUT(VALPAR) ;
      VALPAR* := CHAR2 ;
      PUT(VALPAR) ;
      VALPAR* := CHAR3 ;
      PUT(VALPAR) ;
      WRITEINTEGER(CARDCT,3,5,VALPAR) ;
      VALPAR* := EOL ;
      PUT(VALPAR) ;
    END ;

. . . . . .

    PROCEDURE PNCHALF(VAR HALF : ALFA) ;
    VAR     OUTCT : INTEGER ;
```

```
BEGIN
 VALPAR↑ := '====' ;
 PUT(VALPAR) ;
 UNPACK(HALF,TENCHARS,1) ;
 OUTCT := 0 ;
 REPEAT
   OUTCT := OUTCT + 1 ;
   VALPAR↑ := TENCHARS[OUTCT] ;
   PUT(VALPAR) ;
 UNTIL OUTCT=10 ;
 VALPAR↑ := '====' ;
 PUT(VALPAR) ;
END ;
```

```
PROCEDURE NEWLINE (VAR PRINTCT : INTEGER ;
                   VAR LINEST : INTEGER ) ;
BEGIN
 OUTPUT↑ := EOL ;
 PUT (OUTPUT) ;
 WRITE (' ',EOL) ;
 PRINTCT := LINEST ;
 FOR INDEX := 1 TO LINEST DO
  BEGIN
   OUTPUT↑ := ' ' ;
   PUT (OUTPUT)
  END
END ;
```

```
PROCEDURE PUTBLANK (     NNUM : INTEGER ;
                    VAR LINEST : INTEGER ;
                    VAR LINEFIN : INTEGER ;
                    VAR PRINTCT : INTEGER ) ;
BEGIN
 FOR INDEX := 1 TO NNUM DO
  BEGIN
   OUTPUT↑ := ' ' ;
   PUT (OUTPUT) ;
   PRINTCT := PRINTCT + 1 ;
   IF PRINTCT=LINEFIN
     THEN
          BEGIN
           NEWLINE (PRINTCT,LINEST) ;
           GOTO 1
          END ;
  END ;
1 : ;
END ;
```

```
PROCEDURE PUTCHAR (     HCHAR : CHAR ;
                   VAR LINEST : INTEGER ;
                   VAR LINEFIN : INTEGER ;
                   VAR PRINTCT : INTEGER ) ;
BEGIN
 OUTPUT↑ := HCHAR ;
 PUT (OUTPUT) ;
```

```
    PRINTCT := PRINTCT + 1 ;
    IF PRINTCT EQ LINEFIN
      THEN
        NEWLINE (PRINTCT,LINEST) ;
  END ;
```

```
  PROCEDURE PUTSTRING (VAR PRINTCT : INTEGER ;
                       VAR LINEST : INTEGER ;
                       VAR LINEFIN : INTEGER ;
                       VAR SYMLEN : INTEGER ) ;
  VAR     INDEXA,
          INDEXC : INTEGER ;
  BEGIN
    INDEXC := 1 ;
    FOR INDEXA := 1 TO MAXA DO
      BEGIN
        IF ALAY[INDEXA ] EQ E  E
          THEN
            GOTO 1 ;
        UNPACK (ALAY[INDEXA],CHAY,INDEXC) ;
        INDEXC := INDEXC + 1 ;
      END ;
    GOTO 2 ;
1 :
    INDEXA := INDEXA - 1 ;
2 :
    (*INDEXA HAS THE NUMBER OF ELEMENTS OF ALLAY USED *)
    (*IF COME OUT AT TOP OF LOOP,INDEXA = MAXA *)
    (*FIND OUT THE SIZE OF THE SYMBOL *)
    (*THE END OF A SYMBOL IS DETECTED BY A BLANK *)
    FOR INDEX := 2 TO INDEXA*1. DO
      IF CHAY[INDEX] EQ E E
        THEN
          GOTO 1 :
    SYMLEN := INDEX :
    GOTO 11 ;
1 :
    SYMLEN := INDEX - 1 ;
    (*IF COME OUT AT TOP OF LOOP, SYMLEN = INDEXA*1. *)
11 :
    (*CHECK TO SEE THAT SYMBOL CAN BE PRINTED ON CURRENT LINE *)
    IF PRINTCT + SYMLEN LE LINEFIN
      THEN
        BEGIN
          FOR INDEXC := 1 TO SYMLEN DO
            BEGIN
              OUTPUT* := CHAY[INDEXC] ;
              PUT (OUTPUT) ;
              PRINTCT := PRINTCT + 1
            END ;
          IF PRINTCT EQ LINEFIN
            THEN
              NEWLINE (PRINTCT,LINEST) ;
        END
      ELSE
        BEGIN
          (*I ASSUME HERE THAT IF CANNOT PRINT THE SYMBOL ON*
          (*THIS LINE, THEN CAN PRINT IT AFTER STARTING A NEW LINE*
          (*AS THE LARGEST SYMBOL ALLOWED IS 3, CHARS, THIS IS*
```

```
        *REASONABLE. NOTE HOWEVER IT ASSUMES THAT LINEST AND LINEFI
        *HAVE BEEN SET UP CORRECTLY. IF THEY HAVE NOT, A LOOP HERE*
        *IS POSSIBLE *
        NEWLINE (PRINTCT,LINEST  ) ;
        GOTO 11
       END ;
  END ;
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
  PROCEDURE SPUTSTRING ;
  *A SIMPLIFIED VERSION OF PUTSTRING*
  *ASSUMES CAN PRINT ON THIS LINE*
  VAR      INDEXA,
           INDEXC : INTEGER ;
           FINISH : BOOLEAN ;
  BEGIN
   INDEXC := 1 ;
   FOR INDEXA := 1 TO MAXA DO
    BEGIN
     IF ALAY[INDEXA] EQ =  =
       THEN
          GOTO 1 ;
     UNPACK(ALAY[INDEXA],CHAY,INDEXC) ;
     INDEXC := INDEXC + 10 ;
    END ;
   GOTO 2 ;
1  :
   INDEXA := INDEXA - 1 ;
2  :
   *INDEXA HAS NO OF ELEMENTS OF ALAY USED*
   *NOW PRINT*
   INDEX := 1 ;
   FINISH := FALSE ;
   REPEAT
    IF INDEX GT INDEXA*10
      THEN
         FINISH := TRUE
      ELSE
         IF CHAY[INDEX] EQ = =
           THEN
              FINISH := TRUE
           ELSE
              BEGIN
               WRITE(CHAY[INDEX]) ;
               INDEX := INDEX + 1 ;
              END ;
   UNTIL FINISH ;
  END ;
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
  PROCEDURE PUTSEMAN (P  ?  : POINT ;
                    VAR LINEST : INTEGER ;
                    VAR LINEFIN : INTEGER ;
                    VAR PRINTCT : INTEGER ) ;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  PROCEDURE PUT1CCHAR ;
  VAR      INDEX : INTEGER ;
```

```
BEGIN
 UNPACK(P^.SERVANT,TENCHARS,1) ;
 FOR INDEX := 1 TO 10 DO
  BEGIN
   PUTCHAR (TENCHARS[INDEX],LINEST,LINEFIN,PRINTCT) ;
   IF TENCHARS[INDEX] EQ ' '
     THEN
       GOTO 1 ;
  END ;
1 :
END ;
```

```
BEGIN
 PUTBLANK (2,LINEST,LINEFIN,PRINTCT ) ;
 PUTCHAR ('E',LINEST,LINEFIN,PRINTCT ) ;
 WHILE P <> NIL DO
  BEGIN
   PUTICCHAR ;
   P := P^.PCONT ;
  END ;
END ;
```

END OF SERVICE ROUTINES

CONTROL

```
PROCEDURE CONTROL ;
BEGIN
 CONCHRS[1] := INPUT^ ;
 FOR INDEX := 2 TO 4 DO
   BEGIN
     GET(INPUT) ;
     CONCHRS[INDEX] := INPUT^ ;
   END ;
END ;
```

END CONTROL

GRAMMAR INPUT

```
PROCEDURE INGRAMAR ;
READS THE GRAMMAR FROM CARDS, CREATES A SYNTAX GRAPH FOR THE
GRAMMAR, CHECKS THAT EACH SYNTACTIC ENTITY IS DEFINED
LABEL   999,21 ;
VAR
        NBCHAR : CHAR ;
        ERRORMESS : ALFA ;
        LETORDIG,
        LETDIGLT      : SET OF CHAR ;
        PNEW,
        POLD : POINT ;
        TABSRCH       : INTEGER ;
        EMALAY        : ALFARRAY ;
```

```
PROCEDURE NEWNODEG ;
BEGIN
 NODECT := NODECT + 1 ;
 IF NODECT GT MAXNODE
   THEN
        BEGIN
          ERRSW := 9 ;
          GOTO EXIT 201 ;
        END ;
 NEW(PNEW) ;
 NODEPT[NODECT] := PNEW ;
END ;
```

```
PROCEDURE HASHL3(    TABLE : ALFA ;
                     SIZE : INTEGER ) ;
*THE ALFA ARRAY TO BE HASHED IS CONTAINED IN ALAY*
*TABLE IS EITHER SETAB OR TTAB*
*IF WE FIND, WE SET TABSRCH TO THE LOCATION*
*IF WE FAIL, WE INSERT THE ITEM AND SET TABSRCH TO*
*THE NEGATIVE OF THE LOCATION.*
*IF NO ROOM IN TABLE, TABSRCH IS SET TO 0*
VAR       QUOT : INTEGER ;
          REM : INTEGER ;
          INUM : INTEGER ;
          FINISH : BOOLEAN ;
BEGIN
 FINISH := FALSE ;
 QUOT := 0 ;
 REM := 2 ;
 *HASH THE ALFA ARRAY ALAY*
 FOR INDEX := 1 TO MAXA DO
    BEGIN
      *CONVERT AN ALFA TO INTEGER USING ORD*
      INUM := ORD(ALAY[INDEX]) DIV 4.96 ;
      *GET THE ABSOLUTE VALUE*
      INUM := ABS(INUM) ;
      REM := REM * (INUM MOD SIZE) ;
      REM := REM MOD SIZE ;
      QUOT := QUOT + ((INUM DIV SIZE) MOD SIZE) ;
      QUOT := QUOT MOD SIZE ;
    END ;
*AT PRESENT REM IS IN THE RANGE 0 TO SIZE-1*
REM := REM + 1 ;
IF QUOT EQ 0
  THEN
       QUOT := 1 ;
TABSRCH := REM ;
IF TABLE EQ SETAB
  THEN
       *FOR THE TABLE SETAB*
       REPEAT
         IF SETAB[TABSRCH].SE EQ ALAY
           THEN
              *WE HAVE LOCATED THE ELEMENT IN THE TABLE*
              FINISH := TRUE
         ELSE
```

```
            IF SETAB[TABSRCH].POSN NE 0
              THEN
                  BEGIN
                    ♣WE HAVE A COLLISION, REHASH♣
                    TABSRCH := TABSRCH + QUOT ;
                    IF TABSRCH GT SIZE
                      THEN
                          TABSRCH := TABSRCH - SIZE ;
                    IF TABSRCH EQ REM
                      THEN
                          BEGIN
                            ♣WE HAVE A FULL TABLE♣
                            TABSRCH := 0 ;
                            FINISH := TRUE ;
                          END ;
                  END
              ELSE
                  BEGIN
                    ♣WE HAVE A NEW ENTRY, UPDATE SETAB♣
                    SETAB[TABSRCH].SE := ALAY ;
                    SETABCT := SETABCT + 1 ;
                    SETAB[TABSRCH].POSN := SETABCT ;
                    TABSRCH := -TABSRCH ;
                    FINISH := TRUE ;
                  END ;
        UNTIL FINISH
    ELSE
        ♣FOR THE TABLE TTAB♣
        REPEAT
          IF TTAB[TABSRCH].TERM EQ ALAY
            THEN
                ♣WE HAVE LOCATED THE ELEMENT IN THE TABLE♣
                FINISH := TRUE
            ELSE
                IF TTAB[TABSRCH].POSN NE 0
                  THEN
                      BEGIN
                        ♣WE HAVE A COLLISION, REHASH♣
                        TABSRCH := TABSRCH + QUOT ;
                        IF TABSRCH GT SIZE
                          THEN
                              TABSRCH := TABSRCH - SIZE ;
                        IF TABSRCH EQ REM
                          THEN
                              BEGIN
                                ♣WE HAVE A FULL TABLE♣
                                TABSRCH := 0 ;
                                FINISH := TRUE ;
                              END ;
                      END
                  ELSE
                      BEGIN
                        ♣WE HAVE A NEW ENTRY, UPDATE TTAB♣
                        TTAB[TABSRCH].TERM := ALAY ;
                        TTABCT := TTABCT - 1 ;
                        TTAB[TABSRCH].POSN := TTABCT ;
                        TABSRCH := -TABSRCH ;
                        FINISH := TRUE ;
                      END
        UNTIL FINISH ;
END ;
```

```
    PROCEDURE ENDMARK ;
    ⌐THE ENDMARKER $ IS INSERTED IN THE HASH TABLE⌙
    ⌐IN THE PARSER, WE LOOK UP $ IN THE HASH TABLE TO GET⌙
    ⌐ITS SYMBOL NO. ITS SYMBOL NO IS NOSYMS+1⌙
    ⌐NOTE THAT $ IS NOT IN THE SYMBOL TABLE⌙
    BEGIN
     TTABCT := NOSYMS + 2 ;
     ALAY := EMALAY ;
     ALAY[1] := '$' ;
     HASHLQ(ETTABE,MSHT) ;
    END ;

    PROCEDURE SETPDEF ;
    ⌐THIS PROCEDURE SETS THE PDEF POINTERS⌙
    ⌐ON THE NON TERMINALS⌙
    ⌐IF A NON TERMINAL DOES NOT APPEAR ON THE⌙
    ⌐LHS OF A PROD, PDEF IS LEFT AT ITS ARBITRARY VALUE,⌙
    ⌐AND AN ERROR MESSAGE IS OUTPUT.⌙
    ⌐WE ALSO RESET SYMBOL VALUES FOR TERMINALS⌙
    ⌐THIS IS DONE SO THAT WE CAN CLOSE THE GAP⌙
    ⌐IN THE TABLE SYMBOL BETWEEN TERMS AND NON TERMS⌙
    BEGIN
     SETABCT := 0 ;
110 : :
     SETABCT := SETABCT + 1 ;
     IF SETABCT GT NONCTERM
       THEN
            ⌐WE HAVE REACHED THE END OF NON TERMINALS⌙
            GOTO 2 ;
     IF PSEDEF[SETABCT] EQ NIL
       THEN
            BEGIN
             ⌐UNDEFINED NON TERM⌙
             ERRSW := 2 ;
             WRITE(' UNDEFINED NON TERMINAL  ') ;
             ALAY := SYMTAB[SETABCT] ;
             SPUTSTRING ;
             WRITE(EOL) ;
             GOTO 11 ;
            END;
     PWORK := PSEDEF[SETABCT] ;
     PPRODST := NIL ;
120 : :
     IF PWORK⌙.PDEF NE NIL
       THEN
            ⌐WE HAVE A NON TERMINAL, WE WANT TO SET UP PDEF⌙
            IF PSEDEF[PWORK⌙.SYMBOL] NE NIL
              THEN
                   ⌐APPEARS ON LHS OF PROD, CAN SET PDEF⌙
                   PWORK⌙.PDEF := PSEDEF[PWORK⌙.SYMBOL]
              ELSE
       ELSE
            ⌐RESET SYMBOL VALUE FOR A TERMINAL⌙
            PWORK⌙.SYMBOL := PWORK⌙.SYMBOL-MAXSY+NOSYMS ;
     IF PWORK⌙.PALT NE NIL
       THEN
```

```
        PPRODST := PWORK*.PALT ;
    IF PWORK*.PSUCC NE NIL
      THEN
          BEGIN
            PWORK := PWORK*.PSUCC ;
            GOTO 120
          END ;
    IF PPRODST EQ NIL
      THEN
          GOTO 110 ;
    PWORK := PPRODST ;
    PPRODST := NIL ;
    GOTO 120 ;
200 : ;
    *RESET SYMBOL VALUES IN THE HASH TABLE*
    FOR ITABCT := NONCHTER1*2 TO NOSYMS DO
      BEGIN
        ALAY := SYMTAB[ITABCT] ;
        HASHLO[ETTABCE,HSHT] ;
        TTAB[TABSRCH].POSN := TTAB[TABSRCH].POSN-MAXSY*NOSYMS;
      END ;
  END ;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  PROCEDURE GETNBCHAR ;
  BEGIN
    GET(INPUT) ;
    WHILE (INPUT* EQ B) OR (INPUT* EQ EOL) DO GET (INPUT)   ;
    NBCHAR := INPUT* ;
  END ;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  PROCEDURE GETSTRING(DELCHAR) ;
  *GETS THE REMAINDER OF THE STRING. THE FIRST CHARACTER OF THE     *
  *STRING HAS BEEN OBTAINED BY GETNBCHAR (IT IS IN INPUT*)         *
  *DEL BS > FOR A TERMINAL AND BLANK FOR A NON TERMINAL            *
  *BLANK AND EOL ARE USED TO DELIMIT A NON TERMINAL                *
  *> IS USED TO DELIMIT A TERMINAL. IF WE GET EOL FIRST - ERROR    *
  VAR
        SYMLEN,
        INDEXC,
        INDEXA : INTEGER ;

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

  PROCEDURE TRUNCATE ;
  VAR     INDEXC : INTEGER ;
  BEGIN
    WRITE(E TRUNCATED TO E) ;
    FOR INDEXC := 1 TO MAXC DO WRITE(CHAR[INDEXC]) ;
    WRITE(EOL) ;
    *NOW READ PAST END OF TRUNCATED SYMBOL*
    WHILE (INPUT* NE DEL) AND (INPUT* NE EOL) DO GET(INPUT) ;
    IF (DEL EQ B>E) AND (INPUT* EQ EOL)
      THEN
        *ERROR. TERMINAL IS NOT FINISHED WITH A >*
        BEGIN
          ERRORMESS := E< BUT NO >E ;
          GOTO EXIT 999 ;
```

```
        END ;
    END ;
```

```
BEGIN
  SYMLEN := 0 ;
  *INITIALISE CHAR*
  FOR INDEXC := 2 TO MAXC DO  CHAR[INDEXC] := ' ' ;
  WHILE (INPUT* NE DEL) AND (INPUT* NE EOL) DO
     BEGIN
        SYMLEN := SYMLEN* + 1 ;
        IF SYMLEN GT MAXC
          THEN
             TRUNCATE
          ELSE
             BEGIN
                CHAR[SYMLEN] := INPUT* ;
                GET(INPUT) ;
             END ;
     END ;
  IF (DEL EQ '>') AND (INPUT* EQ EOL)
    THEN
       *ERROR. TERMINAL IS NOT FINISHED WITH A >*
       BEGIN
          ERROMESS := '< BUT NO >' ;
          GOTO EXIT 999 ;
       END ;
  *WE DISCARD THE BLANK, BUT PACK THE > IF WE HAVE ROOM*
  IF SYMLEN LE MAXC
    THEN
       IF DEL EQ '>'
         THEN
            IF SYMLEN EQ MAXC
              THEN
                 TRUNCATE
              ELSE
                 CHAR[SYMLEN+1] := '>' ;
  *CHECK FOR EMPTY PRODUCTION*
  IF (CHAR[1] EQ '<') AND (CHAR[2] EQ '>')
    THEN
       BEGIN
          GET(INPUT) ;
          IF INPUT* EQ '>'
            THEN
               CHAR[3] := '>' ;
            ELSE
               BEGIN
                  ERROMESS := '<> ILLEGAL' ;
                  GOTO EXIT 999 ;
               END ;
       END ;
  INDEXC := 1 ;
  FOR INDEXA := 1 TO MAXA DO
     BEGIN
        PACK(CHAR,INDEXC,ALAY[INDEXA]) ;
        INDEXC := INDEXC + 1 ;
     END ;
END ;
```

```
PROCEDURE GETCHAR ;
BEGIN
 GET (INPUT) ;
 {= INDICATES NEXT CHAR OF INTEREST IS IN COL 1 OF NEXT CARD.
 {THIS IS NEEDED BECAUSE THE SCOPE SYSTEM MAY INSERT .
 {BLANKS BETWEEN THE LAST NON BLANK CHAR ON THE CARD AND EOL.
 IF INPUT. EQ EOL
   THEN
        BEGIN
         ERRORMESS := ESEM [ READE ;
         GOTO EXIT 999 ;
        END ;
 IF INPUT. EQ ELE
   THEN
        BEGIN
         {READ TO EOL.
         WHILE INPUT. NE EOL DO GET (INPUT) ;
         GET (INPUT) ;
        END ;
 NCHAR := INPUT. ;
END ;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROCEDURE GETSEMAN ;
VAR       MORESEM : BOOLEAN ;

. . . . . .

PROCEDURE GET10CHAR ;
BEGIN
 FOR INDEX := 1 TO 10 DO
  BEGIN
   GETCHAR ;
   TENCHARS[INDEX] := NCHAR ;
   IF NCHAR EQ ELE
     THEN
          BEGIN
           MORESEM := FALSE ;
           GOTO 1 ;
          END ;
  END ;
1 : ;
 PACK (TENCHARS,1,PNEW.SEMANT) ;
END ;

. . . . . .

BEGIN
 MORESEM := TRUE ;
 POLD := PNEW ;
 NEWNODEG ;
 PNEW.TAGFIELD := ISEMANT ;
 POLD.PSEM := PNEW ;
 GET10CHAR ;
 WHILE MORESEM DO
       BEGIN
        POLD := PNEW ;
        NEWNODEG ;
        PNEW.TAGFIELD := ISEMANT ;
```

```
            POLD^.PCONT := PNEW ;
            GETICCHAR ;
          END ;
      PNEW^.PCONT := NIL ;
    END ;

(*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*)
(*                 INGRAMMAR STATEMENT PART                          *)

    BEGIN
     LETORDIG := ['.','1','2','3','4','5','6','7','8','9',
                  'A','B','C','D','E','F','G','H','I','J',
                  'K','L','M','N','O','P','Q','R','S','T',
                  'U','V','W','X','Y','Z' ] ;
     SETABCT := 0 ;
     XTABCT := MAXSY + 1 ;
     NODECT := 0 ;
     ERRSW := 0 ;
     FOR INDEX := 1 TO MAXA DO
        EMALAY[INDEX] := ' ' ;
     (*INIT SYMTAB*)
     FOR INDEX := 1 TO MAXSY DO  SYMTAB[INDEX] := EMALAY ;
     LETDIGLT := ['<'] ;
     LETDIGLT := LETDIGLT + LETORDIG ;
     FOR INDEX := 1 TO HSHT DO
        BEGIN
         TTAB[INDEX].TSEM := EMALAY ;
         TTAB[INDEX].POSN := 0 ;
        END ;
     FOR INDEX := 1 TO HSHSET DO
        BEGIN
         SETAB[INDEX].SE := EMALAY ;
         SETAB[INDEX].POSN := 0 ;
        END ;
     FOR INDEX := 2 TO 1 DO  TENCHARS[INDEX] := ' ' ;
     (*THIS INITIALISATION IS NOT THEORETICALLY NEEDED*)
     (*HOWEVER, IF WE ATTEMPT TO PACK CERTAIN CHARACTERS*)
     (*THE ALFA VARIABLE IS UNDEFINED*)
     WRITE('1','VALIDATION OF GRAMMAR',EOL,'0',EOL) ;
1  : GETNBCHAR ;
     IF NBCHAR EQ '.'
       THEN
          GOTO 2 ;
     IF ¬ (NBCHAR IN LETORDIG )
      THEN
          BEGIN
           ERRORMESS := 'NOT NON TME ;
           GOTO 999
          END ;
     GETSTRING(' ') ;
     (*READ IN REMAINDER OF THE SYNTACTIC ENTITY ON L.H.S. OF
     (*PRODUCTION *)
     NEWNODEG ;
     PNEW^.TAGFIELD := TSYMBOL ;
     PPRODST := PNEW ;
     HASHLD('SETAB',HSHSE) ;
     IF TABSRCH EQ
      THEN
          BEGIN
           ERRSW := 3 ;
           WRITE(' MAKE MAXSE BIGGER',EOL) ;
```

```
        GOTO 200 ;
      END ;
  IF TABSRCH GT 0
    THEN
      *NOT A NEW ENTRY*
      IF PSEDEF[SETAB[TABSRCH].POSN] NE NIL
        THEN
          *THIS SE HAS BEEN ON LHS OF PROD PREV*
          BEGIN
            PWORK := PSEDEF[SETAB[TABSRCH].POSN] ;
            WHILE PWORK*.PALT NE NIL DO
              PWORK := PWORK*.PALT ;
            PWORK*.PALT := PNEW ;
          END
        ELSE
          *THIS SE HAS NOT BEEN ON THE LHS OF A PROD YET*
          *SET UP PSEDEF*
          PSEDEF[SETAB[TABSRCH].POSN] := PNEW
    ELSE
      BEGIN
        *WE HAVE A NEW ENTRY*
        *CHECK IF MAXSE EXCEEDED*
        IF SETABCT GT MAXSE
          THEN
            *MAXSE IS TOO SMALL*
            BEGIN
              ERRSW := 7 ;
              GOTO 200 ;
            END ;
        *CHECK IF SYNTAB IS FULL*
        IF SYNTAB[SETABCT] NE ERALAY
          THEN
            BEGIN
              *TABLE IS FULL*
              ERRSW := 6 ;
              GOTO 200 ;
            END ;
        *SET UP PSEDEF*
        PSEDEF[SETABCT] := PNEW ;
        *UPDATE SYNTAB*
        SYNTAB[SETABCT] := ALAY ;
      END ;
  GETNBCHAR ;
  IF NBCHAR NE EOF
    THEN
      IF NBCHAR NE EOL
        THEN
          BEGIN
            ERRORMESS := 'NOT =OR;' ;
            GOTO 999
          END
        ELSE
          BEGIN
            GETNBCHAR ;
            IF NBCHAR NE EOL
              THEN
                IF NBCHAR NE EOL
                  THEN
                    BEGIN
                      ERRORMESS := 'NOT =OR;' ;
                      GOTO 999
```

```
                    END
                ELSE
                    BEGIN
                     GETNBCHAR ;
                     IF NBCHAR NE ===
                       THEN
                          BEGIN
                           ERRORMESS := =NOT == ;
                           GOTO 999
                          END
                    END ;
            END ;
5   ! ;
    GETNBCHAR ;
    IF ~ (NBCHAR IN LETDIGLI )
      THEN
        BEGIN
         ERROMESS := =NOT SYMBOLE ;
         GOTO 999
        END ;
10  ! IF NBCHAR E? =<=
      THEN
        BEGIN
         GETSTRING(=>=) ;
         *WE HAVE A TERMINAL*
         PNEW*.PDEF := NIL ;
         HASHLO(=TTABE,HSHT) ;
         IF TABSPCH =0 (
           THEN
              BEGIN
               ERRSW := 3 ;
               WRITE(= MAKE MAXT -BIGGERE,EOL) ;
               GOTO 2.. ;
              END ;
         IF TABSPCH LT :
           THEN
              BEGIN
               *THE NEW ENTRY FOR TTAB HAS MADE IN HASHLO*
               *WE NOW HAVE TO UPDATE SYMTAB*
               *CHECK IF SYMTAB IS FULL*
               IF SYMTAB(TTABCT) NE EMALAY
                 THEN
                    BEGIN
                     *TABLE IS FULL*
                     ERRSW := 6 ;
                     GOTO 2)' ;
                    END ;
               SYMTAB(TTABCT) := ALAY ;
               TABSPCH := -TABSPCH ;
              END ;
         PNEW*.SYMBOL := TTAB(TABSPCH).POSN ;
        END
      ELSE
        BEGIN
         GETSTRING(= =) ;
         *WE HAVE A NON TERMINAL*
         *SET PDEF TO NON NIL*
         PNEW*.PDEF := PNEW ;
         HASHLO(=SETABE,HSHSE) ;
         IF TABSPCH =0 .
           THEN
```

```
                BEGIN
                 ERRSW := 3 ;
                 WRITE(E,MAKE MAXSE BIGGER,EOL) ;
                 GOTO 200 ;
                END ;
          IF TABSPCH LT 0
            THEN
                BEGIN
                 *CHECK IF MAXSE EXCEEDED*
                 IF SETABCT GT MAXSE
                   THEN
                       *MAXSE IS TOO SMALL*
                       BEGIN
                         ERRSW := 7 ;
                         GOTO 200 ;
                       END ;
                 *CHECK IF SYMTAB IS FULL*
                 IF SYMTAB[SETABCT] NE EMALAY
                   THEN
                       BEGIN
                         *TABLE IS FULL*
                         ERRSW := 6 ;
                         GOTO 200 ;
                       END ;
                 TABSPCH := -TABSPCH ;
                 *WE HAVE TO SET PSEDEF TO NIL. THIS IS NOT A LHS*
                 PSEDEF[SETABCT] := NIL ;
                 *WE HAVE TO UPDATE SYMTAB*
                 SYMTAB[SETABCT] := ALAY ;
                END ;
          PNEW*.SYMBOL := SETAB[TABSPCH].POSN ;
        END ;
    *EVERY PALT IS INITIALLY SET TO NIL *
    PNEW*.PALT := NIL ;
    GETNBCHAR ;
    IF (NBCHAR IN LETDIGLT)
      THEN
        BEGIN
         POLD := PNEW ;
         NEWNODEG ;
         PNEW*.TARGFLD := TSYMBOL ;
         POLD*.PSUCC := PNEW ;
         GOTO 10 ;
        END ;
    PNEW*.PSUCC := NIL ;
    *PSUCC MUST BE NIL FOR THIS NODE*
    IF NBCHAR EQ '/'
      THEN
        BEGIN
         PNEW*.PSEM := NIL ;
200 :    ;
         NEWNODEG ;
         PNEW*.TARGFLD := TSYMBOL ;
         *PALT IS SET FOR FIRST SYMBOL OF PROD*
         *IF ONLY ONE SYMBOL IN PROD, PALT IS RESET*
         *IT WAS ORIG SET TO NIL*
         PPRODST*.PALT := PNEW ;
         PPRODST := PNEW ;
         GOTO 5 ;
        END ;
    IF NBCHAR EQ ':'
```

```
      THEN
          BEGIN
            PNEW^.PSEM := NIL ;
            GOTO 1
          END ;
      IF NBCHAR EQ EOL
        THEN
          BEGIN
            {PALT AND PSUCC HAVE BEEN SET ON THE CURRENT NODE}
            {PSEM IS SET IN GETSEMAN}
            {GET IN SEMANTICS}
            GETSEMAN ;
            GETNBCHAR ;
            IF NBCHAR EQ EOL
              THEN
                GOTO 2 ;
            IF NBCHAR EQ EOL
              THEN
                GOTO 1 ;
            {IF NEITHER / OR ; FOLLOWS 1, THEN ERROR}
            ERRORMESS := 'INV FOL 1' ;
            GOTO 999 ;
          END ;
      {IF GET HERE, ERROR SITUATION. A SYMBOL, /, ; OR ( MUST }
      {FOLLOW A SYMBOL}
      ERRORMESS := 'FOL SYMBOL' ;
      {DROP THROUGH TO ERROR RECOVERY }
      {ERROR RECOVERY }
999 : WRITE(OUTPUT,ERRORMESS,NBCHAR) ;
      WRITE(EOL) ;
      ERRSW := 1 ;
      WHILE NBCHAR NE EOL DO
        BEGIN
          GETNBCHAR ;
          IF NBCHAR EQ EOL
            THEN
              GOTO 2 ;
        END ;
      GOTO 1 ;
2 :   ;
      NONONTERM := SETABOT ;
      NOTERM := MAXSY - ITABOT + 1 ;
      NOSYMS := NOTERM + NONONTERM ;
      {CLOSE UP GAP BETWEEN TERMS AND NON TERMS}
      FOR INDEX := MAXSY+NOTERM+1 TO MAXSY DO
        SYNTAB[INDEX-MAXSY+NOSYMS] := SYNTAB[INDEX] ;
      {CHECK ON THE ERROR SWITCHES}
      {POSSIBLE VALUES HERE ARE 0,1,3,6,7,9}
      IF ERRSW LE 1
        THEN
          IF ERRSW EQ 1
            THEN
              WRITE(OUTPUT,'TERMINAL DATA ERRORS',EOL)
            ELSE
              BEGIN
                WRITE(OUTPUT,'NO DATA ERRORS',EOL) ;
                SETBOFF ;
                {CHECK TO SEE IF ERRSW HAS BEEN SET TO 2}
                IF ERRSW EQ 2
                  THEN
                    WRITE(OUTPUT,'UNDEF NON TERM',EOL)
```

```
                ELSE
                    BEGIN
                      ENDMARK :
                      WRITE(EOF,'ALL NON TERMINALS DEFINED',EOL);
                    END ;
    WRITE(EOF,'NUMBER OF NON TERMINALS',NONONTERMS,EOL) ;
    WRITE(EOF,'NUMBER OF TERMINALS    ',NOTERMS,EOL) ;
    WRITE(EOF,'TOTAL NO OF SYMBOLS    ',NOSYMS,EOL) ;
  END ;

                END GRAMMAR INPUT
. . . . . .
                PRINT GRAMMAR

  PROCEDURE PRINTGRAM ;
  VAR
        PRINTCT : INTEGER ;
        LINEST : INTEGER ;
        LINEFIN : INTEGER ;
        SYMLEN : INTEGER ;
  BEGIN
    SETABCT := 0 ;
    LINEST := MAXC + 12 ;
    LINEFIN := 95 ;
    WRITE (OUTPUT,'GRAMMAR',EINPUT,EOL ) ;
    WRITE (OUTPUT,EOL) ;
1:  SETABCT := SETABCT + 1 ;
    IF SETABCT GT NONTERMS
      THEN
        BEGIN
          WRITE (OUTPUT,EOL ) ;
          WRITE (OUTPUT,'END OF',PRINTOUT,EOL) ;
          GOTO 2
        END ;
    IF PSEUDEF(SETABCT) <> NIL
      THEN
        GOTO 1 ;
    PWORK := PSEUDEF(SETABCT) ;
    PRINTCT := 0 ;
    WRITE(EOF,EOL) ;
    PUTBLANK (5,LINEST,LINEFIN,PRINTCT) ;
    ONE BLANK FOR LINE CONTROL
    ALAY := SYNTAB(SETABCT) ;
    PUTSTRING (PRINTCT,LINEST,LINEFIN,SYMLEN) ;
    PRINTS THE SYNTACTIC ENTITY ON THE L.H.S. OF PRODUCTION
    SYMLEN HOLDS LENGTH OF SYNTACTIC ENTITY IN CHARACTERS
    PUTBLANK (MAXC-SYMLEN,LINEST,LINEFIN,PRINTCT) ;
    PUTCHAR ('=',LINEST,LINEFIN,PRINTCT ) ;
    PUTCHAR ('=',LINEST,LINEFIN,PRINTCT ) ;
    PUTBLANK (3,LINEST,LINEFIN,PRINTCT) ;
    PPRODST := NIL ;
10: ;
    IF PRINTCT <> LINEST
      THEN
        PUTBLANK (2,LINEST,LINEFIN,PRINTCT) ;
```

```
    IF PWORK+.PALT NE NIL
      THEN
          PPRODST := PWORK+.PALT ;
    ALAY := SYMTAB[PWORK+.SYMBOL] ;
    PUTSTRING (PRINTCT,LINEST,LINEFIN ,SYMLEN) ;
    *PRINTS A SYMBOL ON THE R.H.S. OF A PRODUCTION *
    *SYMLEN IS NOT USED ON THIS CALL *
    IF PWORK+.PSUCC NE NIL
      THEN
         BEGIN
          PWORK := PWORK+.PSUCC ;
          GOTO 10
         END ;
    IF PWORK+.PSEM NE NIL
      THEN
         PUTSEMAN(PWORK+.PSEM,LINEST,LINEFIN,PRINTCT) ;
    IF PPRODST EQ NIL
      THEN
         BEGIN
          WRITE(EOL) ;
          GOTO 1
         END ;
    PUTBLANK (2,LINEST,LINEFIN,PRINTCT) ;
    PUTCHAR ('/',LINEST,LINEFIN,PRINTCT ) ;
    PWORK := PPRODST ;
    PPRODST := NIL ;
    GOTO 10 ;
20 : ;
   END ;

*                    END PRINT GRAMMAR                                   *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                    PRINT SYMBOLS                                       *

   PROCEDURE PRNTSY(DESC : ALFA ;
                    LOW : INTEGER ;
                    HIGH : INTEGER ) ;
   VAR      POSNO : INTEGER ;
            SYMCT : INTEGER ;
            INDEX : INTEGER ;
   BEGIN
    WRITE('1',DESC,EOL,' ',EOL) ;
    POSNO := 0 ;
    FOR SYMCT := LOW TO HIGH DO
       BEGIN
        IF POSNO EQ 0
          THEN
             WRITE(' ') ;
        WRITE('   ',SYMCT:3,' ') ;
        FOR INDEX := 1 TO MAXA DO WRITE(SYMTAB[SYMCT][INDEX]) ;
        POSNO := POSNO + 1 ;
        IF POSNO EQ 2
          THEN
             BEGIN
               WRITE(EOL,' ',EOL) ;
               POSNO := 0 ;
             END ;
       END ;
```

```
    IF POSNO EQ 1
      THEN
        WRITE(EOL) ;
    WRITE(EOF,EOL,' END OF PRINT',EOL) ;
  END;

                END PRINT SYMBOLS
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                CREATE PRECEDENCE MATRIX

  PROCEDURE CPRMAT ;

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *

  PROCEDURE LEFTRIGHT ;
  VAR     WI1 : INTEGER ;
          WI2 : INTEGER ;
          PATHREC : ARRAY[1..MAXSE] OF BOOLEAN ;

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

  PROCEDURE LSET(VAR PSTART : POINT) ;
  VAR       PNOW : POINT ;
  BEGIN
   PNOW := PSTART ;
   REPEAT
     *ADD THIS SYMBOL TO THE CURRENT SYMBOLS LEFT SET*
     LRSET[SETABCT].LSET[PNOW*.SYMBOL] := TRUE ;
     *CHECK IF LATEST SYMBOL IS TERM OR NON TERM*
     IF PNOW*.PDEF <> NIL
       THEN
         *WE HAVE A NON TERMINAL*
         *HAVE WE HAD IT BEFORE*
         IF PATHREC[PNOW*.SYMBOL] EQ FALSE
           THEN
             *FIRST TIME WE HAD IT*
             BEGIN
               PATHREC[PNOW*.SYMBOL] := TRUE ;
               *GO ONE LEVEL DEEPER*
               LSET(PNOW*.PDEF) ;
             END ;
     *SET UP NEXT ALTERNATIVE*
     PNOW := PNOW*.PALT ;
   UNTIL PNOW EQ NIL ;
  END ;

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

  PROCEDURE RSET(VAR PSTART : POINT) ;
  VAR       PNOW : POINT ;
            PALT : POINT ;
  BEGIN
   PNOW := PSTART ;
   REPEAT
     *SET PALT*
     PALT := PNOW*.PALT ;
     *GET LAST SYMBOL IN PRODUCTION*
     WHILE PNOW*.PSUCC <> NIL DO PNOW := PNOW*.PSUCC ;
```

```
        {ADD THE SYMBOL TO THE CURRENT SYMBOLS RIGHT SET }
        LRSET[SETABCT].RSET[PNOW^.SYMBOL] := TRUE ;
        {CHECK IF LATEST SYMBOL IS TERM OR NON TERM}
        IF PNOW^.PDEF <> NIL
          THEN
              {WE HAVE A NON TERMINAL}
              {HAVE WE HAD IT BEFORE}
              IF PATHREC[PNOW^.SYMBOL] EQ FALSE
                THEN
                    {FIRST TIME WE HAD IT}
                    BEGIN
                        PATHREC[PNOW^.SYMBOL] := TRUE ;
                        {GO ONE LEVEL DEEPER}
                        RSET[PNOW^.PDEF) ;
                    END ;
        PNOW := PALT ;
      UNTIL PNOW EQ NIL ;
    END ;

. . . . . .

    BEGIN
      {INITIALISE LEFT AND RIGHT SETS}
      FOR WI1 := 1 TO NONONTERM DO
          FOR WI2 := 1 TO NOSYMS DO
              BEGIN
                  LRSET[WI1].LSET[WI2] := FALSE ;
                  LRSET[WI1].RSET[WI2] := FALSE ;
              END ;
      FOR SETABCT := 1 TO NONONTERM DO
          BEGIN
              {INIT  RECORD OF NON TERMS LOOKED AT}
              FOR INDEX := 1 TO NONONTERM DO PATHREC[INDEX] := FALSE ;
              PATHREC[SETABCT] := TRUE ;
              LSET[PSEDEF[SETABCT]) ;
              {INIT  RECORD OF NON TERMS LOOKED AT}
              FOR INDEX := 1 TO NONONTERM DO PATHREC[INDEX] := FALSE ;
              PATHREC[SETABCT] := TRUE ;
              RSET[PSEDEF[SETABCT]) ;
          END ;
    END ;

. . . . . .

    PROCEDURE CREATE ;
    VAR       SYM1 : INTEGER ;
              SYM2 : INTEGER ;
              W1   : INTEGER ;
              W2   : INTEGER ;

. . . . . .

    PROCEDURE ENTRY (VAR IN1 : INTEGER ;
                     VAR IN2 : INTEGER ;
                         REL : CHAR    ) ;
    VAR       POS : INTEGER ;
              ELNO : INTEGER ;
    BEGIN
      {WORK OUT THE ELEMENT NO AND THE POSN IN THE ELEMENT}
      POS := IN2 MOD 10 ;
      IF POS EQ 0
```

```
    THEN
        POS := 10 ;
  ELNO := (IN2 + 9) DIV 10 ;
  {UNPACK CHOSEN ELEMENT}
  UNPACK(PRECMAT[IN1,ELNO],TENCHARS,1) ;
  IF (TENCHARS[POS] EQ = =)
              OR
     (TENCHARS[POS] EQ REL)
    THEN
      TENCHARS[POS] := REL
    ELSE
      BEGIN
        ERRSW := 5 ;
        TENCHARS[POS] := ECE ;
      END ;
  PACK(TENCHARS,1,PRECMAT[IN1,ELNO]) ;
END ;

{ . . . . . }

PROCEDURE LTREL ;
VAR     LEFTCT : INTEGER ;
BEGIN
  FOR LEFTCT := 1 TO NOSYMS DO
    IF LRSET[SYM2].LSET[LEFTCT] EQ TRUE
      THEN
        ENTRY(SYM1,LEFTCT,<=) ;
END ;

{ . . . . . }

PROCEDURE GTREL ;
VAR     LEFTCT : INTEGER ;
        RIGHTCT : INTEGER ;
BEGIN
  IF SYM2 LE NONTERM
    THEN
      {SECOND SYMBOL IS A NON TERMINAL}
      FOR RIGHTCT := 1 TO NOSYMS DO
        IF LRSET[SYM1].RSET[RIGHTCT] EQ TRUE
          THEN
            BEGIN
              ENTRY(RIGHTCT,SYM2,=>) ;
              FOR LEFTCT := 1 TO NOSYMS DO
                IF LRSET[SYM2].LSET[LEFTCT] EQ TRUE
                  THEN
                    ENTRY(RIGHTCT,LEFTCT,=>) ;
            END
          ELSE
    ELSE
      {SECOND SYMBOL IS A TERMINAL}
      FOR RIGHTCT := 1 TO NOSYMS DO
        IF LRSET[SYM1].RSET[RIGHTCT] EQ TRUE
          THEN
            ENTRY(RIGHTCT,SYM2,=>) ;
END ;

{ . . . . . }

BEGIN
  {INIT PREC MATRIX}
```

```
    INDEX := NOSYMS + 1 ;
    FOR W2 := 1 TO MAXPREC DO  PRECMAT[INDEX,W2] := =  = ;
    FOR W1 := 1 TO NOSYMS DO
       BEGIN
          FOR W2 := 1 TO MAXPREC DO
             PRECMAT[W1,W2] := =  = ;
          ENTRY(W1,INDEX,=>=) ;
          ENTRY(INDEX,W1,=<=) ;
       END ;
    WRITE(=1REPORT ON PREC CONFLICTS=,EOL) ;
    FOR SETARCT := 1 TO NONONTERM DO
       BEGIN
          PWORK := PSEUDF[SETARCT] ;
          REPEAT
             PPRODST := PWORK↑.PALT ;
             SYM2 := PWORK↑.SYMBOL ;
             WHILE PWORK↑.PSUCC <> NIL DO
                BEGIN
                   SYM1 := SYM2 ;
                   PWORK := PWORK↑.PSUCC ;
                   SYM2 := PWORK↑.SYMBOL;
                   {UPDATE MATRIX FOR =}
                   ENTRY(SYM1,SYM2,===) ;
                   IF SYM2 <= NONONTERM
                      THEN
                         {SYMBOL IS A NON TERMINAL}
                         {UPDATE MATRIX FOR <}
                         LTREL ;
                   IF SYM1 <= NONONTERM
                      THEN
                         {SYMBOL IS A NON TERMINAL}
                         {UPDATE MATRIX FOR >}
                         GTREL ;
                END ;
             {GET NEXT ALTERNATIVE PRODN}
             PWORK := PPRODST ;
          UNTIL PWORK = NIL ;
       END ;
    IF ERRSW <> 5
      THEN
         WRITE(=1=,=NO PREC CONFLICTS=,EOL)
      ELSE
         WRITE(=1=,=PREC CONFLICTS=,EOL) ;
    END ;
```

```
    BEGIN
     LEFTRIGHT ;
     CREATE ;
    END ;
```

```
                    END CREATE PRECEDENCE MATRIX
                    PRINT LEFT OR RIGHT SET

    PROCEDURE PRNTLRSET(LR : ALFA) ;
    VAR       PRINTCT : INTEGER ;
```

```
        LINEST  : INTEGER ;
        LINEFIN : INTEGER ;
        SYMLEN  : INTEGER ;
        COUNT   : INTEGER ;
```

```
PROCEDURE OUTMEM ;
BEGIN
 ALAY := SYMTAB[COUNT] ;
 PUTSTRING(PRINTCT,LINEST,LINEFIN,SYMLEN) ;
 IF PRINTCT NE LINEST
   THEN
      PUTBLANK(2,LINEST,LINEFIN,PRINTCT) ;
END ;
```

```
BEGIN
 IF LR EQ RIGHTE
   THEN
      WRITE(F1RIGHT SETE,EOL)
   ELSE
      WRITE(F1LEFT SETE,EOL) ;
 WRITE(EOL) ;
 LINEST := MAXC + 6 ;
 LINEFIN := 1 ;
 FOR SETARCT := 1 TO NOMONTERM DO
   BEGIN
      *PRINT SYMBOL WHOSE SET IT IS*
      WRITE(EOL,F1,EOL) ;
      PRINTCT := 1 ;
      PUTBLANK(6,LINEST,LINEFIN,PRINTCT) ;
      *ONE BLANK FOR LINE CONTROL*
      ALAY := SYMTAB[SETARCT] ;
      PUTSTRING(PRINTCT,LINEST,LINEFIN,SYMLEN) ;
      *PAD WITH BLANKS*
      PUTBLANK(MAXC-SYMLEN,LINEST,LINEFIN,PRINTCT) ;
      PUTCHAR(':',LINEST,LINEFIN,PRINTCT) ;
      PUTBLANK(2,LINEST,LINEFIN,PRINTCT) ;
      IF LR EQ RIGHTE
        THEN
           FOR COUNT := 1 TO NOSYMS DO
              IF LPSET[SETARCT].PSET[COUNT] EQ TRUE
                THEN
                   OUTMEM
                ELSE
        ELSE
           FOR COUNT := 1 TO NOSYMS DO
              IF LPSET[SETARCT].LSET[COUNT] EQ TRUE
                THEN
                   OUTMEM ;
   END ;
 WRITE(EOL,EOL,EOL,EOL,EOL) ;
 WRITE('END OF SET',EOL) ;
END ;
```

END PRINT LEFT OR RIGHT SET

PRINT PRECEDENCE MATRIX

```
PROCEDURE PRNTMAT ;
VAR       SPLITCT : INTEGER ;
          WORKCT  : INTEGER ;
          ELNO    : INTEGER ;
          DOWNCT  : INTEGER ;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
PROCEDURE RELOUT ;
VAR       OUTCT : INTEGER ;
BEGIN
 UNPACK(PRECMAT[DOWNCT,ELNO],TENCHARS,1) ;
 OUTCT := 1 ;
 REPEAT
    WRITE(=  =,TENCHARS[OUTCT],= =) ;
    OUTCT := OUTCT + 1 ;
 UNTIL (OUTCT EQ 11) OR (10*ELNO+OUTCT GT NOSYMS+11) ;
END ;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
BEGIN
 WRITE(=1   PRECEDENCE MATRIX=) ;
 SPLITCT := 1 ;
 REPEAT
    WRITE(EOL,=E=,EOL,=E=,EOL,=            =) ;
    *ASSEMBLE HEADING*
    WORKCT := SPLITCT ;
    REPEAT
       WRITE(WORKCT:3,= =) ;
       WORKCT := WORKCT + 1 ;
    UNTIL (WORKCT EQ SPLITCT+30) OR (WORKCT GT NOSYMS+1) ;
    WRITE(EOL,= =) ;
    FOR DOWNCT := 1 TO NOSYMS+1 DO
      BEGIN
         WRITE(EOL,= =,EOL,= =,DOWNCT:3,=     =) ;
         ELNO :=  SPLITCT DIV 10 ;
         REPEAT
            ELNO := ELNO + 1 ;
            RELOUT ;
         UNTIL (ELNO*10 EQ SPLITCT+29) OR (ELNO*10 GE NOSYMS+1) ;
      END ;
    SPLITCT := SPLITCT + 30 ;
 UNTIL SPLITCT GT NOSYMS+1 ;
 WRITE(EOL,=E=,EOL,=E=,EOL) ;
 WRITE(= =,=   END OF PRECEDENCE MATRIX=,EOL) ;
END ;
```

END PRINT PRECEDENCE MATRIX

CREATE REVERSE SYNTAX GRAPH

```
PROCEDURE REVGRAPH ;
LABEL     100 ;
VAR       POSPPROD : INTEGER ;
```

```
        PSYNGR : POINT ;
        PREVSG : POINT ;

PROCEDURE NEWNODEPG (VAR P : POINT) ;
BEGIN
 NODECT := NODECT + 1 ;
 IF NODECT GT MAXNODE
   THEN
       BEGIN
         ERRSW := 8 ;
         GOTO EXIT100 ;
       END ;
 NEW(P) ;
 NODEPT[NODECT] := P ;
END ;

PROCEDURE CREATELHS(VAR PLHS : POINT) ;
BEGIN
 NEWNODEPG(PWORK) ;
 PWORK^.TAGFIELD := TSYMBOLLHS ;
 PWORK^.SYMBOLLHS := SETADDT ;
 {PICK UP SEMANTICS IF THERE ARE ANY}
 {PSYNGR IS POINTING TO THE LAST SYMBOL IN THE PROD}
 PWORK^.PSEMR := PSYNGR^.PSEM ;
 {SET POINTER ON PREVIOUS NODE}
 PLHS := PWORK ;
END ;

PROCEDURE BUILD(VAR PLINK : POINT) ;
{ NOTE ON POINTERS                                                    }
{ ENTRY TO PROCEDURE                                                  }
{       PSYNGR POINTS TO FIRST NODE TO BE INSERTED                    }
{       PREVSG IS UNDEFINED                                           }
{ EXIT FROM PROCEDURE                                                 }
{       PSYNGR POINTS TO LAST NODE OF PROD                            }
{       PREVSG POINTS TO LAST NODE OF REV SYNTAX GRAPH                }
{              (NOT COUNTING THE LHS NODE)                            }
BEGIN
 NEWNODEPG(PREVSG) ;
 PREVSG^.TAGFIELD := TSYMBOL ;
 PLINK := PREVSG ;
 PREVSG^.SYMBOL := PSYNGR^.SYMBOL ;
 PREVSG^.ALT := NIL ;
 WHILE PSYNGR^.PSUCC <> NIL DO
  BEGIN
   PREVSG^.PLHS := NIL ;
   NEWNODEPG(PWORK) ;
   PWORK^.TAGFIELD := TSYMBOL ;
   PREVSG^.PSUCC := PWORK ;
   PREVSG := PWORK ;
   PSYNGR := PSYNGR^.PSUCC ;
   PREVSG^.SYMBOL := PSYNGR^.SYMBOL ;
   PREVSG^.ALT := NIL ;
   PREVSG^.PLHS := NIL ;
  END ;
```

```
    PREVSG↑.PSUCCP := NIL ;
    CREATELHS(PREVSG↑.PLHS) ;
  END ;

PROCEDURE MATCHOLD ;
A NOTE ON POINTERS
ENTRY TO PROCEDURE
      PSYNGR POINTS TO FIRST NODE OF PROD
      PREVSG IS UNDEFINED
EXIT FROM PROCEDURE
      PSYNGR POINTS TO LAST NODE MATCHED
      PREVSG POINTS TO LAST NODE MATCHED
  VAR     FINISH : BOOLEAN ;
  BEGIN
   FINISH := FALSE ;
   PSYNGR := PSYNGR↑.PSUCC ;
   PREVSG := PPROD[POSPPROD].PSECSYM ;
   REPEAT
    IF PSYNGR↑.SYMBOL EQ PREVSG↑.SYMBOLR
      THEN
          ↑MATCHING SYMBOLS↑
          IF PREVSG↑.PSUCCR EQ NIL
            THEN
                IF PSYNGR↑.PSUCC NE NIL
                  THEN
                      BEGIN
                       PSYNGR := PSYNGR↑.PSUCC ;
                       BUILD(PREVSG↑.PSUCCR) ;
                       FINISH := TRUE ;
                      END
                  ELSE
                      BEGIN
                       ↑ERROR, GRAMMAR NOT UNIQUELY INVERTIBLE↑
                       ↑I ASSUME NO RULE APPEARS MORE THAN ONCE↑
                       ERRSW := 4 ;
                       WRITE(E,'PROD# NOT UI, LHS =',E) ;
                       ALAY := SYMTAB[SETABCT] ;
                       SPUTSTRING ;
                       WRITE(EOL) ;
                       FINISH := TRUE ;
                      END
            ELSE
                IF PSYNGR↑.PSUCC NE NIL
                  THEN
                      BEGIN
                       ↑CONTINUE MATCHING, UPDATE POINTERS↑
                       PSYNGR := PSYNGR↑.PSUCC ;
                       PREVSG := PREVSG↑.PSUCCR ;
                      END
                  ELSE
                      BEGIN
                       CREATELHS(PREVSG↑.PLHS) ;
                       FINISH := TRUE ;
                      END
      ELSE
          ↑SYMBOLS DONT MATCH↑
          IF PREVSG↑.PALTR NE NIL
            THEN
                ↑TRY NEXT ALTERNATIVE↑
```

```
                PREVSG := PREVSG^.PALTR
            ELSE
                {NO MORE ALTERNATIVES}
                BEGIN
                 BUILD(PREVSG^.PALTR) ;
                 FINISH := TRUE ;
                END ;
    UNTIL FINISH ;
  END;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  BEGIN
   WRITE(E,'REPORT ON INVERTIBILITY',EOL,EOL,EOL) ;
   FOR INDEX := 1 TO MAXSY DO
    BEGIN
     PPROD[INDEX].PSECSYM := NIL ;
     PPROD[INDEX].PLHS := NIL ;
    END ;
   FOR SETABCT := 1 TO NONONTERM DO
    BEGIN
     PSYNGR := PSETDEF[SETABCT] ;
1    PPRODST := PSYNGR^.PALT ;
     POSPPROD := PSYNGR^.SYMBOL ;
     IF PSYNGR^.PSUCC = NIL
       THEN
            {WE HAVE A PROD WITH ONE SYMBOL ONLY}
            IF PPROD[POSPPROD].PLHS <> NIL
              THEN
                  BEGIN
                   ERRSW := 4 ;
                   WRITE(E,' ONE SYMBOL PROD NOT UI, LHS =',E) ;
                   ALAY := SYMTAB[SETABCT] ;
                   SPUTSTRING ;
                   WRITE(EOL) ;
                  END
              ELSE
                  CREATELHS(PPROD[POSPPROD].PLHS)
       ELSE
            {WE HAVE A PROD WITH MORE THAN ONE SYMBOL}
            {CHECK IF WE NEED TO MATCH}
            IF PPROD[POSPPROD].PSECSYM <> NIL
              THEN
                  {WE NEED TO MATCH AND BUILD}
                  MATCHBLD
              ELSE
                  BEGIN
                   PSYNGR := PSYNGR^.PSUCC ;
                   BUILD(PPROD[POSPPROD].PSECSYM) ;
                  END;
     IF PPRODST <> NIL
       THEN
           BEGIN
            PSYNGR := PPRODST ;
            GOTO 1 ;
           END ;
    END ;
   IF ERRSW <> 4
     THEN
         WRITE(E,'GRAMMAR UNIQUELY INVERTIBLE',EOL)
     ELSE
```

```
            WRITE(ECE,EGRAMMAR NOT UNIQUELY INVERTIBLEE,EOL) ;
100 ! ;
    END ;

                    END CREATE REVERSE SYNTAX GRAPH

                    PRINT REVERSE SYNTAX GRAPH

    PROCEDURE PRNTREVGR ;
    VAR     PPRODCT : INTEGER ;
            INDENT : INTEGER ;
            WIDTH : INTEGER ;
            NODELINE : INTEGER ;
            LINEFIN : INTEGER ;
            LINEST : INTEGER ;
            PRINTCT : INTEGER ;

    PROCEDURE HEADOUT ;
    BEGIN
     WRITE(E CONSTITUTES A RHS, ME) ;
     PWORK := PPROD[PPRODCT].PLHS ;
     ALAY := SYNTAB[PWORK^.SYMROLLHS] ;
     SPUTSTRING ;
     ANOW CHECK FOR SEMANTICS^
     IF PWORK^.PSEMR <> NIL
       THEN
         ^WE HAVE SEMANTICS^
         BEGIN
          PRINTCT := 0 ;
          PUTSEMA(PWORK^.PSEMR,LINEST,LINEFIN,PRINTCT) ;
         END;
     WRITE(EOL) ;
    END ;

PROCEDURE DISPLAY(ROOT:POINT;POINT:INDENT,WIDTH,NODELINE:INTEGER);
PURPOSE: TO DISPLAY A BINARY TREE IN A READABLE FORMAT

GLOBAL TYPE(S)
     POINT------VARIABLES OF THIS TYPE ARE POINTERS TO TREE NODES

CONST
        PRINTLIM = 33;
TYPE
        DIRECTION = (RIGHT,LEFT) ;
        BRARY = ARRAY[1..PRINTLIM] OF BOOLEAN;
VAR
        F:BOOLEAN;
        BRPRINT:BRARY;
```

```
LOCAL CONSTANT(S)

     PRINTLIN--THE DIMENSION OF THE BOOLEAN ARRAY PRARRAY. IT INDICATES
               THE NUMBER OF LEVELS OF THE BINARY TREE WHICH CAN BE
               PRINTED ON A PAGE AND MUST BE DETERMINED BY THE USER.

LOCAL VARIABLE(S)

     F---------BOOLEAN VARIABLE INDICATING IF:
               TRUE :SEGMENTS OF BRANCHES SHOULD BE PRINTED IN THE
                     NEXT PRINT LINE
               FALSE:THE KEY OF THE NEXT NODE SHOULD BE PRINTED IN THE
                     NEXT PRINT LINE
LOCAL TYPE
     DIRECTION--A SCALAR TYPE USED TO INDICATE THE DIRECTIONS WHICH
               MAY BE FOLLOWED FROM A NODE IE DIRECTION = (RIGHT,LEFT)

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

PROCEDURE SPACE(I:INTEGER);

PURPOSE: TO PRINT THE NUMBER OF SPACES INDICATED BY ITS PARAMETER

VAR J:INTEGER;
BEGIN
     FOR J := 1 TO I DO WRITE(' ')
END;

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

PROCEDURE PRNTBRANCH(LEVEL:INTEGER;BRPRINT:BRARY);

PURPOSE: TO PRINT THE CHARACTERS (COLONS) OF THE SEGMENTS OF THE
         BRANCHES BETWEEN A NODE JUST VISITED AND THE NEXT NODE TO BE
         VISITED (THIS IS A DISTANCE OF WIDTH PRINT LINES)

VAR N:INTEGER;
BEGIN
     FOR N := 1 TO WIDTH DO
          BEGIN

               {CARRIAGE CONTROL AND INITIAL SPACING TO FIRST BRANCH-
                CHARACTER POSITION}

               SPACE(NODELINE);

               {PRINT A BRANCH CHARACTER (COLON) AT THIS POSITION IF A
                BRANCH EXISTS (BRPRINT[P] = TRUE) WITH REQUIRED SPACING TO
                NEXT POTENTIAL BRANCH POSITION}

               FOR P := 1 TO LEVEL DO
                    IF BRPRINT[P]
                         THEN
                              BEGIN
                                   WRITE(':');
                                   SPACE(INDENT - 1)
```

```
                END
            ELSE SPACE(INDENT);
        WRITE(EOL)
    END;

    *RESET FLAG INDICATING NEXT PRINT LINE WILL CONTAIN A KEY*

    F := FALSE
END;

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

PROCEDURE VISIT(P:POINT;LEVEL:INTEGER;BRPRINT:BRARY);

PURPOSE: TO PRINT THE KEY OF THE NODE POINTED TO BY P. HOWEVER IT MAY
         ALSO NECESSARY TO (A) PRINT CHARACTERS OF PRECEDING BRANCHES
         (COLONS) SO AS THEY ARE DISPLAYED AS CONTINUOUS BRANCHES
         (B) PRINT FILLER-CHARACTERS (MINUS SIGNS) PRECEDING THE KEY TO
         LINK IT TO ITS FATHERS BRANCH (C) PRINT FILLER-CHARACTERS
         FOLLOWING THE KEY TO LINK IT TO ITS SONS BRANCH

VAR I,J:INTEGER;
BEGIN

    *CARRIAGE CONTROL*

    SPACE(1);

    *IF THE NODE IS NOT THE ROOT NODE,PRINT AND FILLER-CHARACTERS
      IN THE SAME PRINT LINE AS THE PRESENT KEY*

    IF P <> ROOT
        THEN

            *TWO CASES ARISE: EITHER NODELINE<INDENT OR NODELINE>INDENT
             (AND BY DEFINITION NODELINE<2*INDENT)*

            IF NODELINE < INDENT
                THEN

                    *IF NODE TO BE VISITED IS A SON OF THE ROOT NODE
                     (LEVEL = 2),NO BRANCH-CHARACTERS WILL BE PRINTED.
                     HENCE,SPACE TO THE FIRST PRINT POSITION OF THE KEY*

                    IF LEVEL = 2
                        THEN SPACE(INDENT)
                        ELSE
                            BEGIN

                                *SPACE TO THE FIRST POTENTIAL BRANCH-
                                 CHARACTER POSITION*

                                SPACE(NODELINE - 1);

                                *PRINT A BRANCH-CHARACTER (BRPRINT =
                                 TRUE) WITH REQUIRED SPACING TO THE
                                 NEXT POTENTIAL BRANCH-CHARACTER
                                 POSITION*

                                FOR I := 1 TO LEVEL - 2 DO
                                    IF BRPRINT[I]
```

```
                                THEN
                                    BEGIN
                                        WRITE(F,' ');
                                        SPACE(INDENT - 1)
                                    END
                                ELSE SPACE(INDENT);
                            IF BRPRINT[LEVEL - 1]
                                THEN
                                    BEGIN
                                        WRITE(F,' ');
                                        SPACE(2*INDENT - NODELINE)
                                    END
                                ELSE SPACE(2*INDENT - NODELINE);
                        END
                ELSE
                    BEGIN
                        *SPACE TO FIRST POTENTIAL BRANCH-CHARACTER
                         POSITION*
                        SPACE(NODELINE - 1);
                        *PRINT A BRANCH-CHARACTER (BRPRINT[I] = TRUE)
                         WITH REQUIRED SPACING TO THE NEXT POTENTIAL
                         BRANCH-CHARACTER POSITION*
                        FOR I := 1 TO LEVEL - 1 DO
                            IF BRPRINT[I]
                                THEN
                                    BEGIN
                                        WRITE(F,' ');
                                        SPACE(INDENT - 1)
                                    END
                                ELSE SPACE(INDENT);
                        *PRINT FILLER-CHARACTERS (MINUS SIGNS) BEFORE
                         KEY IF NECESSARY*
                        SPACE(1);
                        IF INDENT - NODELINE GT 1
                            THEN
                                BEGIN
                                    FOR I := 1 TO INDENT-NODELINE-1 DO
                                        WRITE(F,'-');
                                    SPACE(1)
                                END
                    END;
*PRINT THE KEY*
ALAY := SYNTAB[P*.SYMBOL];
SPUTSTRING;
IF P*.PLHS NE NIL
  THEN
    *WE ARE AT THE END OF A PROD*
    *PRINT THE LHS AND SEMANTICS(IF PRESENT)*
    BEGIN
      WRITE(F,' ');
      PWORK := P*.PLHS;
      ALAY := SYNTAB[PWORK*.SYMBOLLHS];
      SPUTSTRING;
      IF PWORK*.PSEMA NE NIL
```

```
            THEN
                (*WE HAVE SEMANTICS*)
                BEGIN
                 PRINTCT := 0 ;
                 PUTSEMA(PWORK^.PSEMR,LINEST,LINEFIN,PRINTCT) ;
                END ;
            END ;
    WRITE(EOL) ;

    (*SET FLAG INDICATING NEXT WIDTH PRINT LINES WILL CONTAIN BRANCH-
      CHARACTERS*)

    B := TRUE
END;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

PROCEDURE TRAVERSE(P:POINT;LEVEL:INTEGER;WAY:DIRECTION);

PURPOSE: TO PERFORM A REVERSE POSTORDER TRAVERSAL OF THE BINARY TREE
         AND INITIATE THE PRINTING OF KEYS AND BRANCHES

BEGIN
    (*WHEN P IS NIL THE TRAVERSAL CAN PROCEED NO FURTHER*)
    IF P <> NIL
        THEN
            BEGIN
                CASE WAY OF
                    RIGHT:BRPRINT[LEVEL] := FALSE;
                    LEFT :BRPRINT[LEVEL] := TRUE
                END;
                TRAVERSE(P^.PSUCCR,LEVEL+1,RIGHT) ;
                IF F THEN
                PRINTBRANCH(LEVEL,BRPRINT);
                CASE WAY OF
                    LEFT :BRPRINT[LEVEL] := FALSE;
                    RIGHT:BRPRINT[LEVEL] := TRUE
                END;
                VISIT(P,LEVEL,BRPRINT);
                TRAVERSE(P^.PALTP,LEVEL+1,LEFT) ;
                IF F THEN
                PRINTBRANCH(LEVEL,BRPRINT);
            END
END;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

BEGIN (*DISPLAY*)
    (*INITIALIZE*)
    F := FALSE;
    (*RESTRICT NODELINE TO BE LESS THAN 2*INDENT*)
    IF NODELINE GE 2*INDENT
        THEN NODELINE := 2*INDENT - 1;
    TRAVERSE(ROOT,0,RIGHT)
END;(*DISPLAY*)
```

```
BEGIN
 *WE ARE NOT USING THE NEWLINE FACILITY*
 LINEST := 1 ;
 LINEFIN := 999 ;
 INDENT := 8 ;
 WIDTH := 2 ;
 NODELINE := 4 ;
 WRITE(=0=,EOL) ;
 WRITE(='1PRINT OF REVERSE GRAPH=,EOL) ;
 FOR PPRODCT := 1 TO MAXSY DO
   IF PPROD[PPRODCT].PSECSYM NE NIL
     THEN
         BEGIN
          WRITE(=0=,EOL,=0=,EOL,=0=,EOL) ;
          WRITE(= ROOT OF FOLLOWING TREE =) ;
          ALAY := SYMTAB[PPRODCT] ;
          SPUTSTRING ;
          WRITE(EOL) ;
          IF PPROD[PPRODCT].PLHS NE NIL
            THEN
                *WE HAVE A ONE SYMBOL PROD AS WELL*
                BEGIN
                 WRITE(= THIS SYMBOL=) ;
                 *OUTPUT REST OF INFO FOR THIS PROD*
                 HEADOUT ;
                END ;
          WRITE(=0=,EOL) ;
          DISPLAY(PPROD[PPRODCT].PSECSYM,INDENT,,
                  WIDTH,NODELINE) ;
         END
     ELSE
         *WE HAVE NO TREE, CHECK FOR A ONE SYMBOL PROD*
         IF PPROD[PPRODCT].PLHS NE NIL
           THEN
                *WE HAVE A ONE SYMBOL PROD*
                BEGIN
                 WRITE(=0=,EOL,=0=,EOL,=0=) ;
                 ALAY := SYMTAB[PPRODCT] ;
                 SPUTSTRING ;
                 HEADOUT ;
                END ;
 WRITE(=0=,EOL) ;
 WRITE(= END OF PRINT OF REVERSE GRAPH=,EOL) ;
 WRITE(=1=,EOL) ;
END;
```

END PRINT REVERSE SYNTAX GRAPH

PUNCH MATRIX OR SYMBOL TABLE

```
PROCEDURE PNCHSYMPREC(LIM1,LIM2 : INTEGER ;
                      CHAR1,CHAR2,CHAR3 : CHAR ;
                      IDENT : ALFA ) ;
```

USED TO PUNCH OUT A TWO DIML ARRAY OF ALFAS
THE PRECEDENCE MATRIX AND SYMBOL TABLE BOTH FIT THIS CATEGORY

```
VAR       CT1 : INTEGER ;
          CT2 : INTEGER ;
          CARDCT : INTEGER ;
          ELCT : INTEGER ;
          HALF : ALFA ;
BEGIN
 CARDCT := 0 ;
 ELCT := 0 ;
 FOR CT1 := 1 TO LIM1 DO
    FOR CT2 := 1 TO LIM2 DO
       BEGIN
          ELCT := ELCT + 1 ;
          IF ELCT EQ 1
            THEN
                *START OF NEW CARD. PUNCH 6 BLANKS*
                PRCHBLANKS(6) ;
          *SET UP ELEMENT FROM EITHER SYMTAB OR PREC MATRIX*
          IF IDENT EQ 'MATRIX'
            THEN
                HALF := PRECMAT[CT1,CT2]
            ELSE
                HALF := SYMTAB[CT1][CT2] ;
          *OUTPUT 'HALF'*
          PRCHALF(HALF) ;
          *NOW CHECK FOR LAST ELEMENT*
          IF (CT1 EQ LIM1) AND (CT2 EQ LIM2)
            THEN
                CARDEND(67-ELCT*13,CARDCT,CHAR1,CHAR2,CHAR3)
            ELSE
                BEGIN
                    VALPAR* := ',' ;
                    PUT(VALPAR) ;
                    IF ELCT EQ 5
                      THEN
                          *END OF CARD*
                          BEGIN
                              CARDEND(1,CARDCT,CHAR1,CHAR2,CHAR3) ;
                              ELCT := 0 ;
                          END ;
                END ;
       END ;
 WRITE('PUNCH ',IDENT,' ',CARDCT:4,' CARDS',EOL,'.') ;
END ;

                END PUNCH MATRIX OR SYMBOL TABLE

                PUNCH HASH TABLE

PROCEDURE PRCHALF(INT) ;
CONST     CHAR1='H'; CHAR2='S'; CHAR3='H';
VAR       HASHCT : INTEGER ;
          CARDCT : INTEGER ;
          RECCT : INTEGER ;
          RECSONCARD : INTEGER ;
          BLANKCT : INTEGER ;
```

```
PROCEDURE PNCHREC ;
BEGIN
 *PUNCH THE ALFARRAY PART OF THE RECORD*
 FOR INDEX := 1 TO MAXA DO
    BEGIN
       PNCHALF (TTAB[HASHCT].TERM[INDEX]) ;
       VALPAR* := ' ' ;
       PUT(VALPAR) ;
    END ;
 *PUNCH THE INTEGER PART*
 WRITEINTEGER(TTAB[HASHCT].POSN,10,3,VALPAR) ;
END ;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

BEGIN
 CARDCT := 0 ;
 RECCT := 0 ;
 RECSONCARD := 4-MAXA ;
 *SET UP BLANKCT FOR CARDEND*
 IF MAXA EQ 1
   THEN
      BLANKCT := 15
   ELSE
      IF MAXA EQ 2
        THEN
           BLANKCT := 6
        ELSE
           BLANKCT := 23 ;
 FOR HASHCT := 1 TO HSHT-1 DO
    BEGIN
       RECCT := RECCT + 1 ;
       IF RECCT EQ 1
         THEN
           *START OF A NEW CARD. PUNCH 6 BLANKS*
           PNCHBLANKS(6) ;
       PNCHREC ;
       *NOTE THAT LOOP LIMIT IS MAXT-1*
       *THEREFORE WE ALWAYS NEED A ,*
       VALPAR* := ',' ;
       PUT(VALPAR) ;
       IF RECCT EQ RECSONCARD
         THEN
           *END OF CARD*
           BEGIN
              CARDEND(BLANKCT,CARDCT,CHAR1,CHAR2,CHAR3) ;
              RECCT := 0 ;
           END ;
    END ;
 *NOW PUNCH LAST RECORD*
 IF RECCT EQ 0
   THEN
      *START A NEW CARD*
      PNCHBLANKS(6) ;
 RECCT := RECCT + 1 ;
 HASHCT := HSHT ;
 PNCHREC ;
 *FINISH OFF THE LAST CARD*
 *RESET BLANKCT*
 IF MAXA EQ 1
   THEN
```

```
        BLANKCT := 67-RECCT*17
    ELSE
        IF MAXA EQ 2
          THEN
              BLANKCT := 67-RECCT*30
          ELSE
              BLANKCT := 24 ;
  CARDEND(BLANKCT,CARDCT,CHAR1,CHAR2,CHAR3) ;
  WRITE(=PUNCH HASH TABLE =,CARDCT:4,= CARDS=,EOL,EOE) ;
 END;

*                      END PUNCH HASH TABLE                                  *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                      WRITE GRAPHS                                          *

 PROCEDURE WRITEGRAPHS ;
 BEGIN
  *WRITE NODES*
  INDEX := 0 ;
  REPEAT
      INDEX := INDEX + 1 ;
      NODEFLE* := NODEPT[INDEX]* ;
      PUT(NODEFLE) ;
  UNTIL INDEX EQ NODECT ;
  *WRITE THE ROOT RECORD*
  ROOTFLE* := PROOT ;
  PUT(ROOTFLE) ;
  *FIRST OF ALL DELIMIT THE SYNTAX GRAPH KEY*
  *THIS IS NEEDED IN THE PARSER*
  IF NONONTERM LT MAXSP
    THEN
        PSEDEF[NONONTERM+1] := NIL ;
  *NOW WRITE THE KEY*
  SEDEFLE* := PSEDEF ;
  PUT(SEDEFLE) ;
  WRITE(=GRAPHS WRITTEN=,EOL,EOE) ;
 END ;

*                      END WRITE GRAPHS                                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                      MAIN PROGRAM STATEMENT PART.                           *

 BEGIN
  IF MAXA GT 3
    THEN
        BEGIN
           WRITE(=ABORT: MAXA MUST BE 1,2 OR 3=,EOL) ;
           GOTO 2[illegible] ;
        END ;
  IF MAXC NE 10*MAXA
    THEN
        BEGIN
           WRITE(=ABORT. MAXC NE MAXA*10=,EOL) ;
           GOTO 2[illegible] ;
        END ;
```

```
IF MAXPREC*10 LT MAXSY1
  THEN
      BEGIN
         WRITE(E,'ABORT. MAXPREC*10 LT MAXSY1',EOL) ;
         GOTO 2[illegible] ;
      END ;
CONTROL ;
INGRAMMAR ;
IF ERRSW EQ 3
  THEN
      BEGIN
         WRITE(E,'ABORT. HASH TABLE TOO SMALL',EOL) ;
         GOTO 2[illegible] ;
      END ;
IF ERRSW EQ 6
  THEN
      BEGIN
         WRITE(E,'ABORT. SYMTAB TOO SMALL',EOL) ;
         GOTO 2[illegible] ;
      END ;
IF ERRSW EQ 7
  THEN
      BEGIN
         WRITE(E,'ABORT. MAXSE TOO SMALL',EOL) ;
         GOTO 2[illegible] ;
      END ;
IF ERRSW EQ 8
  THEN
      BEGIN
         WRITE(E,'ABORT. NOT ENOUGH ROOM FOR NODESE',EOL) ;
         GOTO 2[illegible] ;
      END ;
IF CONCHRS(1) NE ENE
  THEN
      BEGIN
         PRNTSY(ENON TERM,1,NONONTERM) ;
         PRNTSY([illegible]TERMINALS,NONONTERM+1,NOSYMS) ;
      END ;
IF ERRSW EQ 1
  THEN
      GOTO 2[illegible] ;
IF CONCHRS(2) NE ENE
  THEN
      PRINTGRAM ;
IF ERRSW EQ 2
  THEN
      GOTO 2[illegible] ;
CRPRMAT ;
IF CONCHRS(3) NE ENE
  THEN
      BEGIN
         PRNTLPSET(RIGHTE) ;
         PRNTLPSET(LEFTE) ;
      END ;
IF CONCHRS(4) NE ENE
  THEN
      PRNTMAT ;
DEVGRAPH ;
IF ERRSW EQ 8
  THEN
      BEGIN
```

```
            WRITE(=1ABORT. NOT ENOUGH ROOM FOR NODES=,EOL) ;
            GOTO 2[[ ;
        END ;
    IF CONCHRS[5] NE =N=
      THEN
        PRNTREVGR ;
    IF ERRSW EQ 4
      THEN
        GOTO 2[[ ;
    IF ERRSW EQ 5
      THEN
        GOTO 2[[ ;
    WRITE(=1=) ;
    IF CONCHRS[6] NE =N=
      THEN
        WRITEGRAPHS ;
    IF CONCHRS[7] NE =N=
      THEN
        PNCHALAYINT ;
    IF CONCHRS[8] NE =N=
      THEN
        PNCHSYMPREC(NOSYMS+1,MAXPREC,=P=,=R=,=M=,=MATRIX=) ;
    IF CONCHRS[9] NE =N=
      THEN
        PNCHSYMPREC(NOSYMS,MAXA,=S=,=Y=,=M=,=SYNTAX=) ;
    WRITE(=*** CONSTRUCTOR RUN O.K. ***=,EOL) ;
2[[ ;
END.
```

```
YYYYYYYYY
PROGRAM      ::=<START> BLOCK                    {PRODUCTION NUMBER 1} ;
BLOCK             ::= BLOCKBODY <END>            {PRODUCTION NO 2}
                      BLOCKBODY STATEMENT <END> {PRO03} ;


BLOCKBODY ::= BLOCKHEAD <;>                      {PRO04}
                          /BLOCKBODY LABELDEF    {PRO05}/
              BLOCKBODY  STATEMENT <;>           {PRO06}/
                                    BLOCKBODY <;> {PRO07} ;
BLOCKHEAD         ::= <BEGIN>                    {PRO08}        /BLOCKHEAD <;> {PRO09}
              BLOCKHEAD DECLARE <;>              {PRO10} ;
DECLARE           ::=SIMPLEDEC                   {PRO11}
                                                 /ARRAYDEC     {PRO12} ;
SIMPLEDEC ::=<INTEGER> <IDENTIFIER>              {PRO013}
   /SIMPLEDEC AAAAA <IDENTIFIER>                 {PRO014} ;
ARRAYDEC          ::=BOUNDSLIST EXPRESSION <,> EXPRESSION <)> {PRO015} ;
BOUNDSLIST ::= ARRAYHEAD <(>                     {PRO016}/
                  BOUNDSLIST EXPRESSION <,> EXPRESSION <,> {PRO017} ;
ARRAYHEAD ::=<INTEGER> <ARRAY> <IDENTIFIER>      {PRO018}/
   ARRAYHEAD AARA, <IDENTIFIER>                  {PRO019} ;
LABELDEF ::=<IDENTIFIER> <:>                     {PRO020} ;
STATEMENT ::= STATEMENT*                         {PRO021} ;
STATEMENT* ::= SIMPLESTAT                        {PRO022}/
      IFTHENCL STATEMENT*                        {PRO023}/
          IFTHENCL ELSECLAUSE STATEMENT*         {PRO024} ;
IFTHENCL ::=<IF> EXPRESSION <THEN>               {PRO025} ;
SIMPLEVAR ::=ARRAYID                             {PRO026} ;
ELSECLAUSE ::=SIMPLESTAT <ELSE>                  {PRO027} ;
SIMPLESTAT ::= <GO> <TO> <IDENTIFIER>            {PRO028}
                                                 /BLOCK    {PRO029} ;
SIMPLESTAT ::= VARIABLE <:=> EXPRESSION          {PRO030}
               PROCHEAD EXPRESSION <)>           {PRO031}
                                                 /PROCNOPAR {PRO032} ;
PROCHEAD ::= PROCID <(>                          {PRO033} ;
PROCHEAD ::= PROCHEAD EXPRESSION <,>             {PRO034} ;
PROCID ::=<READ>                                 {PRO035}
```

```
                    /<WRITE> [PROD36] ;
PROCNOPAR ::= <STOP> [PROD37] ;
EXPRESSION ::= EXPR* [PROD38]
                    /EXPR* RELATIONOP EXPR* [PROD39] ;
EXPR* ::= EXPR** [PROD40] ;
EXPR** ::=TERM  [PROD41]
        /<+> TERM   [PROD42]
      /<-> TERM   [PROD43]
     /EXPR** <+> TERM    [PROD44]
     /EXPR** <-> TERM    [PROD46]
     /EXPR** <v> TERM [PROD47] ;
TERM ::=TERM* [PROD48] ;
TERM* ::=FACTOR [PROD49]
                    /TERM* <*> FACTOR    [PROD50]
      /TERM* </> FACTOR     [PROD51]
     /TERM* <^> FACTOR [PROD52] ;
FACTOR ::=SECONDARY [PROD53]
                    /<-> FACTOR [PROD54] ;
SECONDARY ::=PRIMARY [PROD55]
                    /SECONDARY <**> PRIMARY [PROD56] ;
PRIMARY ::=<NUMBER> [PROD57]
                    /VARIABLE [PROD45]
                              /LEFTPAREN EXPRESSION <)> [PROD58] ;
LEFTPAREN ::=<(> [PROD59]   ;
VARIABLE ::=SIMPLEVAR [PROD60]
                    /ARRAYNAME EXPRESSION <)> [PROD61] ;
ARRAYNAME ::=ARRAYID <(> [PROD62]
                    /ARRAYNAME EXPRESSION <,> [PROD63] ;
ARRAYID ::= <IDENTIFIER> [PROD64] ;
      RELATIONOP ::= <<> [PROD65]

                         /<=> [PROD65]
                              /<=> [PROD67]
                                   /<#> [PROD68]
                                        /<>> [PROD69]
                                             /<>> [PROD70] ;
AAAAO ::= <,> [DUMMY1] ;
```

VALIDATION OF GRAMMAR

NO DATA ERRORS

•ALL NON TERMINALS DEFINED

NUMBER OF NON TERMINALS	33
NUMBER OF TERMINALS	37
TOTAL NO OF SYMBOLS	7

```
NON TERM

      1 PROGRAM          2 BLOCK
      3 BLOCKBODY        4 STATEMENT
      5 BLOCKHEAD        6 LABELDEF
      7 DECLARE          8 SIMPLEDEC
      9 ARRAYDEC        10 AAAAA
     11 BOUNDSLIST      12 EXPRESSION
     13 ARRAYHEAD       14 STATEMENT*
     15 SIMPLESTAT      16 IFTHENCL
     17 ELSECLAUSE      18 SIMPLEVAR
     19 ARRAYID          2 VARIABLE
     21 PROCHEAD        22 PROCPARM
     23 PROCID          24 EXPR*
     25 EXPR**          26 RELATIONOP
     27 TERM            28 TERM*
     29 FACTOR          30 SECONDARY
     31 PRIMARY         32 LEFTPAREN
     33 ARRAYNAME

END OF PRINT
```

TERMINALS

34 <>>	35 <?>
36 <#>	37 <=>
38 <$>	39 <<>
40 <NUMBER>	41 <**>
42 <^>	43 <A>
44 </>	45 <*>
46 <v>	47 <->
48 <+>	49 <STOP>
50 <WRITE>	51 <READ>
52 <:=>	53 <TO>
54 <GO>	55 <ELSE>
56 <THEN>	57 <IF>
58 <:>	59 <ARRAY>
60 <,>	61 <(>
62 <)>	63 <..>
64 <IDENTIFIER>	65 <INTEGER>
66 <BEGIN>	67 <;>
68 <.>	69 <END>
70 <START>	

END OF PRINT

GRAMMAR INPUT

PROGRAM	::=	<START> BLOCK . (PRODUCTION NUMBER 1)
BLOCK	::=	BLOCKBODY <END> (PRODUCTION NO 2) / BLOCKBODY STATEMENT <END> (PRO03)
BLOCKBODY	::=	BLOCKHEAD <;> (PRO04) / BLOCKBODY LABELDEF (PRO 05) / BLOCKBODY STATEMENT <;> (PRO06) / BLOCKBODY <;> (PRO07)
STATEMENT	::=	STATEMENT* (PRO021)
BLOCKHEAD	::=	<BEGIN> (PRO08) / BLOCKHEAD <;> (PRO09) / BLOCKHEAD DECLARE <;> (PRO10)
LABELDEF	::=	<IDENTIFIER> <:> (PRO020)
DECLARE	::=	SIMPLEDEC (PRO011) / ARRAYDEC (PRO012)
SIMPLEDEC	::=	<INTEGER> <IDENTIFIER> (PRO013) / SIMPLEDEC AAAA0 <IDENTIFIER> (PRO014)
ARRAYDEC	::=	BOUNDSLIST EXPRESSION <:> EXPRESSION <)> (PRO015)
AAAA0	::=	<,> (DUMMY1)
BOUNDSLIST	::=	ARRAYHEAD <(> (PRO016) / BOUNDSLIST EXPRESSION <:> EXPRESSION <,> (PRO017)

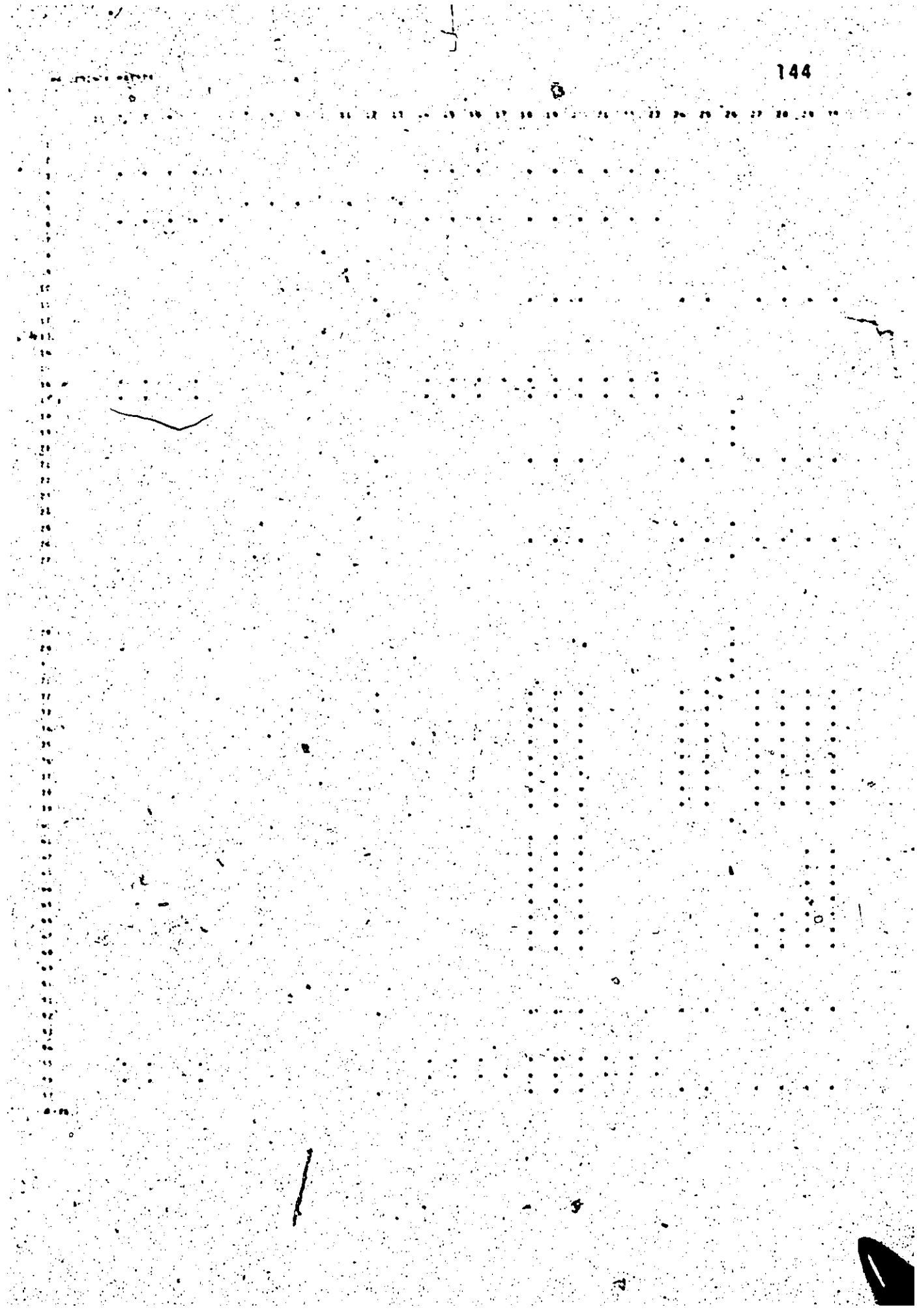
EXPRESSION	::=	EXPR* [PROD38] / EXPR** RELATIONOP EXPR* [PROD34]
ARRAYHEAD	::=	<INTEGER> <ARRAY> <IDENTIFIER> [PROD18] / ARRAYHEAD AAAAC <IDENTIFIER> [PROD19]
STATEMENT*	::=	SIMPLESTAT [PROD22] / IFTHENCL STATEMENT* [PROD23] / IFTHENCL ELSECLAUSE STATEMENT* [PROD24]
SIMPLESTAT	::=	<GO> <TO> <IDENTIFIER> [PROD28] / BLOCK [PROD29] / VARIABLE <:=> EXPRESSION [PROD30] / PROCHEAD EXPRESSION <)> [PROD31] / PROCNOPAR [PROD32]
IFTHENCL	::=	<IF> EXPRESSION <THEN> [PROD25]
ELSECLAUSE	::=	SIMPLESTAT <ELSE> [PROD27]
SIMPLEVAR	::=	ARRAYID [PROD26]
ARRAYID	::=	<IDENTIFIER> [PROD64]
VARIABLE	::=	SIMPLEVAR [PROD60] / ARRAYNAME EXPRESSION <)> [PROD61]
PROCHEAD	::=	PROCID <(> [PROD33] / PROCHEAD EXPRESSION <,> [PROD34]
PROCNOPAR	::=	<STOP> [PROD37]
PROCID	::=	<READ> [PROD35] / <WRITE> [PROD36]

```
EXPR*          ::=   EXPR** [PROD40]

EXPR**         ::=   TERM [PROD41] / <+> TERM [PROD42] / <-> TERM
                     [PROD43] / EXPR** <+> TERM [PROD44] / EXPR**
                     <-> TERM [PROD46] / EXPR** <v> TERM [PROD47]

RELATIONOP     ::=   <<> [PROD65] / <$> [PROD66] / <=> [PROD67] /
                     <#> [PROD68] / <$> [PROD69] / <>> [PROD70]

TERM           ::=   TERM* [PROD48]

TERM*          ::=   FACTOR [PROD49] / TERM* <*> FACTOR [PROD50] /
                     TERM* </> FACTOR [PROD51] / TERM* <^> FACTOR
                     [PROD52]

FACTOR         ::=   SECONDARY [PROD53] / <-> FACTOR [PROD54]

SECONDARY      ::=   PRIMARY [PROD55] / SECONDARY <**> PRIMARY [PROD
                     56]

PRIMARY        ::=   <NUMBER> [PROD57] / VARIABLE [PROD45] /
                     LEFTPAREN EXPRESSION <)> [PROD58]

LEFTPAREN      ::=   <(> [PROD59]

ARRAYNAME      ::=   ARRAYID <(> [PROD62] / ARRAYNAME EXPRESSION <,>
                     [PROD63]
```

END OF PRINTOUT

REPORT ON PREC CONFLICTS

NO PREC CONFLICTS



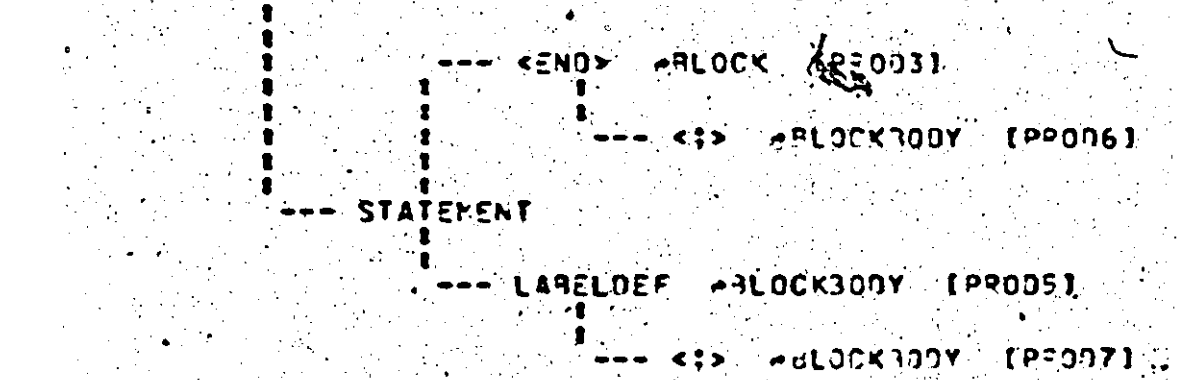
REPORT ON INVERTIBILITY

GRAPHAP UNIQUELY INVERTIBLE

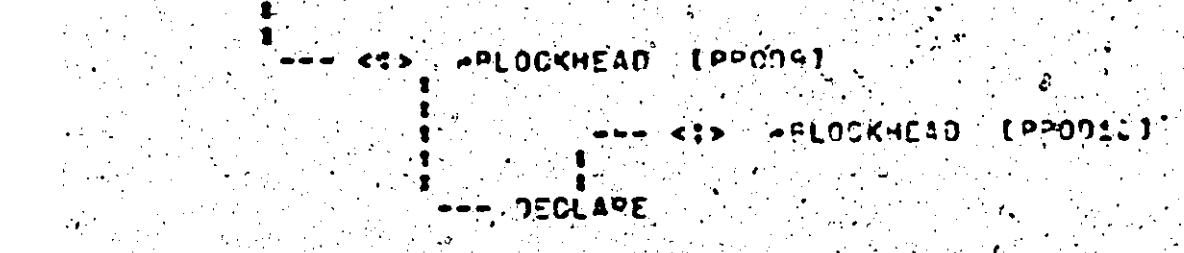
PRINT OF REVERSE GRAPH

BLOCK CONSTITUTES A RHS, ^SIMPLESTAT [PROD29]

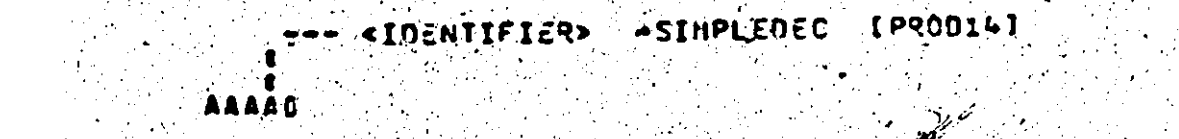
ROOT OF FOLLOWING TREE, BLOCKBODY



ROOT OF FOLLOWING TREE BLOCKHEAD

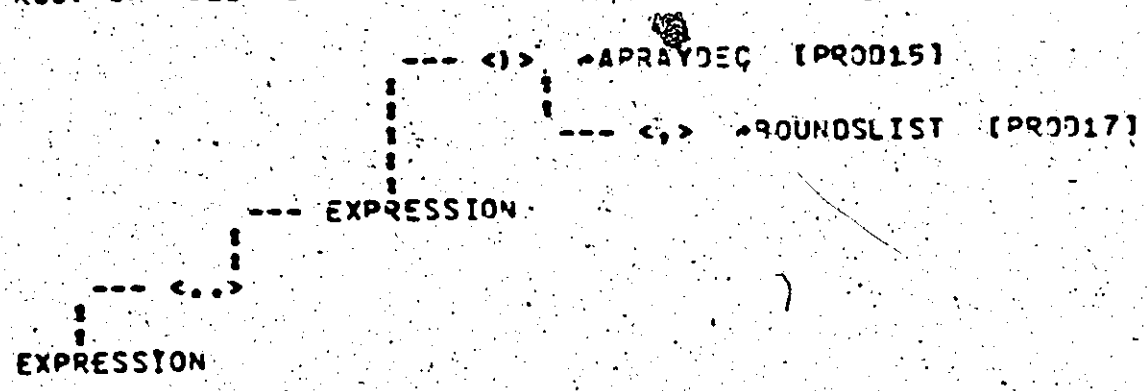


ROOT OF FOLLOWING TREE SIMPLEDEC
THIS SYMBOL CONSTITUTES A RHS, ^DECLARE [PROD11]

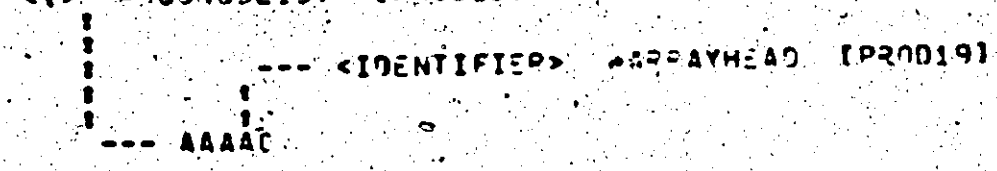


ARRAYDEC CONSTITUTES A RHS, *DECLARE [PROD12]

ROOT OF FOLLOWING TREE BOUNDSLIST



ROOT OF FOLLOWING TREE ARRAYHEAD



STATEMENT* CONSTITUTES A RHS, *STATEMENT [PROD21]

ROOT OF FOLLOWING TREE SIMPLESTAT
THIS SYMBOL CONSTITUTES A RHS, *STATEMENT* [PROD22]

<ELSE> *ELSECLAUSE [PROD27]

```
ROOT OF FOLLOWING TREE  IFTHENCL

STATEMENT*  ←STATEMENT*  [PROD23]
   :
   :
   :          --- STATEMENT*  ←STATEMENT*  [PROD24]
   :          :
   :          :
    --- ELSECLAUSE


SIMPLEVAR CONSTITUTES A RHS, ←VARIABLE  [PROD65]


ROOT OF FOLLOWING TREE  ARRAYID
THIS SYMBOL CONSTITUTES A RHS, ←SIMPLEVAR  [PROD26]

<(>  ←ARRAYNAME  [PROD62]


ROOT OF FOLLOWING TREE  VARIABLE
THIS SYMBOL CONSTITUTES A RHS, ←PRIMARY  [PROD45]

    --- EXPRESSION  ←SIMPLESTAT  [PROD3 ]
   :
   :
<:=>


ROOT OF FOLLOWING TREE  PROCHEAD

    --- <)>  ←SIMPLESTAT  [PROD31]
   :          :
   :          :
   :           --- <,>  ←PROCHEAD  [PROD34]
   :
   :
EXPRESSION
```

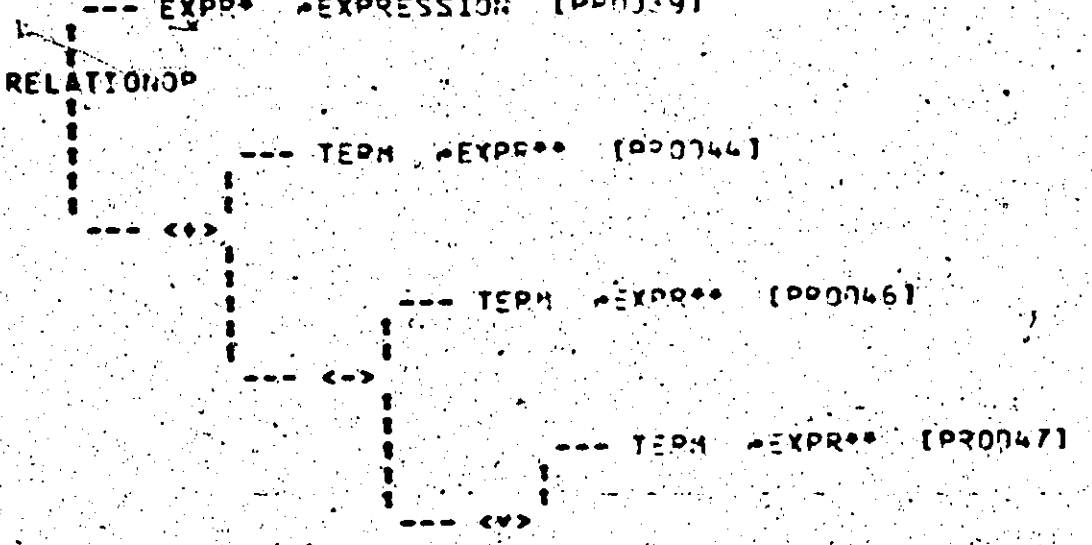
PROCNOPAR CONSTITUTES A RHS, ^SIMPLESTAT \ [PROD32]

ROOT OF FOLLOWING TREE PROCID

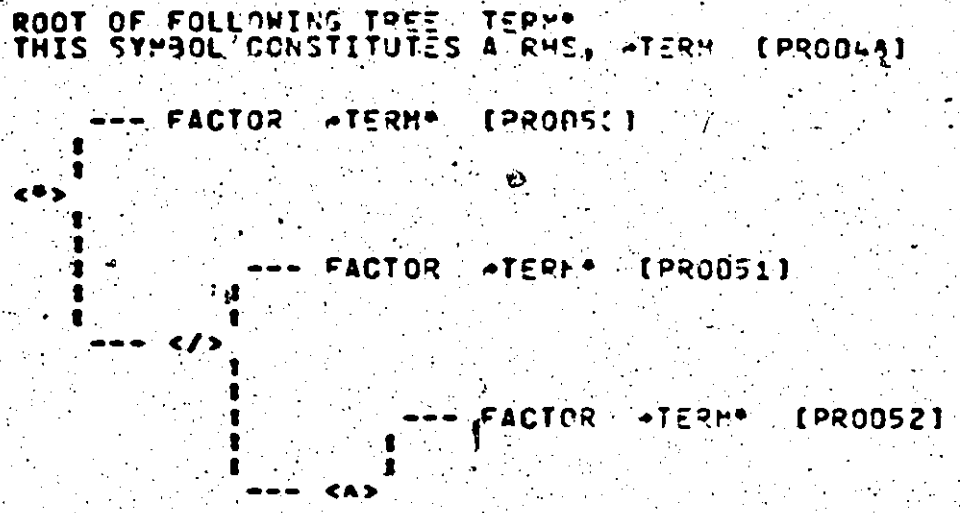
<(> ^PROCHEAD [PROD33]

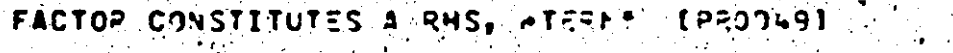
EXPR* CONSTITUTES A RHS, ^EXPRESSION [PROD38]

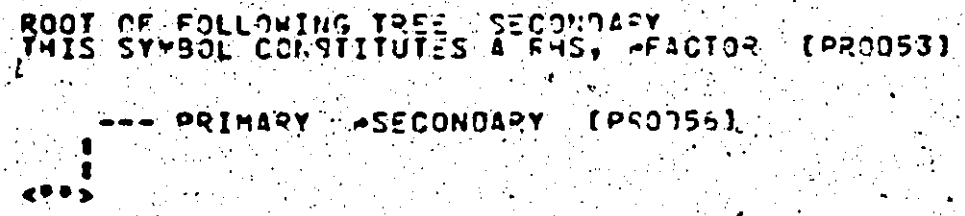
ROOT OF FOLLOWING TREE EXPR**
THIS SYMBOL CONSTITUTES A RHS, ^EXPR* [PROD41]

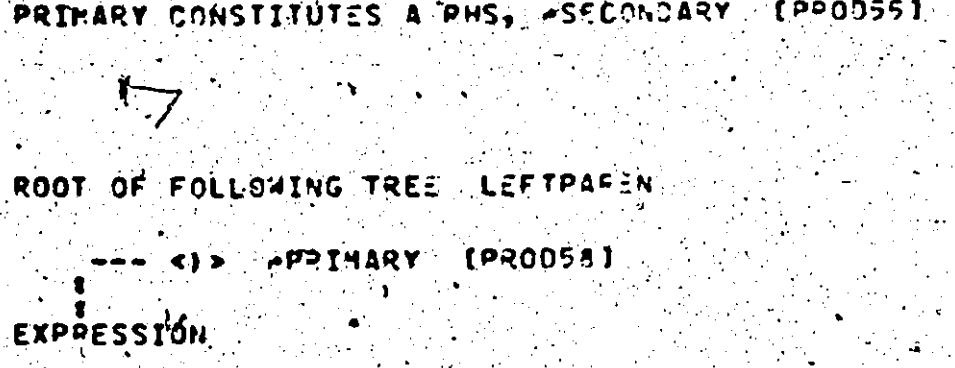


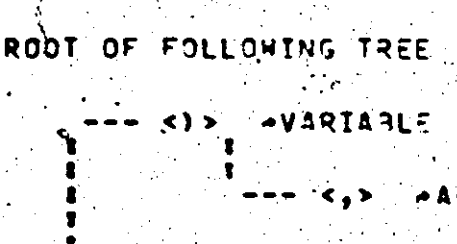
TERM CONSTITUTES A RHS, ^EXPR** [PROD41]


ROOT OF FOLLOWING TREE TERM*
THIS SYMBOL CONSTITUTES A RHS, *TERM [PROD48]
--- FACTOR *TERM* [PROD50]
<*>
--- FACTOR *TERM* [PROD51]
--- </>
--- FACTOR *TERM* [PROD52]
--- <A>


FACTOR CONSTITUTES A RHS, *TERM* [PROD49]


ROOT OF FOLLOWING TREE SECONDARY
THIS SYMBOL CONSTITUTES A RHS, *FACTOR [PROD53]
--- PRIMARY *SECONDARY [PROD56]
<**>


PRIMARY CONSTITUTES A RHS, *SECONDARY [PROD55]
ROOT OF FOLLOWING TREE LEFTPAREN
--- <(> *PRIMARY [PROD58]
EXPRESSION



<)> CONSTITUTES A RHS, ^RELATIONOP [PROD70]

<>> CONSTITUTES A RHS, ^RELATIONOP [PROD69]

<=> CONSTITUTES A RHS, ^RELATIONOP [PROD68]

<#> CONSTITUTES A RHS, ^RELATIONOP [PROD67]

<$> CONSTITUTES A RHS, ^RELATIONOP [PROD66]

<<> CONSTITUTES A RHS, ^RELATIONOP [PROD65]

<NUMBER> CONSTITUTES A RHS, ^PRIMARY [PROD57]

ROOT OF FOLLOWING TREE <^>

FACTOR ^FACTOR [PROD54]

ROOT OF FOLLOWING TREE <->

TERM ^EXPR** [PR0043]

ROOT OF FOLLOWING TREE <+>

TERM ^EXPR** [PR0042]

<STOP> CONSTITUTES A RHS, ^PROCNOPAR [PR0037]

<WRITE> CONSTITUTES A RHS, ^PROCID [PR0036]

<READ> CONSTITUTES A RHS, ^PROCID [PR0035]

ROOT OF FOLLOWING TREE <GO>

--- <IDENTIFIER> ^SIMPLESTAT [PR0028]
|
<TO>

ROOT OF FOLLOWING TREE <IF>

--- <THEN> ^IFTHENCL [PR0025]
|
EXPRESSION

<,> CONSTITUTES A RHS, ^AAAA0 (DUMMY1)

<(> CONSTITUTES A RHS, ^LEFTPAREN (PROD59)

ROOT OF FOLLOWING TREE <IDENTIFIER>
THIS SYMBOL CONSTITUTES A RHS, ^ARRAYID (PROD64)

<8> ^LABELDEF (PROD20)

ROOT OF FOLLOWING TREE <INTEGER>

<IDENTIFIER> ^SIMPLEDEC (PROD13)

<IDENTIFIER> ^ARRAYHEAD (PROD18)

<ARRAY>

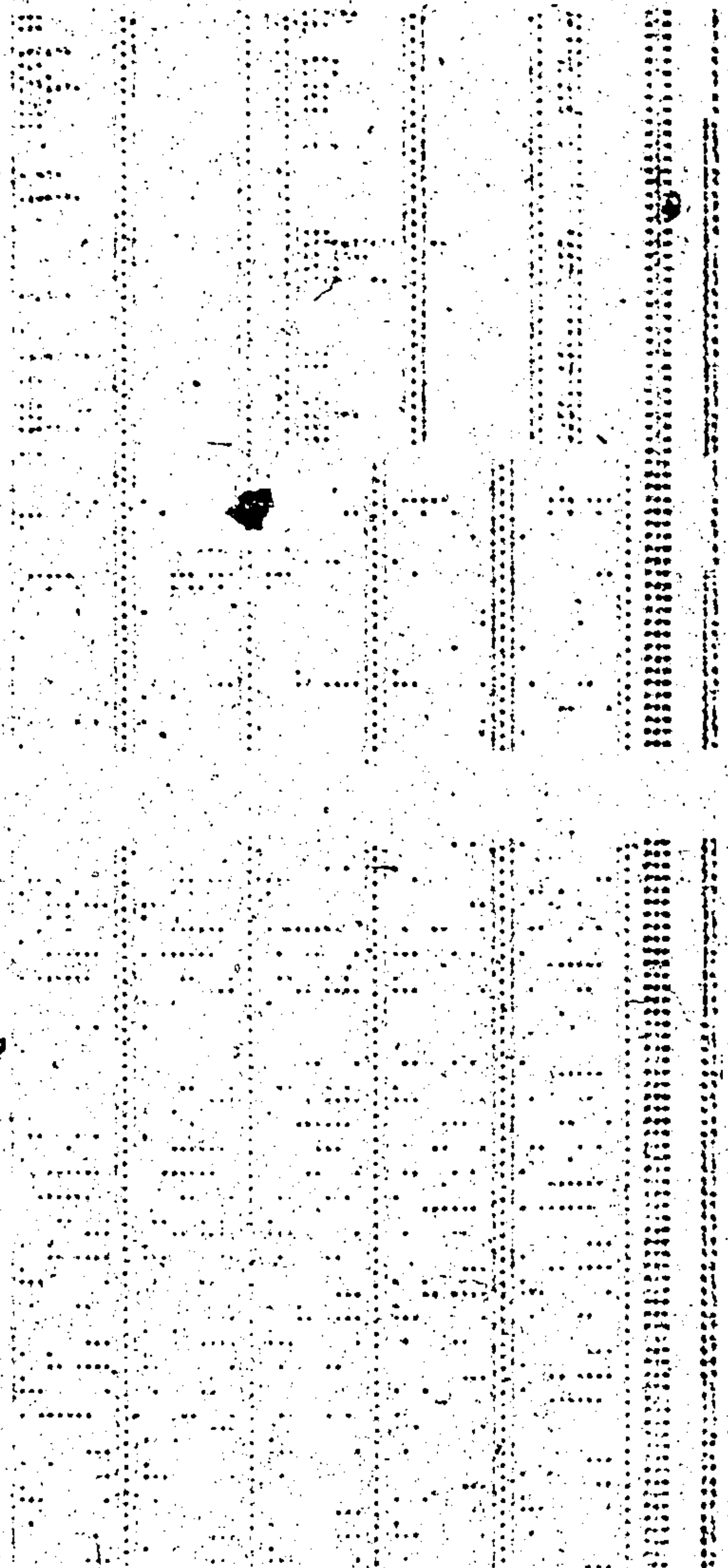
<BEGIN> CONSTITUTES A RHS, ^BLOCKHEAD (PROD9)

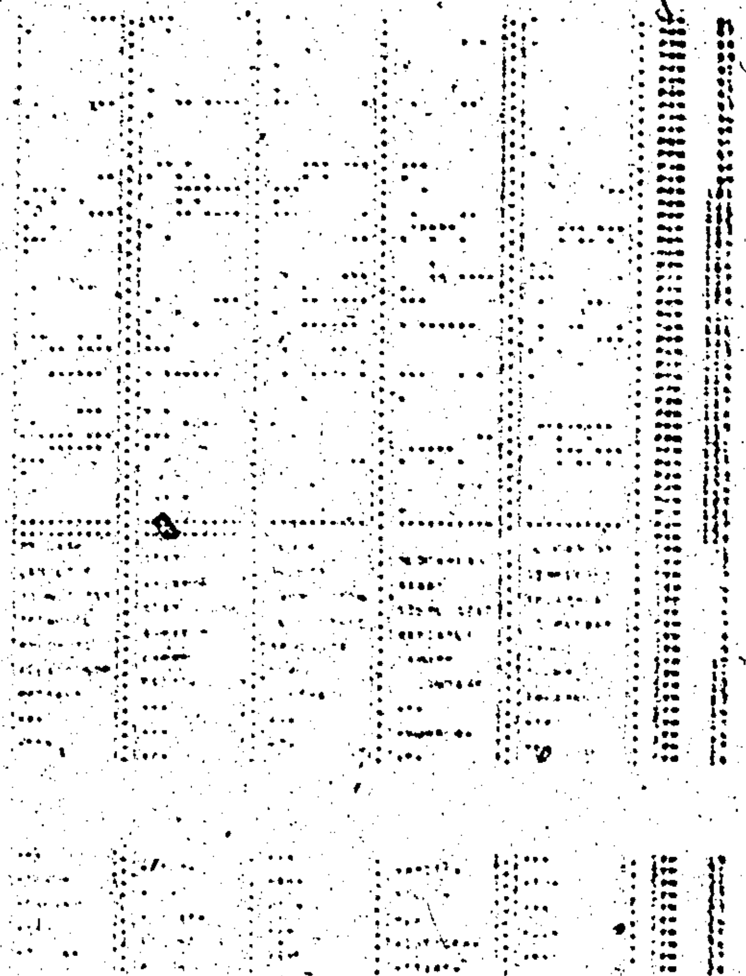
ROOT OF FOLLOWING TREE <START>

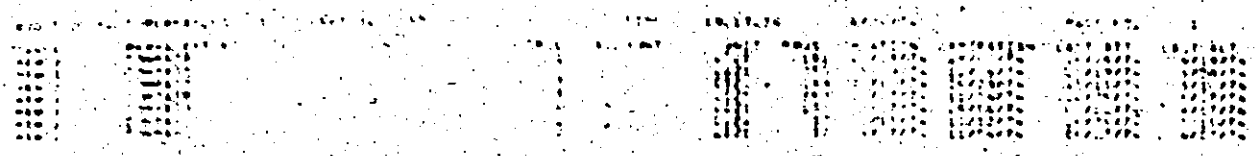
BLOCK ^PROGRAM (PRODUCTION NUMBER 1)

END OF PRINT OF REVERSE GRAPH

```
GRAPHS WRITTEN
PUNCH HASH TABLE    37 CARDS
PUNCH MATRIX       128 CARDS.
PUNCH SYNTAX        28 CARDS
*** CONSTRUCTOR RUN O.K. ***
```



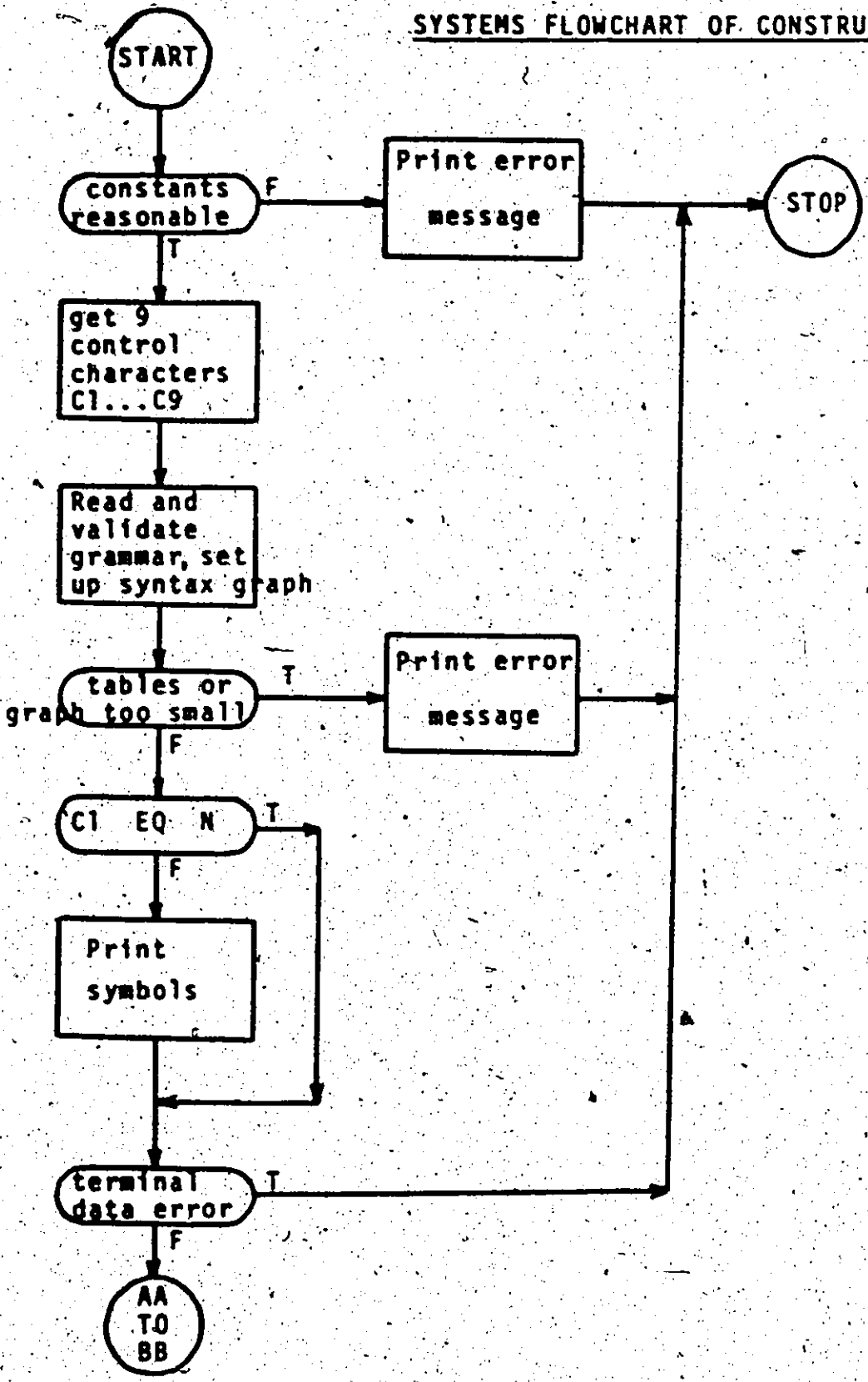


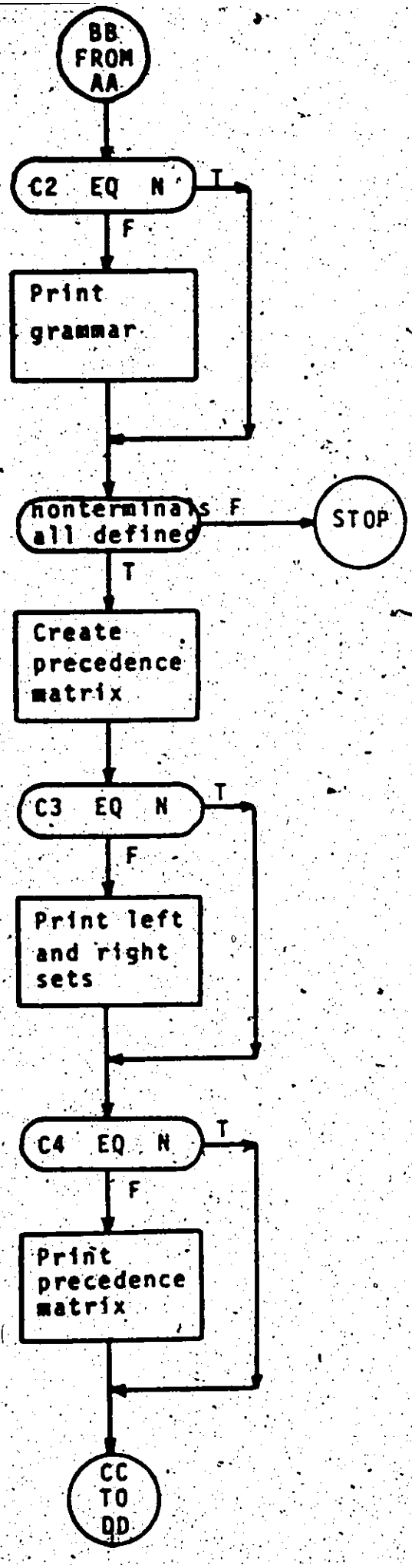


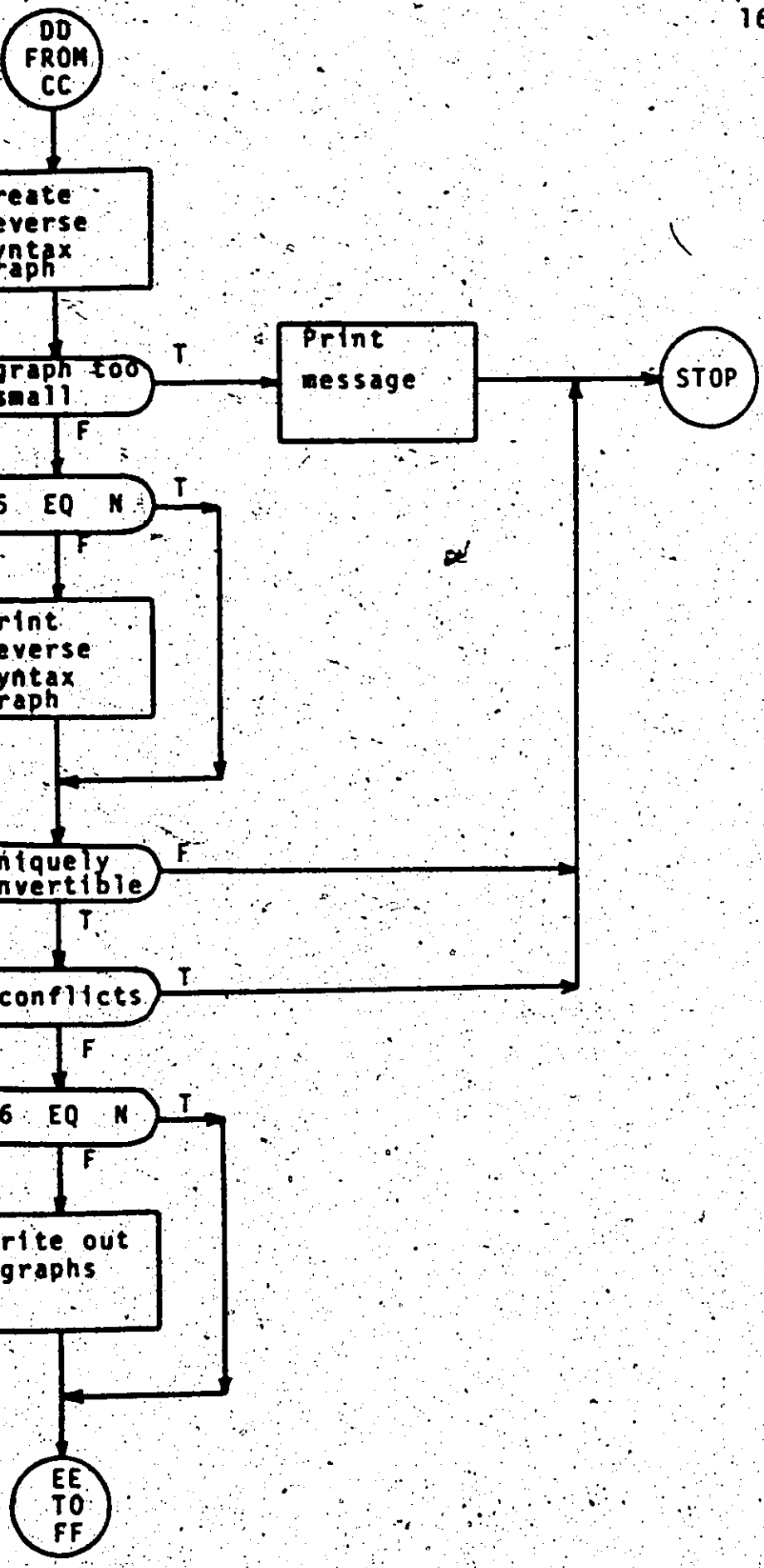
```
 06/26/74 MCMASTER - SCOPE 3.4 AT L348   03/22/74
19.12.44.
19.12.44.HAWCJIJ                              HAWCJIJ
19.12.44.IP  00.00576 WORDS - FILE INPUT  , DC 00
19.12.44.HAWCJ,CM55000,T40.         CONSTRUCTOR
19.12.44.                                JOHNS,C.R.
19.12.44.DISPOSE(OUTPUT,*P2)
19.12.44.EOJ DISPOSITION SPECIFIED, KY IGNORED
19.12.44.ATTACH(FULLCON,COMPAID,ID=HAWCJ,CY=5)
19.12.44.ATTACH(PASCAL,ID=GGGPASCAL,MR=1)
19.12.44.PFN IS
19.12.44.PASCAL
19.12.45.PF CYCLE NO. = 020
19.12.45.COPYCR(INPUT,GRAM)
19.12.45.REWIND(GRAM)
19.12.45.PASCAL(O=GRAM,R=CON,LL=4 00,P=FULLCON)
19.12.46.- PASCALSYSTEM VERS. 04/16/73
19.13.24.- END COMPILATION
19.13.31.- END PASCAL
19.13.32.REWIND(GRAM)
19.13.32.REWIND(CON)
19.13.32.COPYSBF(GRAM,OUTPUT)
19.13.32.COPY(CON,OUTPUT)
19.13.33.REWIND(VALPAR)
19.13.33.COPYSBF(VALPAR,OUTPUT)
19.13.33.REWIND(VALPAR)
19.13.33.COPYCF(VALPAR,PUNCH)
19.13.34.EOF/EOI ENCOUNTERED
19.13.34.AUDIT(AI=P,ID=HAWCJ)
19.13.35.  EXIT
19.13.35.REWIND(SEDEFLE)
19.13.35.REWIND(POSIFLE)
19.13.36.REWIND(NODEFLE)
19.13.36.REQUEST(CONPAR,*PF)
19.13.36.COPYBR(SEDEFLE,CONPAR)
19.13.36.( CONPAR  ASSIGNED TO EST 22 )
19.13.36.COPYBR(POSIFLE,CONPAR)
19.13.36.COPYBR(NODEFLE,CONPAR)
19.13.36.CATALOG(CONPAR,ID=HAWCJ,PW=*---*,PP=15 ,
19.13.36.CY=1)
19.13.37.NEWCYCLE CATALOG
19.13.37.CY NO. ALREADY IN USE
19.13.37.CT ID=     HAWGJ PFN=CONPAR
19.13.37.CT CY= 013   00023 WORDS.
19.13.37.AUDIT(AI=P,ID=HAWCJ)
19.13.39.  EXIT
19.13.39.OP  00027954 WORDS - FILE OUTPUT , DC 42
19.13.39.OP  00031456 WORDS - FILE PUNCH  , DC 10
19.13.39.CP       326.837 SEC.
19.13.39.PP       338.561 SEC.
19.13.39.IO       212.313 SEC.
19.13.39.IOM      576.940 WS/512
19.13.39.CP      1298.620 WS/512
19.13.39.ENDJOB 06/26/74

***********          HAWCJIJ  //// END OF LIST ////
***********          HAWCJIJ  //// END OF LIST ////
```

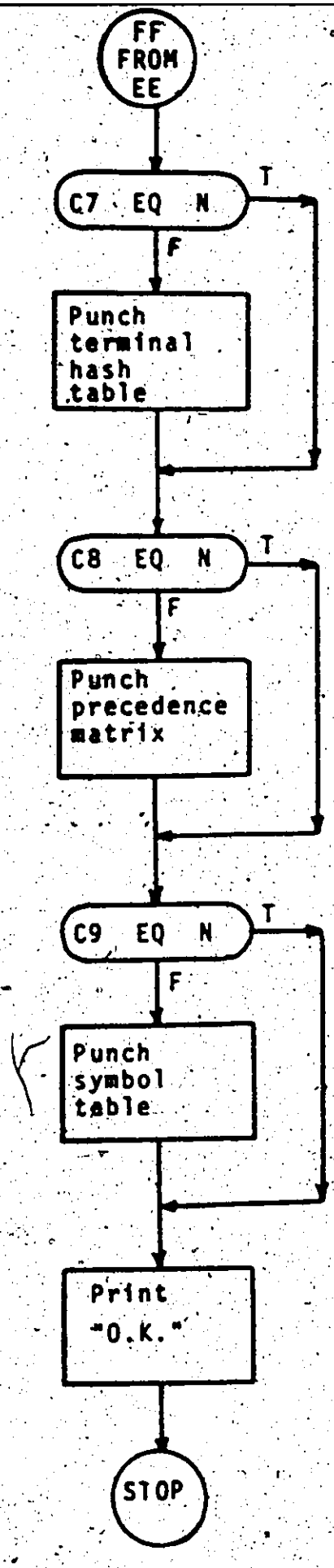
APPENDIX B

SYSTEMS FLOWCHART OF CONSTRUCTOR




BB FROM AA
C2 EQ N
T
F
Print grammar
nonterminals all defined
F
STOP
T
Create precedence matrix
C3 EQ N
T
F
Print left and right sets
C4 EQ N
T
F
Print precedence matrix
CC TO DD


DD FROM CC
Create reverse syntax graph
graph too small
T
Print message
STOP
F
C5 EQ N
T
F
Print reverse syntax graph
uniquely invertible
F
T
conflicts
T
F
C6 EQ N
T
F
Write out graphs
EE TO FF


FF
FROM
EE
C7 EQ N
T
F
Punch
terminal
hash
table
C8 EQ N
T
F
Punch
precedence
matrix
C9 EQ N
T
F
Punch
symbol
table
Print
"O.K."
STOP

APPENDIX C

LEXICOGRAPHIC INPUT ERROR SUMMARY

An error message is at most ten characters.

message	meaning
FOL SYMBOL	A symbol, /, ; or [must follow a symbol
INV FOL]	A / or ; must follow a]
NOT NON TM	A nonterminal symbol must start with a letter or digit.
NOT SYMBOL	A symbol must start with a letter, digit or <.
NOT =OR: NOT = NOT →OR:	Either → or := or ::= is allowed
SEM [REQD	Semantics are continued to next card by terminating present card with a [
< BUT NO >	Terminal not delimited properly. Terminal must be on one card
<> ILLEGAL	The empty rule is not allowed for precedence grammars

Note: Reading past end of file is detected by the operating system. This is caused by not terminating the grammar with a period (.).

APPENDIX D

Parser: i). program listing (ALGSUB)

ii) outputs for small correct sentence and same sentence with one input error

iii) dayfile ("job control")

DOCUMENTATION

****

PARSER FOR SENTENCE OF SIMPLE PRECEDENCE LANGUAGE

****

AUTHOR C.R.JOHNS
DEPARTMENT OF APPLIED MATH
MCMASTER UNIVERSITY

LANGUAGE PASCAL (DEC 1972 VERSION)
COMPUTER CDC 64
DATE JUNE 1974

ERROR DETECTION AND RECOVERY BASED ON :-

LEINIUS,R.P.,1970. ERROR DETECTION AND RECOVERY FOR SYNTAX DIRECTED COMPILER SYSTEMS
PH.D. THESIS, UNIVERSITY OF WISCONSIN

RHODES,S.P.,1973 PRACTICAL SYNTACTIC ERROR RECOVERY FOR PROGRAMMING LANGUAGES
TECHNICAL REPORT 15, UNIVERSITY OF CALIFORNIA, BERKELEY

SUMMARY OF INPUT/OUTPUT FILES

	NO.	NAME	TYPE	DESCRIPTION
INPUT	1)	SEQ-FILE	BINARY	FILES 1,2, AND 3
	2)	ROOTFILE	BINARY	CONSTITUTE THE SYNTAX GRAPH
	3)	NO.FILE	BINARY	AND REVERSE SYNTAX GRAPH
	4)	SOURCE	CODED	THE PROGRAM(SENTENCE)
OUTPUT	1)	OUTPUT	CODED	OPTIONAL PRINTING TO SHOW PARSE.
	2)	SCRATCH	CODED	LIST OF SEMANTIC ITEMS USED IN PARSE. GIVES A BRIEF TRACE OF PARSE AND RECOVERY

SUMMARY OF ERROR CODES (ABORT CONDITIONS)

CODE NUMBER	MEANING
1	TERMINAL NOT FOUND IN HASH TABLE
2	STACK OVERFLOW

INITIALISATIONS BY VALUE STATEMENT

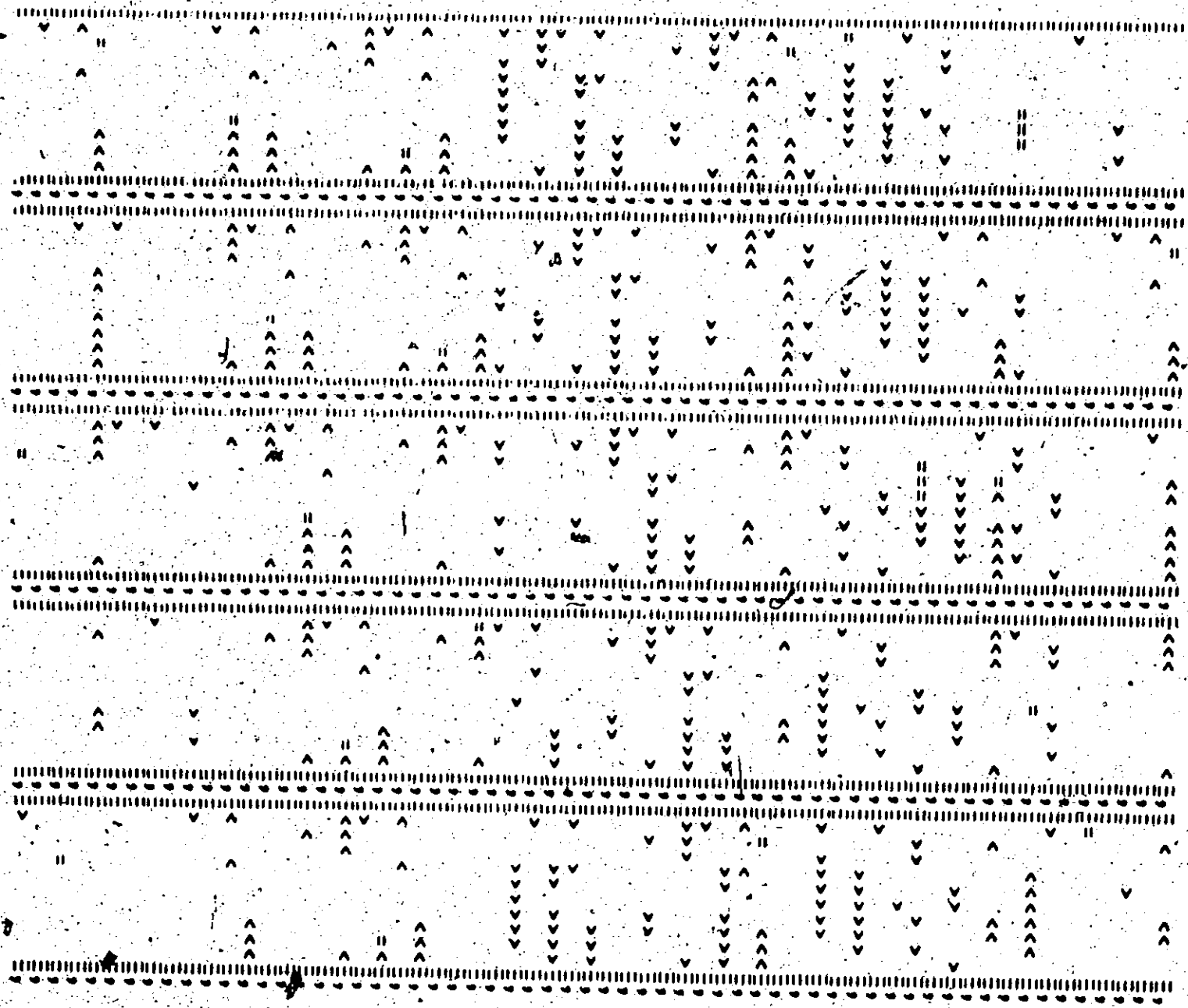
TRANMAT	HAND PUNCHED
COLNO	HAND PUNCHED
SYNCOST	HAND PUNCHED
PRECMAT	PRODUCED BY CONSTRUCTOR

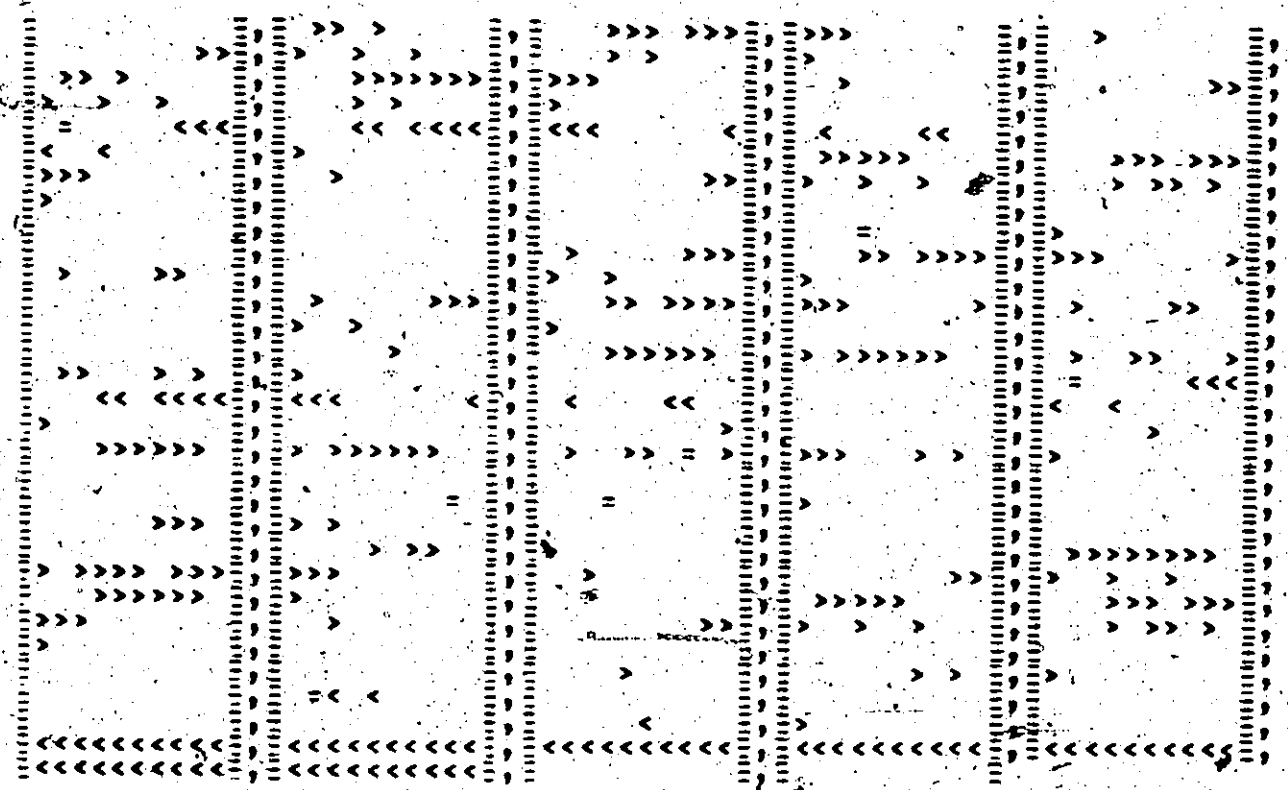
```
SYMTAB          PRODUCED BY CONSTRUCTOR
TTAB            PRODUCED BY CONSTRUCTOR

------------

PROCEDURE AND FUNCTION STRUCTURE

FIRST LEVEL          SECOND LEVEL          THIRD LEVEL

READGRAPHS

PARSE
                     HASHLD
                     INIT
                     LEXALGSUR
                                           OUTCHAR
                                           PUTBACK
                                           PACKTOK
                                           ENDTOK
                                           TOKERR
                                           EOFCHECK
                                           NXTCHAR
                     PUTSYM
                     PUTSYMS
                     PUTREDN
                     STCKSTAT
                     STCKFLOW
                     REDUCE
                     PARSE
                     LSTCKLEFR
                     MOPEL
                     SEMANTICS
                     ERROR
                                           SHUTDOWN
                                           SHUTUP
                                           SYMBOLIN
                                           STROUTSYMIN
                                           LSTCKERR
                                           CONDENSE
                                           BACKMOVE
                                           SPECIALFREC
                                           LOCSYMCORR
                                           COSTLSCPHR
                                           FINDTEST
                                           PUTCANDREP
                                           INSERTBEST

                END DOCUMENTATION

                GLOBALS

CONST
     MAXSE = 7 ;  MAXSY = 8. ;  MAXSY1 = 81 ;  MAXPREC = 9 ;
     MSHT = 73 ;
     MAXSTACK = 1.. ;
     LOOPLIM = 5 ;
     MAXCOST = 9. ;
     MAXA = 2 ;  MAXC = 2. ;
```

```
        ALTERNATIVES FOR MAXA AND MAXC
        MAXA = 1 : MAXC = 1 ;
        MAXA = 3 : MAXC = 30 ;
        END OF ALTERNATIVES
    MAXSE     MAXIMUM NUMBER OF NONTERMINALS
    MAXSY     MAXIMUM NUMBER OF SYMBOLS
    MAXSY1    MAXSY+1
    MAXPREC   MAXSY1 DIVIDED BY 16 AND ROUNDED UP
    MSHT      SIZE OF TERMINAL HASH TABLE
    MAXSTACK  SIZE OF STACK
    LOOPLIM   LOOP LIMIT FOR RECOVERY
    MAXCOST   MAXIMUM COST ALLOWED IN RHODES RECOVERY
    MAXC      MAXIMUM NUMBER OF CHARACTERS IN A SYMBOL
    MAXA      MAXC/2
TYPE
    POINT = ^GRAPH ;
    ALFARRAY = ARRAY[1..MAXA] OF ALFA ;
    CHARARRAY = ARRAY[1..MAXC] OF CHAR ;
    PSEUDETYPE = ARRAY[1..MAXSE] OF POINT ;
    NODEDES = (TSYMBOL,TSEMANT,TSYMBOLR,TSYMBOLLHS) ;
    NODE = RECORD
             CASE TAGFIELD : NODEDES OF
               TSYMBOL    : (SYMBOL  : INTEGER ;
                             PDEF    : POINT ;
                             PALT    : POINT ;
                             PSUCC   : POINT ;
                             PSEA    : POINT ) ;
               TSEMANT    : (SEMANT  : ALFA ;
                             PCONT   : POINT ) ;
               TSYMBOLR   : (SYMBOLR  : INTEGER ;
                             PLHS    : POINT ;
                             PALTP   : POINT ;
                             PSUCCR  : POINT ) ;
               TSYMBOLLHS : (SYMBOLLHS : INTEGER ;
                             PSEA9   : POINT ) ;
             END ;
    PPRODTYPE = ARRAY[1..MAXSY] OF RECORD
                                     PSEUSYM : POINT ;
                                     PLHS    : POINT ;
                                   END ;
    TTABTYPE = ARRAY[1..MSHT] OF RECORD
                                   TERM : ALFARRAY ;
                                   POSN : INTEGER ;
                                 END ;
    SYMCOSTYPE = ARRAY[1..MAXSY] OF RECORD
                                      INSERT  : INTEGER ;
                                      DELETE  : INTEGER ;
                                      REPLACE : INTEGER ;
                                      REPMENT : INTEGER ;
                                    END ;
    NOSTATES = 1..2 ;
VAR
    GRAPH      : CLASS 4,5 OF NODE ;
                 =SYNTAX GRAPH AND REV SYNTAX GRAPH+
    CHAY       : CHARARRAY ;
                 =FOR SYMBOL IN CHARACTERS+
    ALAY       : ALFARRAY ;
                 =FOR SYMBOL AFTER PACKING+
    IDSTATES   : SET OF NOSTATES ;
                 =USED TO DEFINE RESERVED WORDS+
    CHARNOT    : SET OF CHAR ;
                 =CHARS THAT DO NOT APPEAR IN TERMINALS+
    PWORK      : POINT ;
```

```
                            *WORK FIELD*
      INDEX            : INTEGER ;
                            *WORKFIELD*
      SETABCT          : INTEGER ;
                            *WORK FIELD*
      ITABCT           : INTEGER ;
                            *WORK FIELD*
      ERRSW            : INTEGER ;
                            *ERROR SWITCH*
      NONONTERM        : INTEGER ;
                            *COUNTER FOR NON TERMINALS*
      LHS              : INTEGER ;
                            *A NONTERMINAL*
                            *USED TO BE FOR LEFT HAND SIDE*
      TENCHARS         : ARRAY[1..10] OF CHAR ;
                            *WORKARRAY*
      TRANMAT          : ARRAY[1..42,1..28] OF INTEGER ;
                            *TRANSITION MATRIX*
      GOLD             : ARRAY[1..63] OF INTEGER ;
                            *COLUMN NUMBER - USED WITH TRANMAT*
      SYMCOST          : SYMCOSTYPE ;
                            *RHODES SYMBOL COSTS*
      PRECMAT          : ARRAY[1..MAXSY,1..MAXPREC] OF ALFA ;
                            *PRECEDENCE MATRIX*
      SYMTAB           : ARRAY[1..MAXSY] OF ALFARRAY ;
                            *SYMBOL TABLE*
      TTAB             : TTABTYPE ;
                            *TERM HASH TABLE*
      PPROD            : PPRODTYPE ;
                            *ARRAY OF ROOTS FOR REV SYNTAX GRAPH*
      PSEDEF           : PSEDEFTYPE ;
                            *KEY FOR NODES OF GRAPH*
      SEMAN[OUT]       : TEXT ;
      SOURCE[IN]       : TEXT ;
      SEDEFLE[IN]      : FILE OF PSEDEFTYPE ;
      ROOTFLE[IN]      : FILE OF PPRODTYPE ;
      NODEFLE[IN]      : FILE OF NODE ;
VALUE
      TRANMAT = ([illegible],4,[illegible],2,[illegible],2,16,2,17,5*2,
                 [illegible],3,23,32,2,2,37,[illegible],36,[illegible],35,[illegible],39,-2,
                 [illegible]*2,3*-1,  3,27*-1,   14*2,5,5*2,8*-1,
                 14*2,[illegible],5*2,4*-1,   2,7,[illegible]*2,3*-1,
                 [illegible]*2,[illegible],2,3*-1,  4*2,[illegible],5*2,3*-1,
                 5*2,1[illegible],3*2,4*-1,   3*2,[illegible],1,[illegible],3*-1,
                 [illegible],8*2,6*-1,   9*2,[illegible],3,2,[illegible],5*2,8*-1,
                 1[illegible]*2,[illegible],4*2,3*-1,   4*2,[illegible],15*2,9*-1,
                 3*2,[illegible],5*2,3*-1,   2*2,4[illegible],7*2,8*-1,
                 5*2,[illegible],[illegible],8*2,3*-1,   16*2,19,3*2,3*-1,
                 4*2,2[illegible],15*2,8*-1,   [illegible]*2,2[illegible],1[illegible],[illegible]-1,
                 6*2,22,15*2,8*-1,   14*2,4[illegible],[illegible]*2,8*-1,
                 7*2,2[illegible],[illegible],7*2,4*-1,   4[illegible],5,15*2,4*-1,
                 1[illegible]*2,4[illegible],4*2,3*-1,   [illegible]*2,27,[illegible]*2,8*-1,
                 2,29,13*2,9*-1,   3*2,4[illegible],16*2,3*-1,
                 [illegible]*2,3[illegible],3*2,4*-1,   2,[illegible],1[illegible],[illegible]*2,30,7*2,8*-1,
                 [illegible]*2,[illegible],5*2,4*-1,   14*2,33,5*2,8*-1,
                 3*2,34,[illegible]*2,4*-1,   16*2,35,3*2,8*-1,
                 4*2,[illegible],5*2,3*-1,   22*-1,[illegible],5*-1,
                 2[illegible]*-1,[illegible],4*-1,   24*-2,[illegible],3*-2,
                 27,39,[illegible],2[illegible],2,8*-1,
                 1[illegible]*2,42,5*2,8*-1,   16*2,4[illegible],3*2,8*-1
                 ) ;
```

```
COLNO = (2,3,2[illegible],4,5,6,7,8,9,2[illegible],2[illegible],10,11,12,13,
        14,2[illegible],15,16,17,20,20,18,2[illegible],19,2[illegible],1,1,
        1,1,1,1,1,1,1,1,22,22,23,22,22,22,999,
        24, 25,22,25,999,999,999,21,22, 27,22,
        22 , 22, 28,22,22,22,22,22,22 ) ;
SYMCOST = (3,3,0,0,  3,3,0,0,  8,99,2,5,  3,4,0,0,
          8,99,0,0,  3,3,0,0,  3,3,0,0,  3,3,0,0,
          3,3,0,0,  2,3,0,0,  5,6,0,0,  3,3,2,4,
          5,6,0,0,  2,4,0,0,  3,3,0,0,  3,3,0,0,
          3,3,0,0,  3,3,0,0,  3,5,5,64,  3,3,0,0,
          5,6,0,0,  3,5,5,64,  3,3,0,0,  3,3,0,0,
          3,3,0,0,  3,3,0,0,  3,3,0,0,  3,3,0,0,
          3,3,0,0,  3,3,0,0,  3,3,0,0,  6,5,0,0,
          3,6,0,0,
          3,3,0,0,  3,3,0,0,  3,3,0,0,  3,3,0,0,
          3,3,0,0,  3,3,0,0,  3,5,0,0,  3,3,0,0,
          3,3,0,0,  3,3,0,0,  2,3,0,0,  2,3,0,0,
          3,3,0,0,  2,3,0,0,  2,3,0,0,  3,4,5,64,
          3,4,5,64,  3,4,5,64,  3,4,0,0,  1,5,5,64,
          3,5,5,64,  4,4,5,64,  4,4,5,64,  6,5,5,64,
          2,3,0,0,  3,4,5,64,  2,3,0,0,  6,5,0,0,
          3,3,0,0,  3,5,0,0,  3,5,0,0,  3,5,5,64,
          6,5,5,64,  1,99,0,0,  8,99,0,0,  8,99,0,0,
          8,99,0,0) ;
PRECMAT = (
[illegible]
```





SYNTAX = (

<PROGRAM>
<LABELDEF>
<BOUNDSLIST>
<IFTHENCL>
<DOCHEAD>
<RELATIONOP>
<PRIMARY>
<=>
<**>
<v>
<READ>
<THEN>
<(>
<BEGIN>

<STATEMENT>
<ARRAYDEC>
<STATEMENT*>
<ASSIGN>
<EXPR**>
<FACTOR>
<>>
<<>
</>
<STOP>
<GO>
<ARRAY>
<IDENTIFIER>
<END>

<BLOCK>
<DECLARE>
<EXPRESSION>
<ELSECLAUSE>
<PROCNOPAR>
<TERM>
<LEFTPAREN>
<=>
<^>
<->
<!=>
<IF>
<)>
<&>

<BLOCKHEAD>
<ARRAY>
<SIMPLESTAT>
<VARIABLE>
<EXPR**>
<SECONDARY>
<?>
<NUMBER>
<*>
<WRITE>
<ELSE>
<,>
<INTEGER>
<START>

<BLOCKBODY>
<SIMPLEDEC>
<ARRAYHEAD>
<SIMPLEVAR>
<PROCID>
<TERM*>
<ARRAYNAME>
<$>
<A>
<+>
<TO>
<1>
<..>
<+>

```
TTAB     = (
'<V>       ', 46, '<THEN>    ', 56,
'<Z>       ', 36, '<:>       ', 58,
'          ', [illegible], '          ', [illegible],
'<READ>    ', 51, '          ', [illegible],
'<,>       ', 6[illegible], '<**>      ', 4[illegible],
'<..>      ', 63, '<IF>      ', 57,
'<START>   ', 7[illegible], '<(>       ', 6[illegible],
'<GO>      ', 54, '<*>       ', 45,
'<->       ', 47, '<+>       ', 4[illegible],
'<=>       ', 37, '          ', [illegible],
'          ', [illegible], '          ', [illegible],
'          ', [illegible], '<:=>      ', 52,
'          ', [illegible], '          ', [illegible],
'<END>     ', 69, '          ', [illegible],
'<ARRAY>   ', 59, '          ', [illegible],
'          ', [illegible], '          ', [illegible],
'          ', [illegible], '          ', [illegible],
'          ', [illegible], '<>>       ', 34,
'          ', [illegible], '<IDENTIFIER>', 64,
'          ', [illegible], '<BEGIN>   ', 66,
'          ', [illegible], '<TO>      ', 55,
'          ', [illegible], '<INTEGER> ', 65,
'<ELSE>    ', 55, '          ', [illegible],
'          ', [illegible], '          ', [illegible],
'$         ', 71, '          ', [illegible],
'<NUMBER>  ', 4[illegible], '</>       ', 44,
'          ', [illegible], '<A>       ', 43,
'<<>       ', 38, '          ', [illegible],
'<:>       ', 67, '<^>       ', 4[illegible],
'<>>       ', 35, '<STOP>    ', [illegible],
'<WRITE>   ', 5[illegible], '<<>      ', 39,
'<)>       ', 52  '<;>       ', [illegible]
) ;

(* END GLOBALS *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* READ GRAPHS *)

PROCEDURE READGRAPHS ;
BEGIN
  (*READ IN THE NODES*)
  REPEAT
    (*CREATE A NODE*)
    NEW(PWORK) ;
    (*FILL IT UP FROM NODEFILE*)
    PWORK^ := NODEFLE^ ;
    (*READ A NEW RECORD*)
    GET(NODEFLE) ;
  UNTIL EOF(NODEFLE) ;
  (*READ IN THE REVERSE SYNTAX GRAPH ROOT*)
  PPROD := ROOTFLE^ ;
```

```
  *READ IN THE KEY TO THE SYNTAX GRAPH*
  PSEDEF := SEDEFLE* ;
END ;

                END READ GRAPHS
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                PARSE

PROCEDURE PARSE ;
*ERROR SUMMARY*
*TYPE -   CHARACTER PAIR ERROR*
*TYPE 1   NO RHS FOR POTENTIAL HANDLE* ;
*TYPE 2   HANDLE UNSTACKABLE ON LEFT*
*TYPE 3   HANDLE UNSTACKABLE ON RIGHT*
*ERROR TYPES 1 THROUGH 3 ARE CALLED PHRASE ERRORS*
*IF NO ERRORS ARE FOUND THE REDUCTION IS CARRIED OUT*
*WE ACT ON SEMANTICS AFTER DOING THE REDUCTION*
*IF WE HAVE ERRORS, WE EXIT TO THE PROCEDURE ERROR*
LABEL     999,333 ;
VAR       STATE : INTEGER ;
          CHARIN : CHAR ;
          SYMBOLNO : INTEGER ;
          TABSCH          : INTEGER ;
          IDENTALF : ALFARRAY ;
          NUMALF : ALFARRAY ;
          ENDALF : ALFARRAY ;
          PERPSALF : ALFARRAY ;
          INPCHARS : ARRAY[1..33] OF CHAR ;
          STACK : ARRAY[1..MAXSTACK] OF INTEGER ;
          STIND : INTEGER ;
          REL : CHAR ;
          ERRST : INTEGER ;
          LOOPCT : INTEGER ;
          ENDCT : INTEGER ;
          QUEST : INTEGER ;
          PREVSG : POINT ;
          FINISH : BOOLEAN ;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROCEDURE HASHLU ;
*IN THIS VERSION WE JUST LOCATE THE ELEMENT*
*IF WE CANNOT FIND IT, AN ERROR MESSAGE IS PRINTED, AND WE ABORT*
VAR       QUOT : INTEGER ;
          REM : INTEGER ;
          INUM : INTEGER ;
          FINISH : BOOLEAN ;
BEGIN
 FINISH := FALSE ;
 QUOT := 0 ;
 REM := 0 ;
 *HASH THE ALFA ARRAY ALAY*
 FOR INDEX := 1 TO MAXA DO
     BEGIN
       *CONVERT ANY ALFA TO INTEGER USING ORD*
       INUM := ORD(ALAY[INDEX]) DIV 4,96 ;
       *GET THE ABSOLUTE VALUE*
       INUM := ABS(INUM) ;
```

```
      REM := REM + ( INCR MOD HSHT ) ;
      REM := REM MOD HSHT  ;
      QUOT := QUOT + ( ( INCR DIV HSHT ) MOD HSHT ) ;
      QUOT := QUOT MOD HSHT  ;
    END ;
  AT PRESENT REM IS IN THE RANGE 1 TO  HSHT-1
  REM := REM + 1 ;
  IF QUOT EQ 0
    THEN
      QUOT := 1 ;
  TABSRCH := REM ;
  REPEAT
    IF TTAB[TABSRCH].TERM EQ ALAY
      THEN
        WE HAVE LOCATED THE ELEMENT
        FINISH := TRUE
      ELSE
        IF TTAB[TABSRCH].POSN NE
          THEN
            BEGIN
              WE HAVE A COLLISION, REHASH
              TABSRCH := TABSRCH + QUOT ;
              IF TABSRCH GT HSHT
                THEN
                  TABSRCH := TABSRCH - HSHT  ;
              IF TABSRCH EQ REM
                THEN
                  ERROR. FULL TABLE AND NOT FOUND
                  BEGIN
                    WRITE(E1ABORT. HASH ERROR1E,EOL) ;
                    ERRSW := 1 ;
                    GOTO EXIT 999 ;
                  END ;
            END
          ELSE
            ERROR. NOT FULL TABLE AND NOT FOUND
            BEGIN
              WRITE(E1ABORT. HASH ERROR2E,EOL) ;
              ERRSW := 1 ;
              GOTO EXIT 999 ;
            END ;
  UNTIL FINISH ;
END ;
```

```
PROCEDURE INIT ;
VAR
      NOMORE : BOOLEAN ;
BEGIN
  STATE := 1 ;
  LAST := 1 ;
  FINISH := FALSE ;
  INPCHARS[1] := EOL ;
  FOR INDEX := 2 TO MAXA DO
    BEGIN
      PERSRALF[INDEX] := ' ' ;
      NUMALF[INDEX] := ' ' ;
      IDENTALF[INDEX] := ' ' ;
      ENDALF[INDEX] := ' ' ;
    END ;
```

```
  PERPERALF[1] := '<...>' ;
  NUMALF[1] := '<NUMBER>' ;
  ENDALF[1] := 'EOS' ;
  IDENTALF[1] := '<IDENTIFIER' ;
  IF MAXA NE 1
    THEN
        BEGIN
          INDEX := 2 ;
          IDENTALF[INDEX] := 'R>' ;
        END ;
  *SET UP DELIMITER IN STACK*
  ALAY := ENDALF ;
  HASHL2 ;
  STACK[1] := TTAB[TARSPCH].POSN ;
  STIND := 1 ;
  *CALCULATE THE NO OF NON TERMINALS*
  NOMORE := FALSE ;
  NONONTERM := 1 ;
  REPEAT
     NONONTERM := NONONTERM + 1 ;
     IF NONONTERM EQ MAXSE
       THEN
          NOMORE := TRUE
       ELSE
          IF PSEUDF[NONONTERM+1].EQ NIL
            THEN
              NOMORE := TRUE ;
  UNTIL NOMORE ;
  ALAY[1] := '' ;
  *START A NEW PAGE*
  SEMAN* := '1' ;
  PUT(SEMAN) ;
  SEMAN* := EOL ;
  PUT(SEMAN) ;
  @LOOPCT := 1 ;
  END ;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  PROCEDURE LEXALGOL ;
  LABEL   333,444 ;
  VAR
          PREVSTATE : INTEGER ;
          INPCHARSCT : INTEGER ;

+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +

  PROCEDURE OUTCHAR ;
  BEGIN
   INPCHARSCT := INPCHARSCT + 1 ;
   IF INPCHARSCT LE 3
     THEN
        INPCHARS[INPCHARSCT] := CHAPIN ;
  END ;

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

  PROCEDURE PUTBACK ;
  BEGIN
   *USED TO DELETE LAST CHAR FROM INPCHARS*
   IF INPCHARSCT LT 3
```

```
    THEN
        BEGIN
          INPCHARS[INPCHARSCT] := E_E ;
          INPCHARSCT := INPCHARSCT - 1 ;
        END ;
  END ;

  PROCEDURE PACKTOK ;
  VAR       INDEXA : INTEGER ;
            INDEXC : INTEGER ;
  BEGIN
   *IF ROOM INSERT >*
   IF INPCHARSCT LT 2 *MAXA
     THEN
        INPCHARS[INPCHARSCT+1] := E>E ;
   INDEXC := 1 ;
   FOR INDEXA := 1 TO MAXA DO
      BEGIN
        PACK(INPCHARS,INDEXC,ALAY[INDEXA]) ;
        INDEXC := INDEXC + 1 ;
      END ;
  END ;

  PROCEDURE ENDTOK ;
  BEGIN
   IF STATE EQ -1
     THEN
        *WE HAD PUTBACK*
        IF PREVSTATE EQ 3
          THEN
            *WE HAVE A NUMBER*
            ALAY := NUMALF
          ELSE
            IF (PREVSTATE IN IDSTATES)
              THEN
                *WE HAVE AN IDENTIFIER*
                ALAY := IDENTALF
              ELSE
                *NOT IDENT OR NUM*
                PACKTOK
     ELSE
        *NO PUTBACK, CANNOT BE NUM OR IDENT*
        *PACKTOK ;
  END ;

  PROCEDURE TOKERR ;
  *ERROR SUMMARY*
  *NOT AT EOF    1) PREVSTATE 38 . NOT FOLLOWED BY .*
  *              2) PREVSTATE 41 JUST A DOWN ARROW BY ITSELF*
  *AT EOF        1) STATE     38 MIDWAY THRO A ..*
  *              2) STATE     39 MIDWAY THRO A COMMENT*
  *WE EMPLOY ARGV ERROR CORRECTION*
  VAR       INDEXA : INTEGER ;
  BEGIN
   *PRINT OUT INVALID TOKEN*
```

```
ENDTOK ;
WRITE(= INVALID TOKEN =) ;
FOR INDEXA := 1 TO AKA DO  WRITE(ALAY(INDEXA)) ;
WRITE(EOL) ;
IF STATE EQ -2
  THEN
      IF PREVSTATE EQ 38
        THEN
            *RETURN ..*
            ALAY := PERPERALF
        ELSE
            IF PREVSTATE EQ 1
              THEN
                  *DELETE IT AND LOOK FOR A NEW TOKEN*
                  GOTO EXIT 333
              ELSE
  ELSE
      IF STATE EQ 38
        THEN
            BEGIN
               *RETURN A ..*
               ALAY := PERPERALF ;
               *RESET STATE FOR LAST CALL*
               STATE := 1 ;
            END
        ELSE
            IF STATE EQ 39
              THEN
                  BEGIN
                     *DELETE IT*
                     *FIRST SET STATE FOR LAST CALL*
                     STATE := 1 ;
                     GOTO EXIT 333 ;
                  END ;
END ;
```

* *

```
PROCEDURE EOFCHECK ;
BEGIN
IF (STATE EQ 38) OR (STATE EQ 39)
  THEN
      *INCOMPLETE TOKEN*
      BEGIN
         WRITE(= EOF AND INCOMPLETE TOKEN=,EOL) ;
         TOKERR ;
      END
  ELSE
      *COMPLETE TOKEN OR NO TOKEN*
      BEGIN
         WRITE(= CORRECT EOF=,EOL) ;
         IF INPCHARCOUNT EQ 1
           THEN
               *NO TOKEN*
               ALAY := ENDALF
           ELSE
               *COMPLETE TOKEN, WE HAD PUTBACK*
               BEGIN
                  ENDTOK ;
                  *FORCE BACK TO NATCHAR FOR FIRST CHAR NEXT TOK*
                  STATE := 1 ;
```

```
            END ;
        END ;
    GOTO EXIT 444 ;
END ;

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

PROCEDURE NXTCHAR ;
BEGIN
 IF EOF(SOURCE)
   THEN
       EOFCHECK
   ELSE
       WHILE (SOURCE↑ IN CHARNOT) DO
           BEGIN
             GET(SOURCE) ;
             IF EOF(SOURCE)
               THEN
                   EOFCHECK ;
           END ;
 ↑SET UP CHARIN↑
 ↑AN EOL IS REPLACED BY A BLANK↑
 ↑THIS SIMPLIFIES DELIMITING↑
 IF SOURCE↑ EQ EOL
   THEN
       CHARIN := ' '
   ELSE
       CHARIN := SOURCE↑ ;
 GET(SOURCE) ;
END ;

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

BEGIN
333 : ;
 FOR INDEX := 2 TO 3 DO  INPCHARS[INDEX] := ' ' ;
 INPCHARSCT := 1 ;
 IF STATE NE -1
   THEN
       ↑WE DID NOT HAVE PUTBACK LAST TOKEN↑
       ↑GET NEXT VALID CHAR↑
       NXTCHAR ;
 ↑NOW STRIP OUT BLANKS AND EOLS↑
 ↑NOTE THAT PUTBACK CHARS ONLY HAVE BEEN BLANK↑
 WHILE CHARIN EQ ' ' DO  NXTCHAR ;
 OUTCHAR ;
 PREVSTATE := 1 ;
 STATE := TRANMAT[1,COLNO[ORD(CHARIN)]]↑;
 WHILE STATE GT 0 DO
     BEGIN
       PREVSTATE := STATE ;
       NXTCHAR ;
       OUTCHAR ;
       STATE := TRANMAT[STATE,COLNO[ORD(CHARIN)]] ;
     END ;
 ↑NOW TEST FINAL STATE↑
 IF STATE EQ -2
   THEN
       ↑WE HAVE AN INVALID TOKEN↑
       TOKERR
   ELSE
```

```
        *WE HAVE A VALID TOKEN*
        IF PREVSTATE EQ 39
          THEN
              *WE HAVE A COMMENT*
              *GET NEXT TOKEN*
              GOTO 333
          ELSE
              *PREPARE TO RETURN TO PARSER*
              BEGIN
                *FIRST CHECK FOR PUTBACK*
                IF STATE EQ -1
                  THEN
                    PUTBACK ;
                ENDTOK ;
              END ;
444 ! ;
    *GET SYMBOLNO FOR PARSE*
    HASHLO ;
    SYMBOLNO := TTA[TABSRCH].POSN ;
  END ;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  PROCEDURE PUTSYM(ALAY:ALFARRAY) ;
  *ASSUMES CAN PRINT ON THIS LINE*
  VAR       INDEXA,
            INDEXC : INTEGER ;
            FINISH : BOOLEAN ;
  BEGIN
    INDEXC := 1 ;
    FOR INDEXA := 1 TO MAXADD
      BEGIN
        IF ALAY[INDEXA] EQ " "
          THEN
            GOTO 2 ;
        UNPACK(ALAY[INDEXA],CHAY,INDEXC) ;
        INDEXC := INDEXC + 1 ;
      END ;
    GOTO 2 ;
  1 :
    INDEXA := INDEXA - 1 ;
  2 :
    *INDEXA HAS NO OF ELEMENTS OF ALAY USED*
    *NOW PRINT*
    INDEX := 1 ;
    FINISH := FALSE ;
    REPEAT
      IF INDEX GT INDEXA*10
        THEN
          FINISH := TRUE
        ELSE
          IF CHAY[INDEX] EQ " "
            THEN
              FINISH := TRUE
            ELSE
              BEGIN
                WRITE(CHAY[INDEX]) ;
                INDEX := INDEX + 1 ;
              END ;
    UNTIL FINISH ;
  END ;
```

```
PROCEDURE PUTSYMS(FIRST,LAST:INTEGER) ;
BEGIN
 WHILE FIRST LE LAST DO
   BEGIN
     IF STACK[FIRST] NE -1
       THEN
         BEGIN
           PUTSYM(SYNTAB[STACK[FIRST]]) ;
           WRITE(' ') ;
         END ;
     FIRST := FIRST + 1 ;
   END ;
END ;
```

```
PROCEDURE PUTRED(FIRST,LAST:INTEGER) ;
BEGIN
 WRITE('RHS ') ;
 PUTSYMS(FIRST,LAST) ;
 WRITE(EOL) ;
END ;
```

```
PROCEDURE STCKSTAT ;
VAR   COUNT : INTEGER ;
BEGIN
 WRITE(EVE,EOL,'TOP OF STACK AT ',STIND:4,EOL) ;
 FOR COUNT := 1 TO STIND DO WRITE(STACK[COUNT]:5) ;
 WRITE(EOL,EVE,EOL) ;
END ;
```

```
PROCEDURE STCKFLOW ;
BEGIN
 IF STIND GT MAXSTACK
   THEN
     BEGIN
       WRITE(FILE_SET. STACK OVERFLOWE,EOL) ;
       ERRSW := 2 ;
       GOTO EXIT999 ;
     END ;
END ;
```

```
PROCEDURE REDUCE(BOTTOM,TOP : INTEGER) ;
{THIS PROCEDURE LOOKS IN THE REVERSE SYNTAX GRAPH
{FOR A STRING SPECIFIED BY STACK POSITIONS BOTTOM AND TOP.
{THE PROCEDURE DOES NOT CHANGE THE STACK. IT SETS LHS
{TO INDICATE THE RESULT OF THE SEARCH.
{WE ARE INTERESTED IN THREE OUTCOMES :
{   1) STRING IS A RHS OF A PROD. SET LHS TO THE LHS SYMBOL
{   2) STRING IS A PREFIX OF A RHS OF A PROD. SET LHS TO -2
{   3) STRING IS NOT A RHS OF A PROD, OR A PREFIX OF ONE.
{      SET LHS TO -1
```

```
VAR
        COUNT : INTEGER ;
BEGIN
 *LOOK FOR POTL HANDLE IN REV SYNTAX GRAPH*
 *CHECK FOR ONE SYMBOL POTL HANDLE*
 IF BOTTOM EQ TOP
   THEN
        *ONE SYMBOLSTRING*
        IF PPROD[STACK[BOTTOM]].PLHS EQ NIL
          THEN
              *NOT RHS OF A PRODUCTION*
              IF PPROD[STACK[BOTTOM]].PSECSYM EQ NIL
                THEN
                    *NOT PREFIX*
                    LHS := -1
                ELSE
                    *PREFIX*
                    LHS :=
          ELSE
              *RHS OF A PROD*
              BEGIN
                PREVSG := PPROD[STACK[BOTTOM]].PLHS ;
                LHS := PREVSG*.SYMBOLLHS ;
              END
   ELSE
        *STRING HAS MORE THAN ONE SYMBOL*
        BEGIN
          *LOCATE SUBTREE OF REV SYNTAX GRAPH*
          PREVSG := PPROD[STACK[BOTTOM]].PSECSYM ;
          COUNT := BOTTOM ;
          REPEAT
            IF PREVSG EQ NIL
              THEN
                BEGIN
                  *POSSIBLE RHS OF PROD HAS TOO FEW SYMBOLS*
                  LHS := -1 ;
                  GOTO 1 ;
                END ;
            COUNT := COUNT + 1 ;
            IF PREVSG*.SYMBOLR NE STACK[COUNT]
              THEN
                  *LOOK AT ALTERNATIVES*
                  REPEAT
                    PREVSG := PREVSG*.PALTP ;
                    IF PREVSG EQ NIL
                      THEN
                          BEGIN
                            *NO MORE ALTERNATIVES, FAILED TO FIND*
                            LHS := -1 ;
                            GOTO 1 ;
                          END ;
                  UNTIL PREVSG*.SYMBOLR EQ STACK[COUNT] ;
            *CHECK FOR LAST TIME THRU*
            IF COUNT NE TOP
              THEN
                  *PREPARE TO MATCH NEXT SYMBOL*
                  PREVSG := PREVSG*.PSUCCR
              ELSE
                  *CHECK IF MATCHED STRING IS THE RHS OF A PROD*
                  IF PREVSG*.PLHS EQ NIL
                    THEN
```

```
                        *NOT RHS OF A PROD*
                        *CHECK IF WE HAVE A PREFIX OF A RHS*
                        IF PREVSG*.PSUCCR NE NIL
                          THEN
                              *A PREFIX, WE HAVE SYMBOLS LEFT*
                              LHS := 0
                          ELSE
                              *NOT A RHS, AND NOT A PREFIX OF ONE*
                              LHS := -1
                    ELSE
                      BEGIN
                        PREVSG := PREVSG*.PLHS ;
                        LHS := PREVSG*.SYMBOLLHS ;
                      END ;
            UNTIL COUNT EQ TOP ;
          END ;
    *IF NO ERROR*
    *PREVSG POINTS TO THE LHS NODE*
  : :
*OPTIONAL
    WRITE(' LHS ',LHS:4,'  ') ;
    IF LHS GT 0
      THEN
        PUTSYM(SYMTAB[LHS]) ;
    WRITE(EOL) ;
PRINTING*
    END ;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *

  PROCEDURE PREC(IN1 : INTEGER ;
                 IN2 : INTEGER ) ;
  VAR       POS : INTEGER ;
            ELNO : INTEGER ;
  BEGIN
   *WORK OUT THE ELEMENT NO AND THE POSN IN THE ELEMENT*
   POS := IN2 MOD 10 ;
   IF POS EQ 0
     THEN
       POS := 10 ;
   ELNO := (IN2 + 9) DIV 10 ;
   *UNPACK CHOSEN ELEMENT*
   UNPACK(PRECMAT[IN1,ELNO],TENCHARS,1) ;
   REL := TENCHARS[POS] ;
  END ;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *

  FUNCTION LSTCKLERR(SYM1,SYM2:INTEGER):BOOLEAN;
  *CHECKS FOR LEINIUS LEFT STACKABILITY*
  BEGIN
   PREC(SYM1,SYM2) ;
   IF (REL EQ '>') OR (REL EQ '=')
     THEN
       LSTCKLERR := TRUE
     ELSE
       LSTCKLERR := FALSE ;
  END ;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *
```

```
FUNCTION NOREL(SYM1,SYM2:INTEGER):BOOLEAN;
{CHECKS FOR NO RELATIONSHIP BETWEEN TWO SYMBOLS}
BEGIN
 PREC(SYM1,SYM2) ;
 IF REL E1 = E
   THEN
       NOREL := TRUE
   ELSE
       NOREL := FALSE ;
END ;

-----------------------------------------------------------

PROCEDURE SEMANTICS(P:POINT;COND:CHAR) ;
VAR
        COUNT : INTEGER ;
        FINISH : BOOLEAN ;
BEGIN
 SEMAN := E.E ;
 PUT(SEMAN) ;
 SEMAN := COND ;
 PUT(SEMAN) ;
 SEMAN := E.E ;
 PUT(SEMAN) ;
 WHILE P <> NIL DO
    BEGIN
       UNPACK(P^.SE.ENT,TENCHARS,1) ;
       FINISH := FALSE ;
       COUNT := 1 ;
       REPEAT
          COUNT := COUNT + 1 ;
          IF COUNT = 11
             THEN
                FINISH := TRUE
             ELSE
                BEGIN
                   SEMAN := TENCHARS[COUNT] ;
                   IF SEMAN = E
                      THEN
                         FINISH := TRUE
                      ELSE
                         PUT(SEMAN) ;
                END ;
       UNTIL FINISH ;
       P := P^.POINT ;
    END ;
 SEMAN := EOL ;
 PUT(SEMAN) ;
END ;

-----------------------------------------------------------

PROCEDURE ERROR ;
TYPE
        CHEAPEST = RECORD
                     COST : INTEGER ;
                     PSTART : POINT ;
                     SYMBOL : INTEGER ;
                     CANONO : INTEGER ;
                   END ;
VAR     COUNT : INTEGER ;
```

```
        LSC : ARRAY[..MAXSE,1..31 OF BOOLEAN ;
        CHEAPEST : CHEAPESTT ;
        BOTTOM : INTEGER ;
        QUESTIA : INTEGER ;

PROCEDURE SHUNTDOWN(PRESERVE,DESTROY:INTEGER) ;
VAR     SHUNTCT : INTEGER ;
        COUNT : INTEGER ;
BEGIN
 SHUNTCT := STIND - DESTROY ;
 FOR COUNT := 1 TO SHUNTCT DO
    STACK[PRESERVE+COUNT] := STACK[DESTROY+COUNT] ;
 STIND := PRESERVE + SHUNTCT ;
END ;

PROCEDURE SHUNTUP(PRESERVE,NUMBER:INTEGER) ;
VAR     SHUNTCT : INTEGER ;
        COUNT : INTEGER ;
BEGIN
 SHUNTCT := STIND - PRESERVE ;
 FOR COUNT := SHUNTCT DOWNTO 1 DO
    STACK[PRESERVE+COUNT+NUMBER] := STACK[PRESERVE+COUNT] ;
 STIND := STIND + NUMBER ;
 STCKFLOW ;
END ;

PROCEDURE SYMBOLIN(VAR BOTTOM : INTEGER ;
                       LHS    : INTEGER) ;
(*INSERT A SYMBOL AT A < BOUNDARY*)
BEGIN
 PREC(STACK[BOTTOM-1],LHS) ;
 IF REL IN [<,=]
   THEN
     BOTTOM := BOTTOM - 1 ;
 STACK[BOTTOM] := LHS ;
END ;

PROCEDURE STROUTSYMBOLIN(VAR BOTTOM : INTEGER ;
                             TOP    : INTEGER ;
                             LHS    : INTEGER ) ;
(*THIS IS USED TO REPLACE A STRING WITH A SYMBOL.*
(*THE STRING IS ASSUMED IN THE STACK.*
(*A < IS AT THE LOWER END OF THE STRING*
(*NOTE THAT (TOP+1) CANNOT BE -1. WHEN THE PROCEDURE IS USED*
(*WITH CONDENSE, THIS IS ALWAYS TRUE AS (TOP) > (TOP+1). HOWEVER*
(*WHEN USED WITH INSERTBEST, WE COULD HAVE A < BETWEEN CAND2,*
(*AND CAND3. TO ALLOW FOR THIS TOP IS SET TO QUESTIA - 1*
  BEGIN
OPTIONAL
  STCKSTAT ;
PRINTING
  SYMBOLIN(BOTTOM,LHS) ;
  (*GET RELATION BETWEEN NEW SYMBOL AND BOTTOM OF SHUNT*
```

```
PREC(LHS,STACK[TOP+1]) ;
IF REL EQ =<=
  THEN
    IF TOP EQ BOTTOM
      THEN
        BEGIN
          SHUNTUP(TOP,1) ;
          STACK[TOP+1] := -1 ;
        END
      ELSE
        BEGIN
          STACK[BOTTOM+1] := -1 ;
          IF BOTTOM+1 LT TOP
            THEN
              SHUNTDOWN(BOTTOM+1,TOP) ;
          END
  ELSE
    IF BOTTOM NE TOP
      THEN
        SHUNTDOWN(BOTTOM,TOP) ;
*OPTIONAL
    STCKSTAT ;
PRINTING*
  END ;
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

FUNCTION LSTCKRERP(SYM1,SYM2,POS1:INTEGER):BOOLEAN;
*CHECK FOR RHODES LEFT STACKABILITY*
VAR       BOTTOM : INTEGER ;
          WSTACK : INTEGER ;
          WLHS : INTEGER ;
          WPREVSS : POINT ;
BEGIN
 PREC(SYM1,SYM2) ;
 IF REL EQ = = ;
   THEN
     LSTCKRERP := TRUE
   ELSE
     IF REL NE =<=
       THEN
         BEGIN
           *WE NEED TO CHECK OUT = AND >*
           *PRESERVE VALUE OF LHS*
           WLHS := LHS ;
           *PRESERVE POINTER TO LHS NODE*
           WPREVSS := PREVSS ;
           *LOOK IN STACK FOR FIRST -1 BELOW POS1*
           BOTTOM := POS1 ;
           WHILE STACK[BOTTOM-1] NE -1 DO BOTTOM := BOTTOM-1 ;
           IF REL EQ =>=
             THEN
               *LOOK FOR A PREFIX ENDING WITH SYM2*
               BEGIN
                 *INSERT SYM2 INTO THE STACK TEMPORARILY*
                 POS1 := POS1 + 1 ;
                 WSTACK := STACK[POS1] ;
                 STACK[POS1] := SYM2 ;
                 REDUCE(BOTTOM,POS1) ;
                 IF LHS EQ -1
                   THEN
```

```
                                LSTCKRERR := TRUE
                            ELSE
                                LSTCKRERR := FALSE ;
                            *RESTORE THE STACK*
                            STACK[POS1] := WSTACK ;
                        END
                    ELSE
                        *LOOK FOR A HANDLE ENDING WITH SYM1*
                        BEGIN
                            REDUCE(BOTTOM,POS1) ;
                            IF LHS LE
                              THEN
                                LSTCKRERR := TRUE
                            ELSE
                                LSTCKRERR := FALSE ;
                        END ;
                    *RESET VALUE OF LHS*
                    LHS := WLHS ;
                    *RESET POINTER*
                    PREVSS := WPREVSS ;
                END
            ELSE
                LSTCKRERR := FALSE ;
    END ;

. . . . . .

    PROCEDURE CONDENSE(VAR TOP : INTEGER ;
                           RTSYM : INTEGER ;
                       VAR ACTION : BOOLEAN ) ;
    VAR     BOTTOM : INTEGER ;
            SHUNT : INTEGER ;
            WORKCT : INTEGER ;
    BEGIN
*OPTIONAL
      WRITE(' ENTER CONDENSE',EOL) ;
      STKSTAT ;
*PRINTING*
      ACTION := FALSE ;
      *CALCULATE THE NO OF SYMBOLS ABOVE TOP*
      *THIS IS CONSTANT IN THIS PROCEDURE*
      SHUNT := STKNO - TOP ;
      BOTTOM := TOP ;
      WHILE REL EQ EQUAL DO
          BEGIN
              *LOCATE HANDLE IN STACK*
              WHILE STACK[BOTTOM-1] NE =: DO  BOTTOM := BOTTOM - 1 ;
              *LOOK IN REVERSE SYNTAX GRAPH*
              REDUCE(BOTTOM,TOP) ;
              IF LHS GT .
                THEN
                    *CHECK FOR LEFT STACKABILITY*
                    IF NLSTCKRERR(STACK[BOTTOM-2],LHS,BOTTOM-2)
                      THEN
                          *CHECK FOR RIGHT STACKABILITY*
                          IF NOREL(LHS,RTSYM)
                            THEN
                                *ACCEPTABLE REDUCTION*
                                BEGIN
                                  ACTION := TRUE ;
                                  *PRINT OUT REDUCTION*
```

```
*OPTIONAL
 PRINTING*
                                    PUTREDN(BOTTOM,TOP) ;
                                    IF ERRST EQ 1
                                      THEN
                                        *HAS QUEST2 ABSORBED QUEST1*
                                        IF BOTTOM LE QUEST1
                                          THEN
                                            QUEST1 := 2 ;
                                    IF PREVSG*.PSEMR NE NIL
                                      THEN
                                        SEMANTICS(PREVST*.PSEMR,ECE) ;
                                    *UPDATE THE STACK*
                                    IF SHUNT EQ 2
                                      THEN
                                        *STRING AT TOP OF STACK*
                                        BEGIN
                                          SYMBOLIN(BOTTOM,LHS) ;
                                          STIND := BOTTOM ;
                                          *ASSUME A >*
                                          REL := E>E ;
                                        END
                                      ELSE
                                        *WE REPLACE AN EMBEDDED STRING*
                                        STROUTSYMIN(BOTTOM,TOP,LHS) ;
                                    TOP := BOTTOM ;
                                  END
                                ELSE
                                  REL := E E
                              ELSE
                                REL := E E
                          ELSE
                            REL := E E
                      END ;
*OPTIONAL
    STCKSTAT ;
    WRITE(E LEAV  COMMENCE,EOL) ;
 PRINTING*
    END ;

. . . . . .

    PROCEDURE RACKOVE  ;
    VAR       FINISH : BOOLEAN ;
              STARTPT : INTEGER ;
              ACTION : BOOLEAN ;
    BEGIN
*OPTIONAL
      WRITE(E ENTER RACK-OVE E,ERRST:2,EOL) ;
 PRINTING*
      *ASSUME A > IS ALWAYS BETWEEN TOP OF STACK AND INPUT SYMBOL*
      REPEAT
        *FIRST CHECK OUT TOP OF STACK*
        REL := E>E ;
        CONDENSE(STIND,SYMBOLNO,ACTION) ;
        *NOW GO DOWN STACK LOOKING FOR EMBEDDED REDUCTIONS. IF WE*
        *FIND A RED, HERE, WE REPEAT STARTING AT TOP*
        FINISH := TRUE ;
        STARTPT := STIND ;
        WHILE STARTPT GE 3 DO
          BEGIN
```

```
            *LOOK FOR A >*
            REL := EQL ;
            WHILE (STARTPT GE 3) AND (REL NE EQL) DO
              BEGIN
                STARTPT := STARTPT - 1 ;
                IF STACK[STARTPT] EQ -1
                  THEN
                    *DROP THRO A <*
                    STARTPT := STARTPT - 1
                  ELSE
                    *WE HAVE ADJACENT SYMBOLS*
                    PREC(STACK[STARTPT],STACK[STARTPT+1]) ;
              END ;
            IF REL EQ EQL
              THEN
                *WE HAVE A POTL REDUCTION*
                BEGIN
                  CONDENSE(STARTPT,STACK[STARTPT+1],ACTION) ;
                  IF ACTION
                    THEN
                      BEGIN
                        FINISH := FALSE ;
                        *CONDENSATIONS ARE DISJOINT*
                        IF STACK[STARTPT-1] EQ -1
                          THEN
                            STARTPT := STARTPT - 2
                          ELSE
                            STARTPT := STARTPT - 1 ;
                      END ;
                END ;
            END ;
        UNTIL FINISH ;
        *CHECK FOR END OF PASS*
        IF (ALAY EQ ENDALF) AND (STACK[3] EQ 1) AND (STIND EQ 3)
          THEN
            *END OF PASS*
            GOTO EXIT999 ;
    *OPTIONAL
        WRITE(LEAVE TASK OVER,EOL) ;
    PRINTING*
        END ;

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        PROCEDURE SPECIALREC ;
        BEGIN
          *WRITE(E,SPECIAL RECOVERY REQUIRED,EOL) ;
          STCKSTAT ;
          GOTO EXIT999 ;
        END ;

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        PROCEDURE LOCSYNCOMP ;
        *ON ENTRY TWO ERROR POINTS ARE DEFINED*
        *FIRST ERROR POINT OCCURS ABOVE QUEST1*
        *SECOND ERROR POINT IS BETWEEN TOP OF STACK AND INPUT SYMBOL*
        *IF THE FIRST ERROR POINT HAS BEEN ABSORBED BY THE SECOND, *
        *QUEST1 IS : *
        *CANDIDATE 1  < TO ERROR POINT 2*
        *CANDIDATE 2  < TO ERROR POINT 1 (QUEST1)*
```

```
    ♦CANDIDATE 3  ERROR POINT 1 (QUEST1A) TO ERROR POINT 2♦
    VAR
            CANDNO : INTEGER ;
    BEGIN
      ♦INITIALISE THE REPLACEMENT SETS♦
      ♦THE ZEROTH ELEMENTS ARE USED FOR DELETION♦
      FOR LHS := 0 TO NONTERM DO
        FOR CANDNO := 1 TO 3 DO
          LSC[LHS,CANDNO] := FALSE ;
      ♦INITIALISE LEFTMOST RECOVERY POINT, AND SET QUEST1A♦
      IF QUEST1 EQ 0
        THEN
          BOTTOM := STIND
        ELSE
          BEGIN
            BOTTOM := QUEST1 ;
            IF STACK[QUEST1+1] EQ -1
              THEN
                QUEST1A := QUEST1 + 2
              ELSE
                QUEST1A := QUEST1 + 1 ;
          END ;
      ♦NOW LOCATE LEFTMOST RECOVERY POINT♦
      WHILE STACK[BOTTOM-1] NE -1 DO BOTTOM := BOTTOM - 1 ;
      ♦THE < FOR LEFTMOST RECOVERY IS NOW UNDER BOTTOM♦
♦OPTIONAL
      WRITE(ELSE,♦CANDIDATES ATE♦,EOL) ;
PRINTING♦
      WRITE(♦CANDIDATES ATE♦,EOL) ;
      WRITE(♦ BOTTOM:♦,BOTTOM:5,EOL,
            ♦ QUEST1:♦,QUEST1:5,EOL,
            ♦ STACK :♦,STIND:5,EOL) ;
♦OPTIONAL
      WRITE(♦CALLS TO REDUCE FOR RHODES LEFT STACKABILITY♦,
            EOL,♦ ♦,EOL) ;
PRINTING♦
      ♦FIRST EXAMINE LSC DELETIONS FOR THE CANDIDATES♦
      IF PLSTCKREPR(STACK[BOTTOM-2],SYMBOLNO,BOTTOM-2)
        THEN
          LSC[ ,2] := TRUE ;
      IF QUEST1 GT
        THEN
          BEGIN
            IF PLSTCKREPR(STACK[BOTTOM-2],STACK[QUEST1A],BOTTOM-2)
              THEN
                LSC[ ,2] := TRUE ;
            IF PLSTCKREPR(STACK[QUEST1],SYMBOLNO,QUEST1)
              THEN
                LSC[ ,3] := TRUE ;
          END ;
      ♦NOW FIND THE LSC NTS FOR EACH CANDIDATE♦
      FOR LHS := 1 TO NONTERM DO
        BEGIN
          IF PLSTCKREPR(STACK[BOTTOM-2],LHS,BOTTOM-2)
            THEN
              IF PNOPEL(LHS,SYMBOLNO)
                THEN
                  LSC[LHS,1] := TRUE ;
          IF QUEST1 NE
            THEN
              BEGIN
```

```
                IF ¬LSTCKRERR(STACK[BOTTOM-2],LHS,BOTTOM-2)
                  THEN
                     IF ¬NOREL(LHS,STACK[QUEST1A])
                       THEN
                          LSC[LHS,2] := TRUE ;
                IF ¬LSTCKRERR(STACK[QUEST1],LHS,QUEST1)
                  THEN
                     IF ¬NOREL(LHS,SYMBOLNO)
                       THEN
                          LSC[LHS,3] := TRUE ;
             END ;
      END ;
*OPTIONAL
    WRITE(=1  REPLACEMENT SETS=,EOL) ;
    WRITE(=  NT      CANDIDATES=,EOL) ;
    WRITE(=         1     2     3=,EOL,= =,EOL) ;
    FOR LHS := 1 TO NONONTERM DO
      BEGIN
        WRITE(= =,LHS:4) ;
        FOR CANDNO := 1 TO 3 DO
          WRITE(LSC[LHS,CANDNO]:5) ;
        WRITE(EOL) ;
      END ;
    WRITE(=0NON TERMINAL NUMBER 1 IS DELETION=,EOL) ;
PRINTING*
    FOR LHS := 2 TO NONONTERM DO
      FOR CANDNO := 1 TO 3 DO
        IF LSC[LHS,CANDNO]
          THEN
            GOTO 2 ;
    *THE SET HAS NO MEMBERS*
    WRITE(=1REPLACEMENT SETS ARE EMPTY=,EOL) ;
    SPECIALREC ;
2 :
   END ;
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

    FUNCTION COSTLSCPHF(PWORK:POINT;
                        LOW,HIGH:INTEGER):INTEGER;
    *THE SYMBOL OPERATIONS INSERTION, DELETION AND REPLACEMENT APPLY *
    *TO THE CANDIDATE IN THE STACK. THE LSC PHRASE IS USED*
    *AS THE CONTROL*
    *PATTERN MATCHING STRATEGY IS AS FOLLOWS*
    *IF WE HAVE A MISMATCH WE LOOK AT THE REST OF THE CANDIDATE FROM*
    *LEFT TO RIGHT. IF WE (1) GET A MATCH, WE ADD IN THE COST OF*
    *DELETING THE INBETWEEN SYMBOLS. IF WE (2) DO NOT GET A MATCH,*
    *WE LOOK AT THE REST OF THE CANDIDATE FOR A REPLACE. IF THIS*
    *IS (A) O.K., WE ADD IN THE COST OF REPLACEMENT AND ANY SYMBOLS*
    *INBETWEEN THAT WERE DELETED. IF THIS IS (B) NOT O.K., WE*
    *ADD IN THE COST OF INSERTION*
    VAR
          WCOST : INTEGER ;
          COUNT : INTEGER ;
          FNV : INTEGER ;
    BEGIN
      FNV := 0 ;
      WHILE (PWORK NE NIL) AND (LOW LE HIGH) DO
        *STRIP OUT -1 FROM STACK*
        IF STACK[LOW] EQ -1
          THEN
```

```
        LOW := LOW + 1
    ELSE
        BEGIN
            IF PWORK^.SYMBOL EQ STACK[LOW]
                THEN
                    LOW := LOW + 1
                ELSE
                    (*MISMATCH*)
                    (*CHECK REST OF STRING ON STACK*)
                    BEGIN
                    WCOST := 0 ;
                    COUNT := LOW ;
                    WHILE (COUNT LE HIGH) AND
                        (STACK[COUNT] NE PWORK^.SYMBOL) DO
                            BEGIN
                                WCOST := WCOST + SYMCOST
                                        [STACK[COUNT]].DELETE ;
                                IF STACK[COUNT+1] EQ -1
                                    THEN
                                        COUNT := COUNT + 2
                                    ELSE
                                        COUNT := COUNT + 1 ;
                            END ;
                    (*CHECK ON FINAL CONDITION*)
                    IF COUNT LE HIGH
                        THEN
                            (*MATCH FOUND IN REMAINDER OF STRING*)
                            (*ADD IN DELETION COST OF INBETWEEN SYMBOLS*)
                            BEGIN
                                ENV := ENV + WCOST ;
                                LOW := COUNT + 1 ;
                            END
                        ELSE
                            (*NO MATCH FOUND IN REMAINDER*)
                            (*CHECK FOR REPLACEMENT OR INSERTION*)
                            BEGIN
                            WCOST := 0 ;
                            COUNT := LOW ;
                            WHILE (COUNT LE HIGH) AND (PWORK^.SYMBOL
                                NE SYMCOST[STACK[COUNT]].REPMENT) DO
                            BEGIN
                                WCOST := WCOST + SYMCOST
                                        [STACK[COUNT]].DELETE ;
                                IF STACK[COUNT+1] EQ -1
                                    THEN
                                        COUNT := COUNT + 2
                                    ELSE
                                        COUNT := COUNT + 1 ;
                            END ;
                            (*CHECK FINAL CONDITION*)
                            IF COUNT LE HIGH
                                THEN
                                    (*REPLACE A SYMBOL*)
                                    BEGIN
                                        LOW := COUNT + 1 ;
                                        ENV := ENV + WCOST + SYMCOST
                                                [STACK[COUNT]].REPLACE ;
                                    END
                                ELSE
                                    (*INSERT*)
                                    ENV := ENV + SYMCOST
```

```
                                        +(PWORK+.SYMBOL).INSERT ;
                        END ;
                    END ;
                PWORK := PWORK+.PSUCC ;
            END ;
        *TEST FOR FINAL CONDITIONS*
        IF STACK[LOW] EQ -1
          THEN
            LOW := LOW + 1 ;
        *ADD IN COST OF DELETING THE EXTRA SYMBOLS*
        WHILE LOW LE HIGH DO
          BEGIN
            FNV := FNV + SYMCOST[STACK[LOW]].DELETE ;
            IF STACK[LOW+1] EQ -1
              THEN
                LOW := LOW + 2
              ELSE
                LOW := LOW + 1 ;
          END ;
        *ADD IN COST OF INSERTING THE EXTRA SYMBOLS*
        WHILE PWORK <> NIL DO
          BEGIN
            FNV := FNV + SYMCOST[PWORK+.SYMBOL].INSERT ;
            PWORK := PWORK+.PSUCC ;
          END ;
*OPTIONAL
        WRITE(FNV:7,EOL) ;
*PRINTING*
        COSTLSCPHR := FNV ;
      END ;

  . . . . . .

      PROCEDURE FINDBEST ;
      VAR       CANDCT : INTEGER ;
                PSYNCE : POINTER ;
                NOCANDS : INTEGER ;
                TEMPCOST : INTEGER ;
                HIGH : INTEGER ;
                LOW : INTEGER ;
      BEGIN
*OPTIONAL*
        WRITE('COST OF LSC PHRASES AND THE EMPTY WORD',EOL) ;
        WRITE('CANDIDATE  NT   COST',EOL) ;
*PRINTING*
        IF QUEST1 EQ 1
          THEN
            NOCANDS := 1
          ELSE
            NOCANDS := 3 ;
        *INITIALISE*
        CHEAPEST.COST := MAXCOST+1 ;
        FOR CANDCT := 1 TO NOCANDS DO
          BEGIN
*OPTIONAL
            WRITE(' ',EOL) ;
*PRINTING*
            *SET UP LOW AND HIGH*
            IF CANDCT EQ 1
              THEN
                BEGIN
```

```
                LOW := BOTTOM ;
                HIGH := STIND ;
            END
        ELSE
            IF CANDCT EQ 2
              THEN
                  BEGIN
                      LOW := BOTTOM ;
                      HIGH := QUEST1 ;
                  END
              ELSE
                  BEGIN
                      LOW := QUEST1A ;
                      HIGH := STIND ;
                  END ;
        *FIND COST OF DELETION*
        IF LSC(,CANDCT)
          THEN
              BEGIN
*OPTIONAL
                  WRITE(F E,CANDCT:9,E    DE) ;
 PRINTING*
                  TEMPCOST := COSTLSCPHR(NIL,LOW,HIGH) ;
                  IF TEMPCOST LE CHEAPEST.COST
                    THEN
                        BEGIN
                          CHEAPEST.COST := TEMPCOST ;
                          CHEAPEST.PSTART := NIL ;
                          CHEAPEST.SYMBOL := ' ' ;
                          CHEAPEST.CANDNO := CANDCT ;
                        END ;
              END ;
        *FIND COST OF REPLACEMENT*
        FOR LHS := 1 TO NONTERM DO
          IF LSC(LHS,CANDCT)
            THEN
                BEGIN
                  PSYNGR := PSIDEF[LHS] ;
                  WHILE PSYNGR NE NIL DO
                    BEGIN
*OPTIONAL
                      WRITE(F E,CANDCT:9,LHS:5) ;
 PRINTING*
                      TEMPCOST := COSTLSCPHR(PSYNGR,LOW,HIGH) ;
                      IF TEMPCOST LE CHEAPEST.COST
                        THEN
                            BEGIN
                              CHEAPEST.COST := TEMPCOST ;
                              CHEAPEST.PSTART := PSYNGR ;
                              CHEAPEST.SYMBOL := LHS ;
                              CHEAPEST.CANDNO := CANDCT ;
                            END ;
                      *GET NEXT ALTERNATIVE*
                      PSYNGR := PSYNGR*.PALT ;
                    END ;
                END ;
    END ;
    *CHECK ON CHEAPEST*
    IF CHEAPEST.COST GT MAXCOST
      THEN
        *TOO EXPENSIVE*
```

```
            BEGIN
              WRITE('MODES RECOVERY TOO EXPENSIVE',EOL) ;
              SPECIAL:=0 ;
            END ;
      (*LOOK AFTER SEMANTICS*)
      PSYNGR := CHEAPEST.PSTART ;
      IF PSYNGR = NIL
        THEN
            (*DELETION*)
            SEMANTICS(NIL,EDE)
        ELSE
            (*REPLACEMENT*)
            BEGIN
              WHILE PSYNGR^.PSUCC <> NIL DO PSYNGR := PSYNGR^.PSUCC ;
              IF PSYNGR^.PSEM <> NIL
                THEN
                    SEMANTICS(PSYNGR^.PSEM,ERE) ;
            END ;
(*OPTIONAL
      WRITE('CHOSEN MEMBERE',EOL) ;
PRINTING*)
      WRITE('CHOSEN MEMBERE',EOL) ;
      WRITE('      COST =',CHEAPEST.COST:5,EOL,
            '    SYMBOL =',CHEAPEST.SYMBOL:5,EOL,
            '     CANO  =',CHEAPEST.CANONO:5,EOL) ;
    END ;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

    PROCEDURE PUTCANDREP ;
    VAR     FIRST : INTEGER ;
            LAST : INTEGER ;
    BEGIN
      (*SET UP FIRST AND LAST*)
      IF CHEAPEST.CANONO = 1
        THEN
            BEGIN
              FIRST := BOTTOM ;
              LAST := STIND ;
            END
        ELSE
            IF CHEAPEST.CANONO = 2
              THEN
                  BEGIN
                    FIRST := BOTTOM ;
                    LAST := QUEST1 ;
                  END
              ELSE
                  BEGIN
                    FIRST := QUEST1A ;
                    LAST := STIND ;
                  END ;
      WRITE('CANDIDATE    ') ;
      PUTSYMS(FIRST,LAST) ;
      WRITE(EOL) ;
      IF CHEAPEST.SYMBOL = 0
        THEN
            WRITE('DELETED') -
        ELSE
            BEGIN
              WRITE('REPLACEMENT  ') ;
```

```
            PUTSYM(SYMTAB[CHEAPEST.SYMBOL]) ;
        END ;
    WRITE(EOL) ;
END ;

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

PROCEDURE INSERTBEST ;
*THE CANDIDATES ARE DEFINED BY BOTTOM, QUEST1, QUEST1A AND STIND*
BEGIN
  STCKSTAT ;
  IF CHEAPEST.PSTART NE NIL
    THEN
        *REPLACEMENT*
        IF CHEAPEST.CANDNO EQ 1
          THEN
              BEGIN
                  SYMBOLIN(BOTTOM,CHEAPEST.SYMBOL) ;
                  STIND := BOTTOM ;
              END
          ELSE
            IF CHEAPEST.CANDNO EQ 2
              THEN
                  STROUTSYMIN(BOTTOM,QUEST1A-1,CHEAPEST.SYMBOL)
              ELSE
                  *CANDIDATE 3*
                  IF QUEST1A EQ QUEST1+2
                    THEN
                        *< AT LOWER END*
                        BEGIN
                            SYMBOLIN(QUEST1A,CHEAPEST.SYMBOL) ;
                            STIND := QUEST1A ;
                        END
                    ELSE
                        *NO < AT LOWER END*
                        BEGIN
                            PREC(STACK[QUEST1],CHEAPEST.SYMBOL) ;
                            IF REL EQ <=
                              THEN
                                  BEGIN
                                      STACK[QUEST1A] := -1 ;
                                      QUEST1A := QUEST1A + 1 ;
                                  END ;
                            STIND := QUEST1A ;
                            STCKFLOW ;
                            STACK[QUEST1A] := CHEAPEST.SYMBOL ;
                        END
    ELSE
        *DELETION*
        IF CHEAPEST.CANDNO EQ 1
          THEN
              STIND := BOTTOM - 2
          ELSE
              IF CHEAPEST.CANDNO EQ 2
                THEN
                    *CAND 2 - DELETE AN EMBEDDED STRING*
                    BEGIN
                        PREC(STACK[BOTTOM-2],STACK[QUEST1A]) ;
                        IF REL EQ <=
                          THEN
                              SHUNTDOWN(BOTTOM-1,QUEST1A-1)
```

```
                    ELSE
                        SHUNTDOWN(BOTTOM-2,QUEST1A-1) ;
                END
            ELSE
                STIND := QUEST1 ;
    STCKSTAT ;
    *CHECK FOR END OF PARSE*
    IF (ALAY EQ ENDALF) AND (STACK[3] EQ 1) AND (STIND EQ 3)
      THEN
        *END OF PARSE*
        GOTO EXIT 999 ;
  END ;
```

```
  BEGIN
*OPTIONAL
    WRITE(E,ERROR,EOL) ;
 PRINTING*
    *FOR TRUNCATED PRINT*
    IF ERRST EQ .
      THEN
        WRITE(E,INITIAL ERROR)
      ELSE
        WRITE(E,AFTER FORWARD MOVE) ;
    WRITE(E, PARSER CONFIGURATION,EOL) ;
    WRITE(ECSYMBOL,SYMBOL,314,E ) ;
    PUTSYM(ALAY) ;
    WRITE(EOL) ;
    STCKSTAT ;
    IF ERRST EQ .
      THEN
        BEGIN
          *FIRST BACKWARD MOVE*
          BACKMOVE ;
          *FOR TRUNCATED PRINT*
*OPTIONAL
PRINTING*
          WRITE(E,STACK AFTER FIRST BACKWARD MOVE,EOL) ;
          STCKSTAT ;
          *CHECK FOR END OF INPUT*
          IF ALAY EQ ENDALF
            THEN
              *CAN JUST DO ONE BACKWARD MOVE*
              *NO FORWARD AND NO SECOND BACKWARD*
              BEGIN
                LOOPCT := LOOPCT + 1 ;
                IF LOOPCT EQ LOOPLIM
                  THEN
                    BEGIN
                      WRITE(E,POTENTIAL LOOP,EOL) ;
                      SPECIALREC ;
                    END ;
                QUEST1 := 1 ;
                GOTO 1 ;
              END ;
          QUEST1 := STIND ;
          ERRST := 1 ;
          *NOW PRIME THE SYSTEM FOR THE FORWARD MOVE*
          *SET REL 0 <, PUT -1 IN THE STACK*
          STIND := STIND + 1 ;
```

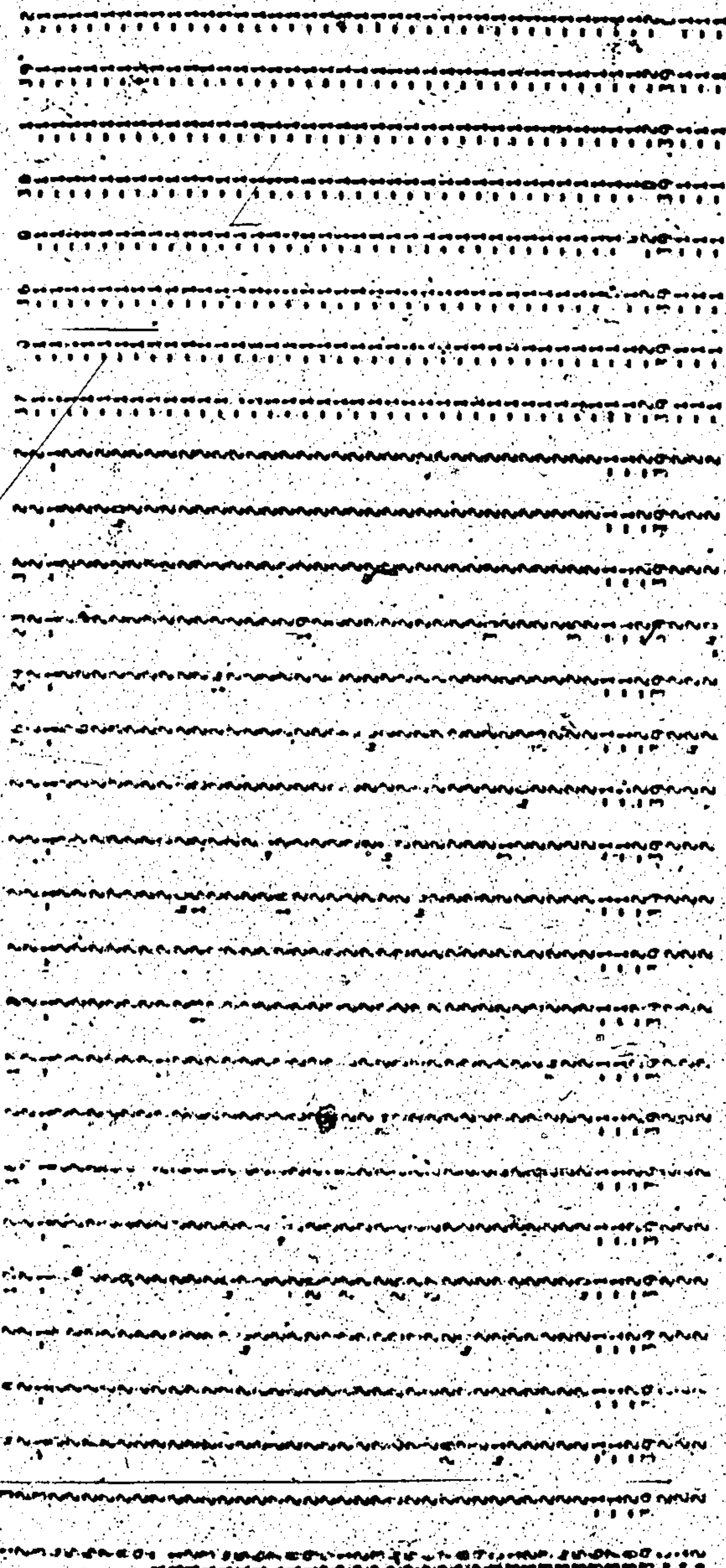
```
                STICKFLOW ;
                STACK(STIND) := -1 ;
                STIND := STIND + 1 ;
                STICKFLOW ;
                STACK(STIND) := SYMBOLNO ;
                *GET NEXT TOKEN*
                LEXASSUB ;
*OPTIONAL
                WRITE(E,'SYMBOLE',SYMBOLNO:4,E,'E') ;
                PUTSYM(ALAY) ;
                WRITE(EOL) ;
 PRINTING*
                IF ALAY <> ENDALF
                  THEN
                    GOTO EXIT 333 ;
                *IF ALAY = ENDALF THEN QUEST2 IS DEFINED AS*
                *THE END OF THE STRING*
              END ;
        *NOW DO A SECOND BACKWARD MOVE*
        IF QUEST2 <> 0
          THEN
            *QUEST1 NOT ABSORBED IN FORWARD MOVE*
            IF STACK(QUEST1+2) = -1
              THEN
                *CHECK IF -1 SHOULD BE REMOVED*
                BEGIN
                  PREC(STACK(QUEST1),STACK(QUEST1+2)) ;
                  IF REL <> EQ
                    THEN
                      SHUNTDOWN(QUEST1,QUEST1+1) ;
                END ;
        BACKMOVE ;
        *FOR TRUNCATED PRINT*
*OPTIONAL
 PRINTING*
        WRITE(E,'STACK AFTER SECOND BACKWARD MOVE',EOL) ;
        STICKSTAT ;
        *THE TWO ERROR ROUTINES ARE NOW DEFINED*
        *FIND THE REPLACEMENT SET FOR EACH CANDIDATE*
        *A MEMBER OF A REPLACEMENT SET IS EITHER A DELETION OR A*
        *NONTERMINAL. THE LATTER DELETION IS CALLED A LOCALLY*
        *SYNTACTICALLY CORRECT(LSC) DELETION. IF THE MEMBER IS A*
        *NONTERMINAL IT IS CALLED AN LSC NONTERMINAL (LSC NT).*
        *THE RIGHT HAND SIDES OF THE RULES FOR AN LSC NT ARE CALLED*
        *LSC PHRASES*
        ;
        LOCSYNCORR ;
        *NOW FIND THE BEST MEMBER OF THE REPLACEMENT SETS*
        FINDBEST ;
        *PRINT OUT RECOVERY ACTION*
*OPTIONAL
 PRINTING*
        PUTCANDREP ;
        *IF THE CHOSEN MEMBER IS DELETION, DELETE ITS CANDIDATE,*
        *OTHERWISE REPLACE ITS CANDIDATE WITH THE CHOSEN MEMBER,*
        *A NONTERMINAL*
        INSERTBEST ;
        *RESET ERROR STATE AND RETURN TO PARSER*
        ERRST := 0 ;
        WRITE(E,'ERRHOOES RECOVERY OVER, PARSING CONTINUES.',E,EOL) ;
*OPTIONAL
```

```
      WRITE(E1E,EOL) ;
PRINTING*
      GOTO EXIT 333 ;
    END ;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    BEGIN
      INIT ;
      WHILE ALAY NE ENDALF DO
        BEGIN
          *GET A NEW TOKEN*
          LEXALGSUB ;
*OPTIONAL
          WRITE(E SY*OLE,SY<3OLNO!4,E  E) ;
          PUTSYM(ALAY) ;
          WRITE(EOL) ;
 PRINTING*
333 !
          ;
          *RETURN HERE AFTER A SYNTAX ERROR HAS BEEN DEALT WITH*
*OPTIONAL
          STCKSTAT ;
 PRINTING*
          *CHECK OUT CHARACTER PAIR RELATION*
          IF NOREL(STACK(STIND),SY*BOLNO)
            THEN
              *ERROR TYPE  - CHARACTER PAIR*
              ERROR ;
          *REL IS = < OR >*
          *IF POSSIBLE CARRY OUT REDUCTIONS*
          ENDCT := STIND ;
          WHILE (REL EQ <*) AND *FINISH DO
            BEGIN
              *LOCATE END OF POTL HANDLE IN STACK*
              *END IS FLAGGED BY -:*
              *ENDCT HAS BEEN SET TO STIND*
              WHILE STACK(ENDCT-1] NE -: DO ENDCT := ENDCT - 1 ;
              *NOW LOOK IN REVERSE SYNTAX GRAPH*
              REDUCE(ENDCT,STIND) ;
              IF LHS EQ
                THEN
                  *: COULD NOT FIND THE POTL HANDLE*
                  *TYPE 1 PHRASE ERROR*
                  ERROR ;
              *CHECK FOR RIGHT STACKABILITY*
              IF NOREL(LHS,SYMBOLNO)
                THEN
                  *TYPE 3 PHRASE ERROR*
                  ERROR ;
              *CHECK FOR LEFTIUS LEFT STACKABILITY*
              IF LSTCKERR(STACK(ENDCT-2),LHS)
                THEN
                  *TYPE 2 PHRASE ERROR*
                  ERROR ;
              *NO STACKABILITY ERRORS*
              *PRINT OUT RED*
*OPTIONAL
              PUTREDN(ENDCT,STIND) ;
PRINTING*
              IF ERRST EQ *
                THEN
```

```
                  IF ENDCT LE QUEST1
                    THEN
                       *QUEST1 WILL BE ABSORBED BY QUEST2*
                       QUEST1 := 0 ;
              *UPDATE THE STACK*
              IF REL NE E<E
                THEN
                    *DELETE -1 FROM STACK*
                    ENDCT := ENDCT - 1 ;
              STIND := ENDCT ;
              STACK[STIND] := LHS ;
              IF PREVSQ*.PSEMR NE NIL
                THEN
                    SEMANTICS(PREVSQ*.PSEMR,E E) ;
              *AFTER A REDN SEE IF THE PARSER IS IN FINAL STATE*
              IF (ALAY EQ ENDALF) AND (STACK[3] EQ 1)
                THEN
                    FINISH := TRUE ;
              PREC(LHS,SYMBOLNO) ;
           END ;
        IF REL EQ E<E
          THEN
              *INSERT A FLAG OF -1 INTO THE STACK*
              BEGIN
                 STIND := STIND + 1 ;
                 STCKFLOW ;
                 STACK[STIND] := -1 ;
              END ;
        *SHIFT THE INPUT SYMBOL*
        STIND := STIND + 1 ;
        STCKFLOW ;
        STACK[STIND] := SYMBOLNO ;
     END ;
   WRITE(E1END OF PARSEE,EOL) ;
999 : ;
   END ;


               END OF PARSE
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


   BEGIN
    WRITE(E1TRANSITION MATRIX - ALGOL SUBSETE,EOL,E E,EOL) ;
    FOR SETABCT := 1 TO 42 DO
       BEGIN
         WRITE(E E,SETABCT:3) ;
         FOR ITABCT := 1 TO 29 DO
            WRITE(E E,TRANMAT[SETABCT,ITABCT]:3) ;
         WRITE(EOL) ;
       END ;
    WRITE(E1RMODES RECOVERY COSTS - ALGOL SUBSETE,EOL) ;
    WRITE(ESYMBOL INSERT DELETE REPLACE REPMENTE,EOL,E E,EOL) ;
    FOR SETABCT := 1 TO 72 DO
      WRITE(E E,SETABCT:6,
                 SYMCOST[SETABCT].INSERT:7,
                 SYMCOST[SETABCT].DELETE:7,
                 SYMCOST[SETABCT].REPLACE:8,
                 SYMCOST[SETABCT].REPMENT:8,EOL) ;
*OPTIONAL
```

```
      WRITE(=1=,EOL) ;
 PRINTING:
      CHARNOT := (=$=,      ====,=[=,=]=);
      IOSTATES := (2,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                  21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,
                  41,42) ;
      READGRAPHS ;
      PARSE ;
      IF ERRSW EQ 1
        THEN
            GOTO 200 ;
      IF ERRSW EQ 2
        THEN
            GOTO 200 ;
 200 : ;
     END.
```

```
¢A SIMPLE SENTENCE WITH NO INPUT ERRORS¢
START
      BEGIN¢
        A:=3;
        GO TO LAB2 ;
        P:=0 ;
      END
```



RHODES RECOVEFY COSTS - ALGOL SUBSET

SYMBOL	INSERT	DELETE	REPLACE	REPHENT
1	3	3	0	0
2	3	3	0	0
3	8	99	2	5
4	3	4	0	0
5	8	99	0	0
6	3	3	0	0
7	3	3	0	0
8	3	3	0	0
9	3	3	0	0
10	2	3	0	0
11	5	6	0	0
12	3	3	2	4
13	5	6	0	0
14	2	4	0	0
15	3	3	0	0
16	3	3	0	0
17	3	3	0	0
18	3	3	0	0
19	3	5	5	64
20	3	3	0	0
21	5	6	0	0
22	3	5	5	54
23	3	3	0	0
24	3	3	0	0
25	3	3	0	0
26	3	3	0	0
27	3	3	0	0
28	3	3	0	0
29	3	3	0	0
30	3	3	0	0
31	3	3	0	0
32	6	5	0	0
33	5	6	0	0
34	3	3	0	0
35	3	3	0	0
36	3	3	0	0
37	3	3	0	0
38	3	3	0	0
39	3	3	0	0
40	3	5	0	0
41	3	3	0	0
42	3	3	0	0
43	3	3	0	0
44	2	3	0	0
45	2	3	0	0
46	3	3	0	0
47	2	3	0	0
48	2	3	0	0
49	3	4	5	64
50	3	4	5	64
51	3	4	5	54
52	3	4	0	0
53	1	5	5	64
54	3	5	5	64
55	4	4	5	64
56	4	4	5	64
57	6	5	5	64
58	2	3	0	0
59	3	4	5	64
60	2	3	0	0
61	6	5	0	0
62	3	3	0	0
63	3	5	0	0
64	3	5	0	0
65	3	5	5	64
66	6	5	5	64
67	1	99	0	0
68	8	99	0	0
69	8	99	0	0
70	8	99	0	0

```
SYMBOL  71  <START>

TOP OF STACK AT     1
  71

SYMBOL  66  <BEGIN>

TOP OF STACK AT     3
  71   -1   7(

SYMBOL  68  <*>

TOP OF STACK AT     5
  71   -1   7(   -1   66

LHS     5  BLOCKHEAD
RHS        <BEGIN>
SYMBOL  64  <IDENTIFIER>

TOP OF STACK AT     6
  71   -1   7(   -1    5   68

LHS     3  BLOCKBODY
RHS        BLOCKHEAD  <*>
SYMBOL  52  <;>

TOP OF STACK AT     7
  71   -1   7(   -1    3   -1   64

LHS    19  ARRAYID
RHS        <IDENTIFIER>
LHS    18  SIMPLEVAR
RHS        ARRAYID
LHS    20  VARIABLE
RHS        SIMPLEVAR
SYMBOL  64  <IDENTIFIER>

TOP OF STACK AT     8
  71   -1   7(   -1    3   -1    2   52

SYMBOL  67  <[>

TOP OF STACK AT    10
  71   -1   7(   -1    3   -1    2   52   -1   64

LHS    19  ARRAYID
```

```
RHS          <IDENTIFIER>
LHS    18    SIMPLEVAR
RHS          ARRAYID *
LHS    20    VARIABLE
RHS          SIMPLEVAR
LHS    31    PRIMARY
RHS          VARIABLE
LHS    30    SECONDARY
RHS          PRIMARY
LHS    29    FACTOR
RHS          SECONDARY
LHS    28    TERM*
RHS          FACTOR
LHS    27    TERM
RHS          TERM*
LHS    25    EXPO**
RHS          TERM
LHS    24    EXPO*
RHS          EXPO**
LHS    12    EXPRESSION
RHS          EXPO*
LHS    15    SIMPLESTAT
RHS          VARIABLE  <:=>  EXPRESSION
LHS    14    STATEMENT*
RHS          SIMPLESTAT
LHS     4    STATEMENT
RHS          STATEMENT*
SYMBOL  54   <GO>

TOP OF STACK AT      7
  71   -1   71   -1   3    4   67

LHS     3    BLOCKBODY
RHS          BLOCKBODY  STATEMENT  <;>
SYMBOL  53   <TO>

TOP OF STACK AT      7
  71   -1   71   -1   3   -1   54

SYMBOL  64   <IDENTIFIER>

TOP OF STACK AT      8
  71   -1   71   -1   3   -1   54   53

SYMBOL  67   <:>

TOP OF STACK AT      9
  71   -1   71   -1   3   -1   54   53   64

LHS    15    SIMPLESTAT
RHS          <GO>  <TO>  <IDENTIFIER>
LHS    14    STATEMENT*
RHS          SIMPLESTAT
```

```
LHS     4   STATEMENT
RHS         STATEMENT*
SYMBOL  64  <IDENTIFIER>

TOP OF STACK AT      7
  71   -1   7   -1   3   4   67

LHS     3   BLOCKBODY
RHS         BLOCKBODY  STATEMENT  <;>
SYMBOL  52  <;>

TOP OF STACK AT      7
  71   -1   7   -1   3   -2   64

LHS    19   ARRAYID
RHS         <IDENTIFIER>
LHS    18   SIMPLEVAR
RHS         ARRAYID
LHS    21   VARIABLE
RHS         SIMPLEVAR
SYMBOL  64  <IDENTIFIER>

TOP OF STACK AT      8
  71   -1   7   -1   3   -1   2   52

SYMBOL  67  <:>

TOP OF STACK AT      10
  71   -1   7   -1   3   -1   2   52   -1   64

LHS    19   ARRAYID
RHS         <IDENTIFIER>
LHS    18   SIMPLEVAR
RHS         ARRAYID
LHS    21   VARIABLE
RHS         SIMPLEVAR
LHS    31   PRIMARY
RHS         VARIABLE
LHS    30   SECONDARY
RHS         PRIMARY
LHS    29   FACTOR
RHS         SECONDARY
LHS    28   TERM*
RHS         FACTOR
LHS    27   TERM
RHS         TERM*
LHS    25   EXPR**
RHS         TERM
LHS    24   EXPR*
RHS         EXPR**
LHS    12   EXPRESSION
RHS         EXPR*
LHS    15   SIMPLESTAT
```

```
RHS          VARIABLE  <:=>  EXPRESSION
LHS     14   STATEMENT*
RHS          SIMPLESTAT
LHS      4   STATEMENT
RHS          STATEMENT*
SYMBOL   69   <END>

TOP OF STACK AT      7
  71    -1    71    -1     3     4    67
                0

LHS      3   BLOCKBODY
RHS          BLOCKBODY  STATEMENT  <;>
CORRECT EOF
SYMBOL   71   $

TOP OF STACK AT      6
  71    -1    71    -1     3    69

LHS      2   BLOCK
RHS          BLOCKBODY  <END>
LHS      1   PROGRAM
RHS          <START>  BLOCK


END OF PARSE
```

```
PR008
PP004
PR0064
PR0026
PR006[
PR0064
PR0026
PR006
PR0045
PP0055
PP0053
PR0049
PP0048
PR0041
PR004[
PR003
PR003[
PR0022
PP0021
PR006
PR0026
PP0022
PR0021
PP006
PF0064
PR0026
PP006
PP0064
PP0026
PP006[
PP0045
PR0055
PP0053
PP0049
PP004
PR0041
PR004[
PP0038
PR003[
PR0022
PP0021
PP006
PRODUCTION NO 2
PRODUCTION NUMBER 1.
```

```
♦A SIMPLE SENTENCE WITH ONE INPUT ERROR♦
START
      BEGIN♦
        A:=B;
        GOTO L♦P2 ; ♦SHOULD BE GO TO♦
        P:=7 ;
      END
```

```
SYMBOL  71  <START>

TOP OF STACK AT     1
  71

SYMBOL  66  <BEGIN>

TOP OF STACK AT     3
  71   -1   71

SYMBOL  68  <*>

TOP OF STACK AT     5
  71   -1   71   -1   66

LHS     5  BLOCKHEAD
RHS        <BEGIN>
SYMBOL  64  <IDENTIFIER>

TOP OF STACK AT     6
  71   -1   71   -1    5   68

LHS     3  BLOCKBODY
RHS        BLOCKHEAD  <*>
SYMBOL  52  <;>

TOP OF STACK AT     7
  71   -1   71   -1    3   -1   64

LHS    19  ARRAYID
RHS        <IDENTIFIER>
LHS    18  SIMPLEVAR
RHS        ARRAYID
LHS     2  VARIABLE
RHS        SIMPLEVAR
SYMBOL  64  <IDENTIFIER>

TOP OF STACK AT     8
  71   -1   71   -1    3   -1    2   52

SYMBOL  67  <:>

TOP OF STACK AT    10
  71   -1   71   -1   35   -1   21   52   -1   64

LHS    19  ARRAYID
```

```
RHS        <IDENTIFIER>
LHS    18  SIMPLEVAR
RHS        ARRAYID
LHS    20  VARIABLE
RHS        SIMPLEVAR
LHS    31  PRIMARY
RHS        VARIABLE
LHS    30  SECONDARY
RHS        PRIMARY
LHS    29  FACTOR
RHS        SECONDARY
LHS    28  TERM*
RHS        FACTOR
LHS    27  TERM
RHS        TERM*
LHS    25  EXPR**
RHS        TERM
LHS    24  EXPR*
RHS        EXPR**
LHS    12  EXPRESSION
RHS        EXPR*
LHS    15  SIMPLESTAT
RHS        VARIABLE  <:=>  EXPRESSION
LHS    14  STATEMENT*
RHS        SIMPLESTAT
LHS     4  STATEMENT
RHS        STATEMENT
SYMBOL  54  <IDENTIFIER>

TOP OF STACK AT     7
  71   -1    7    -1     3     4    67


LHS     3  BLOCKBODY
RHS        BLOCKBODY  STATEMENT  <;>
SYMBOL  54  <IDENTIFIER>

TOP OF STACK AT     7
  71   -1    7    -1     3    -1    67
```

```
ERROR
ENTER BACKMOVE  0
ENTER CONDENSE

TOP OF STACK AT      7
  71    -1    7     -1     3    -1    64

LHS   19  ARRAYID

TOP OF STACK AT      7
  71    -1    7     -1     3    -1    64

LEAVE CONDENSE
LEAVE BACKMOVE
ESYMBOL  67  <:>

TOP OF STACK AT      9
  71    -1    7     -1     3    -1    64    -1    64

LHS   19  ARRAYID
```

```
ERROR
ENTER BACKMOVE  1
ENTER CONDENSE


TOP OF STACK AT      9
 71   -1   71   -1    3   -1    64    64

LHS   -1

TOP OF STACK AT      8
 71   -1   71   -1    3   -1    64    64

LEAVE CONDENSE
LEAVE BACKMOVE
```

```
CANDIDATES AT
BOTTOM      7
QUEST1      7
STACK       8
```

CALLS TO REDUCE FOR RHODES LEFT STACKABILITY

```
LHS     3   BLOCKBODY
LHS    19   ARRAYID
LHS
LHS
LHS     3   BLOCKBODY
LHS     3   BLOCKBODY
LHS    19   ARRAYID
LHS    19   ARRAYID
```

REPLACEMENT SETS

NT	CANDIDATE 1	2	3
0	T	T	T
1	F	F	F
2	T	F	F
3	T	F	F
4	T	F	F
5	T	F	F
6	T	F	F
7	F	F	F
8	F	F	F
9	F	F	F
10	F	F	F
11	F	F	F
12	F	F	F
13	T	F	F
14	T	F	F
15	T	F	F
16	F	F	F
17	F	F	F
18	T	F	F
19	T	F	F
20	T	F	F
21	F	F	F
22	F	F	F
23	F	F	F
24	F	F	F
25	F	F	F
26	F	F	F
27	F	F	F
28	F	F	F
29	F	F	F
30	F	F	F
31	F	F	F
32	F	F	F
33	F	T	F

NON TERMINAL NUMBER 0 IS DELETION

COST OF LSC PHRASES AND THE EMPTY WORD

CANDIDATE	NT	COST
1	0	10
1	2	26
1	2	29
1	3	26
1	3	21
1	3	22
1	3	19
1	4	12
1	5	16
1	5	19
1	5	22
1	6	7
1	14	13
1	14	15
1	14	18
1	15	9
1	15	13
1	15	19
1	15	21
1	15	13
1	16	13
1	[illegible]	5
1	[illegible]	13
1	[illegible]	21
1	22	13
2	[illegible]	5
2	[illegible]	21
2	[illegible]	16
2	3	17
2	3	14
2	6	2
2	16	18
2	21	14
2	21	15
2	33	14
2	33	15
3	0	5

```
CHOSEN MEMBER
  COST      2
  SYMBOL    5
  CAND      2
CANDIDATE    <IDENTIFIER>
REPLACEMENT  LABELDEF

TOP OF STACK AT      8
  71   -1   72   -1    3   -1   64   64


TOP OF STACK AT      8
  71   -1   72   -1    3   -1   64   64


TOP OF STACK AT      7
  71   -1   72   -1    3    6   64


TOP OF STACK AT      7
  71   -1   72   -1    3    6   64

RHODES RECOVERY OVER, PARSING CONTINUES.
```

```
TOP OF STACK AT      7
  71   -1   72   -1    3    6   64

LHS   -1
```

```
ERROR
ENTER BACKMOVE  0
ENTER CONDENSE

TOP OF STACK AT      7
  71    -1    7?    -1     3     6    64

LHS    -1

TOP OF STACK AT      7
  71    -1    7?    -1     3     6    64

LEAVE CONDENSE
ENTER CONDENSE

TOP OF STACK AT      7
  71    -1    7?    -1     3     6    64

LHS     3  BLOCKBODY
RHS        BLOCKBODY  LABELDEF

TOP OF STACK AT      7
  71    -1    7?    -1     3     6    64

TOP OF STACK AT      7
  71    -1    7?    -1     3    -1    64

TOP OF STACK AT      7
  71    -1    7?    -1     3    -1    64

LEAVE CONDENSE
ENTER CONDENSE

TOP OF STACK AT      7
  71    -1    7?    -1     3    -1    64

LHS    19  ARRAYID
RHS        <IDENTIFIER>
LHS    14  SIMPLEVAR
RHS        ARRAYID
LHS    21  VARIABLE
RHS        SIMPLEVAR
LHS    31  PRIMARY

TOP OF STACK AT      7
  71    -1    7?    -1     3    -1     2

LEAVE CONDENSE
LEAVE BACKMOVE
ESYMBOL  64  <IDENTIFIER>

TOP OF STACK AT      9
  71    -1    7?    -1     3    -1     2    -1    67

LHS    -1
```

```
ERROR
ENTER BACKMOVE  1
ENTER CONDENSE

TOP OF STACK AT      8
  71   -1   7[   -1    3   -1   2[   67

LHS   -1

TOP OF STACK AT      8
  71   -1   7[   -1    3   -1   2[   67

LEAVE CONDENSE
ENTER CONDENSE

TOP OF STACK AT      8
  71   -1   7[   -1    3   -1   2    67

LHS   31  PRIMARY

TOP OF STACK AT      8
  71   -1   7[   -1    3   -1   2    67

LEAVE CONDENSE
LEAVE BACKMOVE
```

```
CANDIDATES AT
BOTTOM     7
QUEST1     7
STACK      8
```

```
CALLS TO REDUCE FOR RHODES LEFT STACKABILITY
LHS     3  BLOCKBODY
LHS     1
LHS     1
LHS     3  BLOCKBODY
LHS     3  BLOCKBODY
LHS    31  PRIMARY
```

REPLACEMENT SETS

NT	CANDIDATE 1	2	3
0	T	T	F
1	F	F	F
2	F	T	F
3	T	T	F
4	F	T	F
5	T	T	F
6	F	F	F
7	F	F	F
8	F	F	F
9	F	F	F
10	F	F	F
11	F	F	F
12	F	F	F
13	F	T	F
14	F	T	F
15	T	F	F
16	F	T	F
17	F	T	F
18	F	T	F
19	F	T	F
20	T	F	F
21	F	F	F
22	F	F	F
23	F	F	F
24	F	F	F
25	F	F	T
26	F	F	F
27	F	F	F
28	F	F	F
29	F	F	F
30	F	F	F
31	T	F	F

NON TERMINAL NUMBER 6 IS DELETION

COST OF LSC PHRASES AND THE EMPTY WORD

CANDIDATE	NT	COST
1	[illegible]	102
1	[illegible]	118
1	[illegible]	134
1	[illegible]	141
1	[illegible]	157
1	[illegible]	115
1	[illegible]	114
1	[illegible]	112
1	33	112
2	[illegible]	99
2	[illegible]	92
2	[illegible]	56
2	[illegible]	52
2	[illegible]	95
2	[illegible]	82
2	[illegible]	66
2	[illegible]	55
2	[illegible]	46
2	[illegible]	36
2	[illegible]	26
2	[illegible]	16
2	[illegible]	6
2	[illegible]	2
2	[illegible]	2
2	[illegible]	2
2	26	2

```
CHOSEN MEMBER
  COST      3
  SYMBOL    6
  CAND      2
CANDIDATE    VARIABLE
DELETED

TOP OF STACK AT      8
  71    -1    70    -1     3    -1    20    67


TOP OF STACK AT      5
  71    -1    70    -1     3    67

RHODES RECOVERY OVER, PARSING CONTINUES.


TOP OF STACK AT      6
  71    -1    70    -1     3    67

LHS     7  BLOCKBODY
RHS        BLOCKBODY  <;>
SYMBOL  52  <END>

TOP OF STACK AT      7
  71    -1    70    -1     3    -1    64

LHS    19  ARRAYID
RHS        <IDENTIFIER>
LHS    18  SIMPLEVAR
RHS        ARRAYID
LHS    20  VARIABLE
RHS        SIMPLEVAR
SYMBOL  64  <IDENTIFIER>

TOP OF STACK AT      8
  71    -1    70    -1     3    -1    20    52
```

```
SYMBOL  67  <;>

TOP OF STACK AT    10
  71   -1   70   -1    3   -1   23   52   -1   64

LHS    19  ARRAYID
RHS        <IDENTIFIER>
LHS    18  SIMPLEVAR
RHS        ARRAYID
LHS    2?  VARIABLE
RHS        SIMPLEVAR
LHS    3?  PRIMARY
RHS        VARIABLE
LHS    3?  SECONDARY
RHS        PRIMARY
LHS    29  FACTOR
RHS        SECONDARY
LHS    28  TERM*
RHS        FACTOR
LHS    27  TERM
RHS        TERM*
LHS    25  EXPR**
RHS        TERM
LHS    24  EXPR*
RHS        EXPR**
LHS    12  EXPRESSION
RHS        EXPR*
LHS    15  SIMPLESTAT
RHS        VARIABLE  <:=>  EXPRESSION
LHS    14  STATEMENT*


RHS        SIMPLESTAT
LHS     4  STATEMENT
RHS        STATEMENT*
SYMBOL  69  <END>

TOP OF STACK AT     7
  71   -1   70   -1    3    4   67

LHS     3  BLOCKBODY
RHS        BLOCKBODY  STATEMENT  <;>
CORRECT EOF
SYMBOL  71  $

TOP OF STACK AT     6
  71   -1   70   -1    3   69

LHS     2  BLOCK
RHS        BLOCKBODY  <END>
LHS     1  PROGRAM
RHS        <START>  BLOCK


END OF PASS
```

```
     PROD8
     PROD4
     PRODD64
     PR0026
     PR006[
     PR0064
     PR0026
     PR006
     PR0045
     PR0055
     PR0053
     PR0049
     PR0048
     PR0041
     RR0040
     PR0038
     PR0036
     PR0022
     PR0021
     PR006
R    PR0020
C    PR005
C    PR0064
C    PR0026
C    PR006
O
     PR007
     PR0064
     PR0026
     PR006
     PR0064
     PR0026
     PR0060
     PR0045
     PR0055
     PR0053
     PR0049
     PR0048
     PR0041
     PR0040
     PR0038
     PR0030
     PR0022
     PR0021
     PR006
     PRODUCTION NO 2
     PRODUCTION NUMBER 1
```

```
 06/26/74 MCMASTER - SCOPE 3.4 AT L3-9    03/22/74
21.19.10.
21.19.10.HAWCJJ9                                  HAWCJJ9
21.19.10.IP  00000192 WORDS - FILE INPUT  , DC 00
21.19.10.HAWCJ,CM50000,T30.       PARSER
21.19.10.                                  JOHNS,C.R.
21.19.10.DISPOSE(OUTPUT,*P2)
21.19.10.EOJ DISPOSITION SPECIFIED, KY IGNORED
21.19.11.ATTACH(CONPAR,ID=HAWCJ,CY=1)
21.19.11.PFN IS
21.19.11.CONPAR
21.19.11.COPYBR(CONPAR,SEDEFLE)
21.19.12.COPYBR(CONPAR,ROOTFLE)
21.19.13.COPYBR(CONPAR,NODEFLE)
21.19.14.REWIND(SEDEFLE)
21.19.14.REWIND(ROOTFLE)
21.19.14.REWIND(NODEFLE)
21.19.14.COPYCR(INPUT,SOURCE)
21.19.15.REWIND(SOURCE)
21.19.15.COPYSBF(SOURCE,OUTPUT)
21.19.15.REWIND(SOURCE)
21.19.15.ATTACH(PARSER,CO-PAID,ID=HAWCJ)
21.19.16.PF CYCLE NO. = 006
21.19.16.ATTACH(PASCAL,ID=GSOPASCAL,MR=1)
21.19.16.PFN IS
21.19.16.PASCAL
21.19.16.PF CYCLE NO. = 002
21.19.16.PASCAL(R=PARSER,LL=1...)
21.19.19.- PASCALSYSTEM VERS.  -/16/73
21.19.51.- END COMPILATION
21.19.53.- END PASCAL
21.19.53.REWIND(SEMAN)
21.19.53.COPYSF(SEMAN,OUTPUT)
21.19.53.EOF/EOI ENCOUNTERED
21.19.53.OP  00016142 WORDS - FILE OUTPUT , DC 42
21.19.53.CP      115.685 SEC.
21.19.53.PP      106.466 SEC.
21.19.53.IO      031.187 SEC.
21.19.53.IOM     242.909 WS/512
21.19.53.CPM     665.417 WS/512
21.19.53.ENDJOB 06/26/74

************      HAWCJJ9  //// END OF LIST ////
************      HAWCJJ9  //// END OF LIST ////
```

APPENDIX E

DETAILED FLOWCHART OF BASIC SIMPLE PRECEDENCE PARSING ALGORITHM

The flowchart is developed from Leinius (1970, p. 44).

Notation

i) The parser has a configuration of:
($X_1 \ldots X_m$, $a_1 \ldots a_n$\$, $p_1 \ldots p_n$)
See section 2.4.2 for details.

ii) X_j is the jth element in the stack

iii) The top of the stack is at i

iv) a_{n+1} = \$

v) The current input symbol is stored in I

vi) If $X_m \doteq X_{m+1}$ then Δ is inserted between X_m and X_{m+1} (in our implementation Δ is -1).

Procedures and Functions

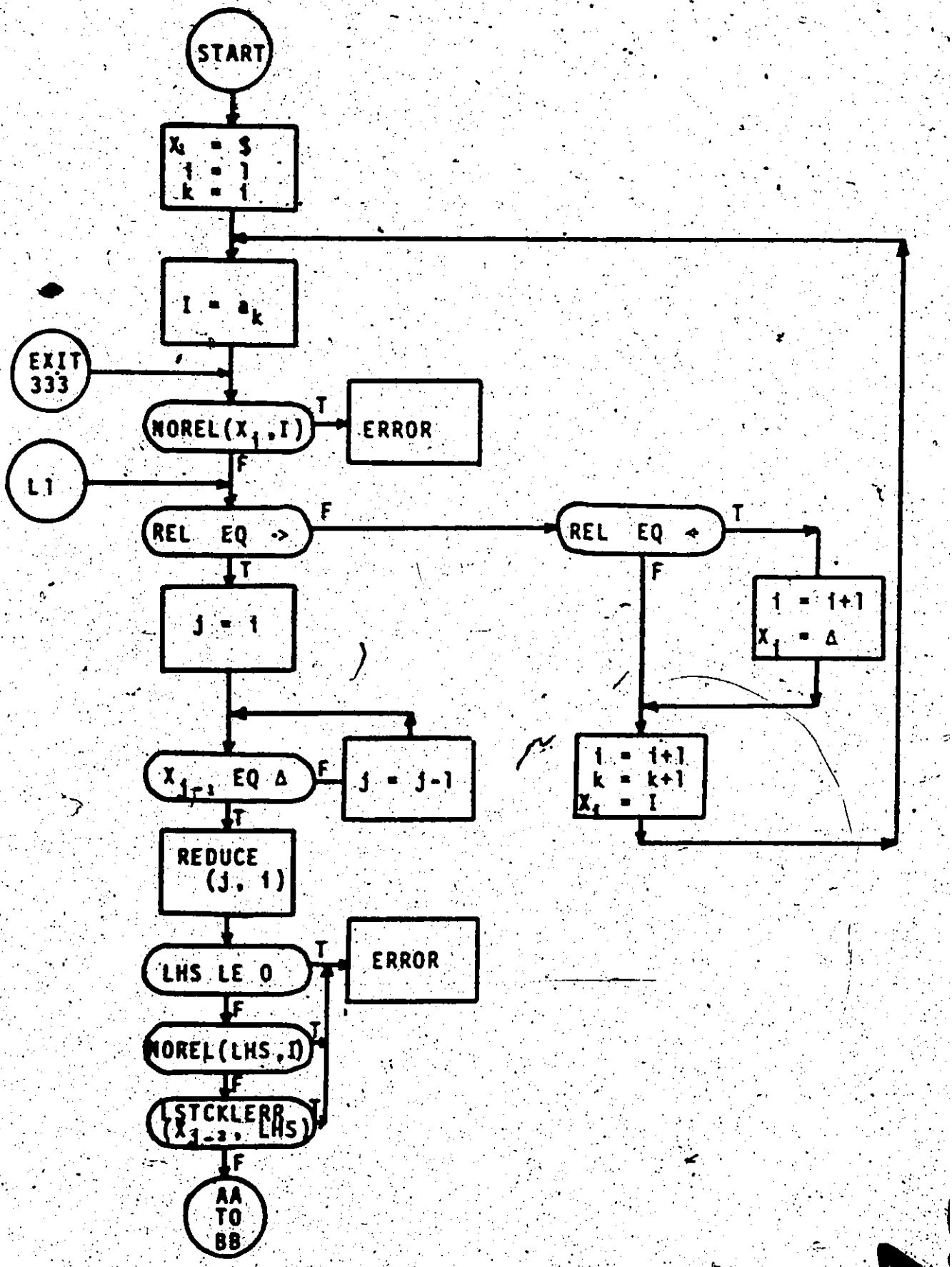
i) The procedure REDUCE (j,i) sets LHS to the nonterminal symbol corresponding to the reduction of $X_j \ldots X_i$, or 0 if $X_j \ldots X_i$ is a prefix of the RHS of a rule, or -1 if $X_j \ldots X_i$ is neither a RHS or a prefix of a RHS.

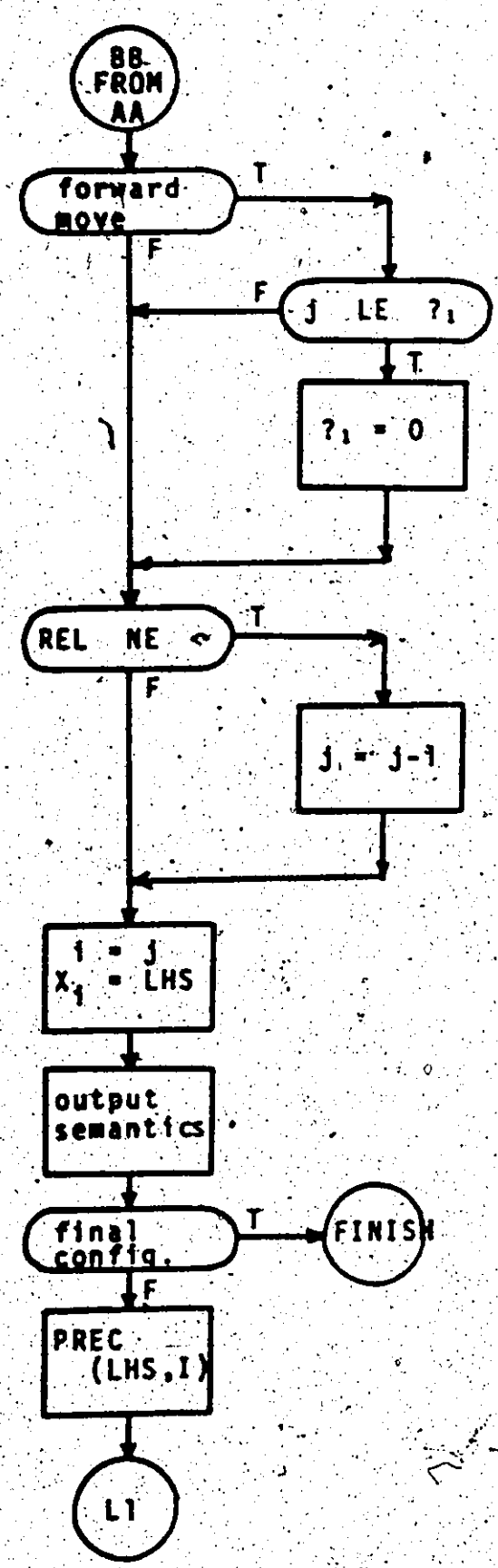
ii) The procedure PREC (X_m,X_n) sets REL to the precedence relation between X_m and X_n. If there is no relation, REL = blank.

iii) The boolean function NOREL (X_m, X_n) is TRUE iff no relation exists between X_m and X_n. If there is a relation, it is assigned to REL.

iv) The boolean function LSTEKLERR (X_m, X_n) is TRUE iff X_n is not Leinius left stackable against X_m. REL is set to the relation between X_m and X_n.

v) If an error is detected, the procedure ERROR is entered.


START
$X_0 = \$$
$i = 1$
$k = 1$
$I = a_k$
EXIT 333
MOREL(X_{i-1}, I)
T
ERROR
F
L1
REL EQ ->
F
REL EQ <-
T
T
F
$j = i$
$i = i+1$
$X_i = \Delta$
X_{j-1} EQ Δ
F
$j = j-1$
$i = i+1$
$k = k+1$
$X_i = I$
T
REDUCE (j, 1)
LHS LE 0
T
ERROR
F
MOREL(LHS, I)
T
F
LSTCKLERR (X_{j-2}, LHS)
T
F
AA TO BB

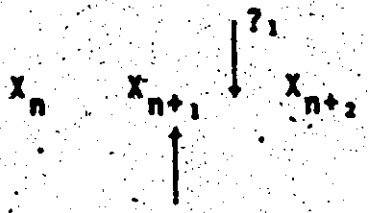

BB FROM AA
forward moves
T
F
F
j LE ?1
T
?1 = 0
REL NE ~
T
F
j = j-1
i = j
X1 = LHS
output semantics
final config.
T
FINISH
F
PREC (LHS,I)
L1

APPENDIX F

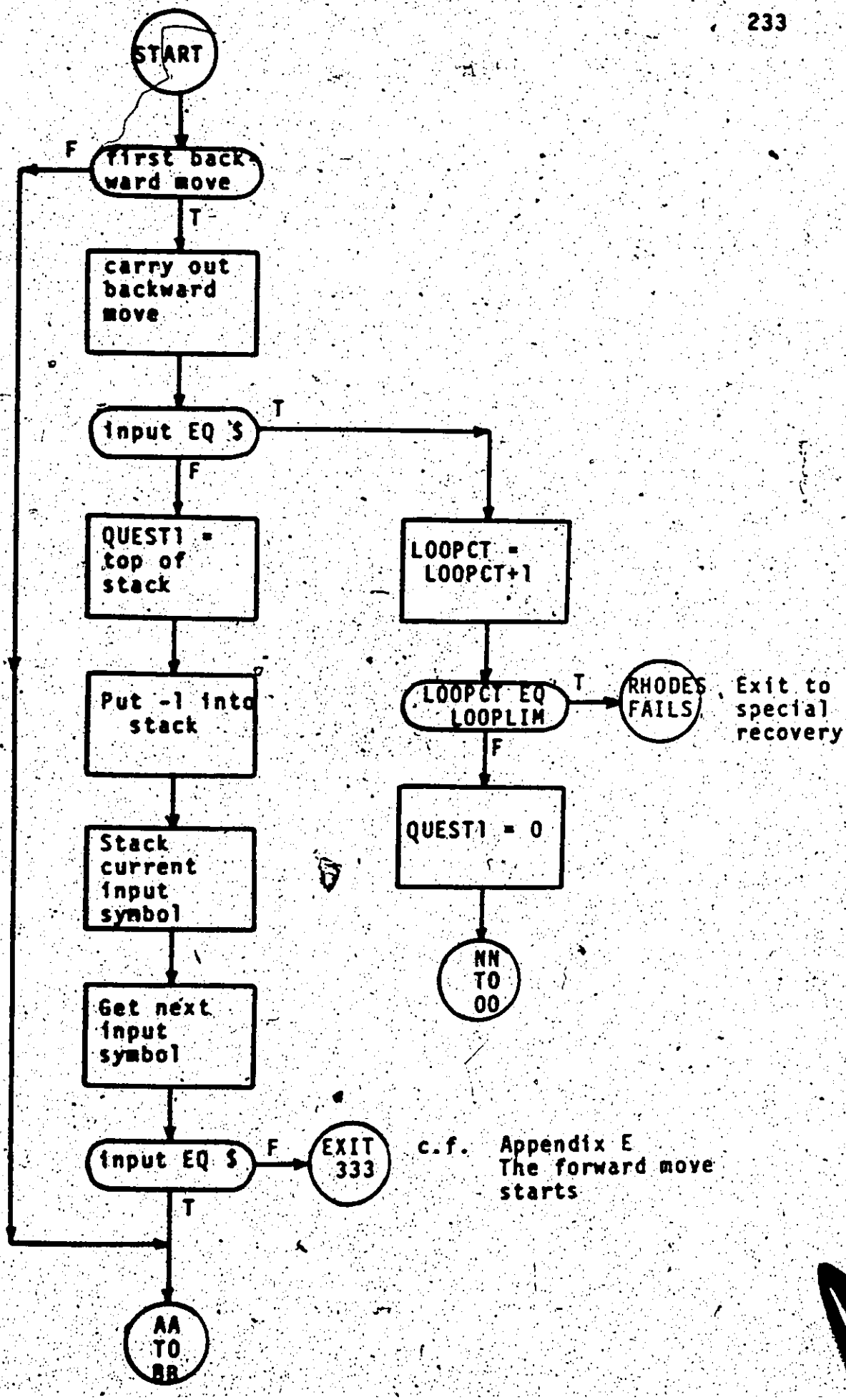
SYSTEM FLOWCHART FOR ERROR RECOVERY

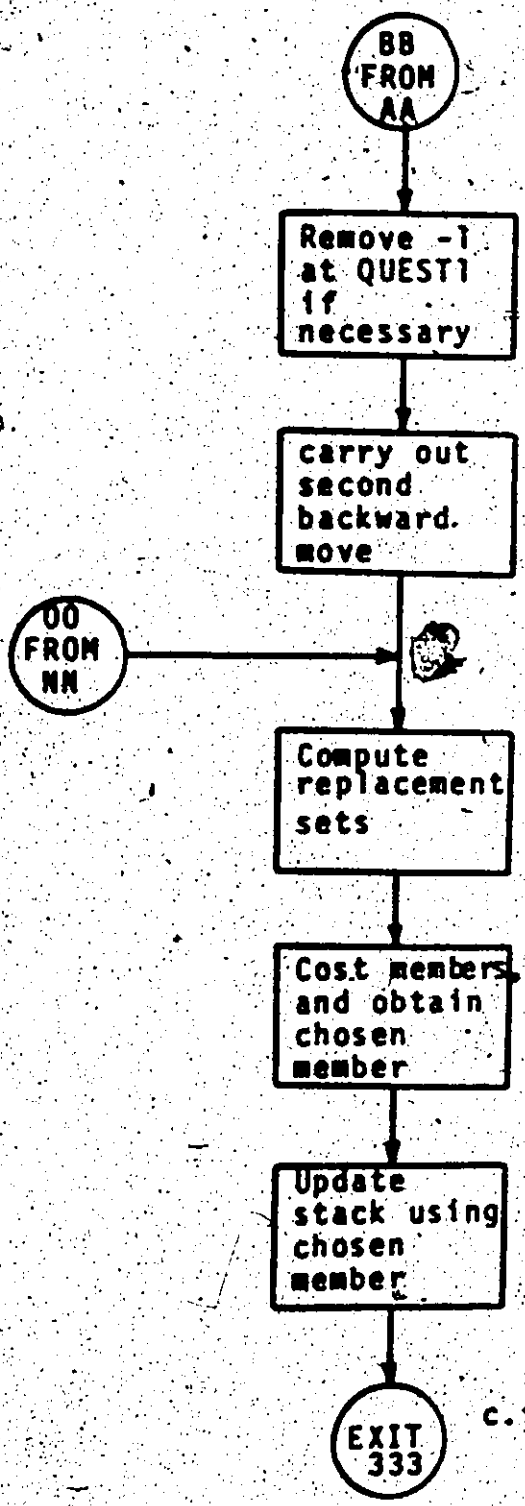
This is the procedure ERROR mentioned in Appendix E. ERROR incorporates a check for looping in the error recovery. LOOPCT is used to count the number of times the error recovery is attempted when the input symbol is $. If LOOPCT is equal to LOOPLIM, Rhodes recovery has failed, and special error recovery is required. LOOPCT is initialised to 0.

Note the meaning of QUEST1:



QUEST1 is a stack position.


START
first backward move
F
T
carry out backward move
input EQ $
T
F
QUEST1 = top of stack
Put -1 into stack
Stack current input symbol
Get next input symbol
input EQ $
F
EXIT 333
T
AA TO BB
LOOPCT = LOOPCT+1
LOOPCT EQ LOOPLIM
T
RHODES FAILS
Exit to special recovery
F
QUEST1 = 0
NN TO OO
c.f. Appendix E The forward move starts



c.f. Appendix E
Recovery is over

APPENDIX G

PARSER OUTPUT -- Rhodes 1

```
*RHODES 1*
START
      BEGIN
        INTEGER AA,BB,CC ;
        *RHODES P4A*
        INTEGER ARRAY A(1..N+2,3..8) ;
        INTEGER ARRAY A1,A2,A3(1..4,1..5) ;
        ;
        AA := 2 ;
        A1(2,3) := A1(7,8) + AA*[illegible]*CC ;
        *RHODES P5A*
        IF I >= 4
          THEN
            GO TO L ;
        BB := 3 ;
        A1(2,3) := A1(7,8) + AA*[illegible]*CC ;
        *RHODES P47*
        IF I = 1
          THEN
            C = 0
        GO TO LAB1 ;
      END
```

Forward Move
Force ends after second backward move

(1)

First input error

First derived error from first input error

Second derived error from first input error, and second input error

(2)

Third input error

Fourth input error

First derived error from third input error

(4)

Second derived term from third input error

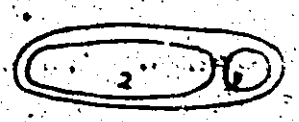
APPENDIX H

PARSER OUTPUT -- Rhodes 2

```
@RHODES 24
START
      BEGIN
        @RHODES P53+
        INTEGER ARRAY FAR(1..25;
        INTEGER AA,BB,CC;
        .
        A1(2,3) := A1(7,8) + AA*BB*CC ;
        CC := 0 ;
        @RHODES P53+
          (I<1) ∨ (I>1.)
          THEN
              C := 2*I
          ELSE
              GO TO OUT ;
        A1(2,3) := A1(7,8) + AA*BB*CC ;
      END
```

FAIL - EMPTY SET

First input error



(1)

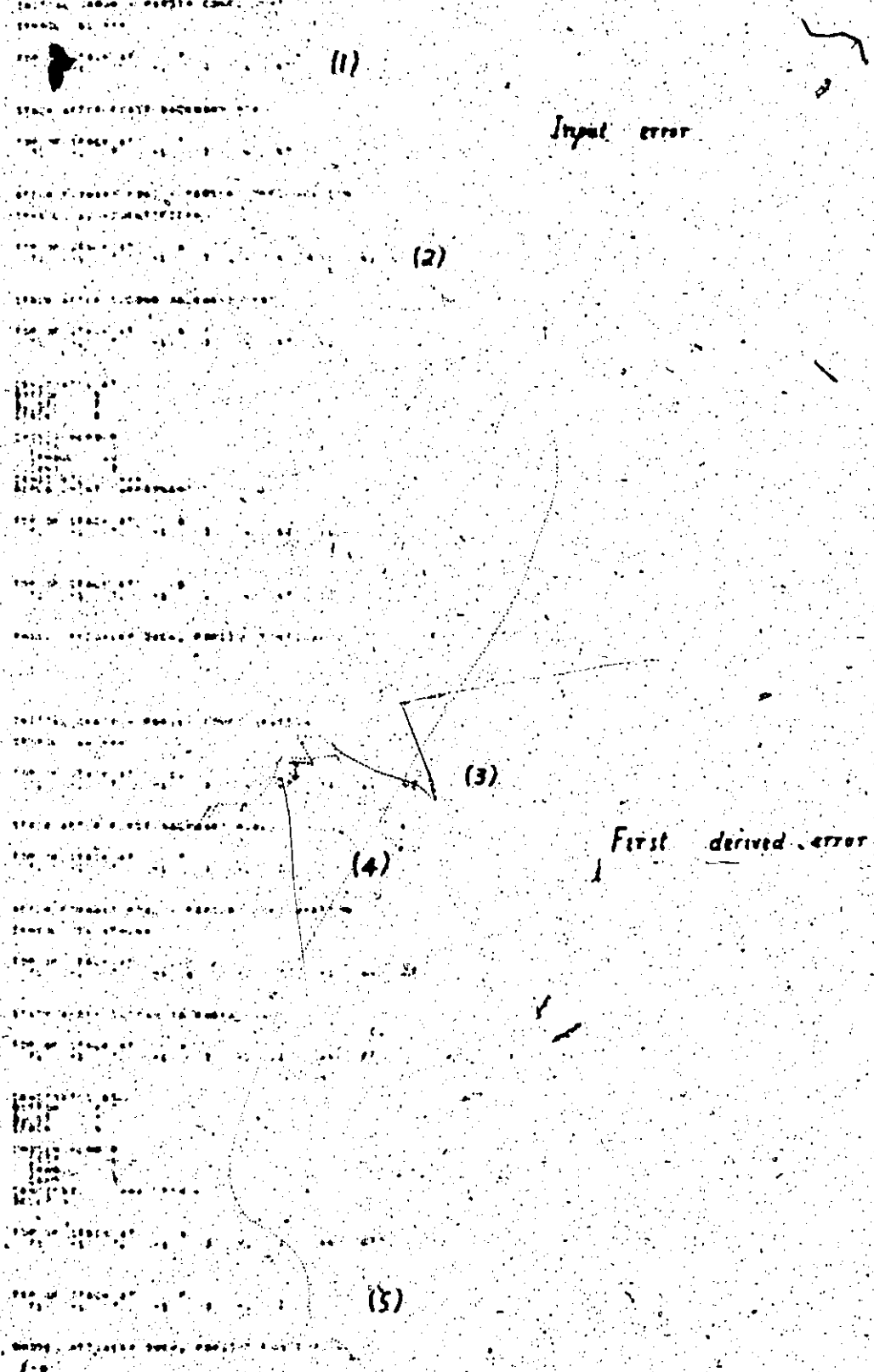
First derived error from first input error, and second input error

APPENDIX I

PARSER OUTPUT -- Rhodes 2A

```
#RHODES 24+
START
     BEGIN+
      CC := [illegible] ;
      #RHODES P53+
        (I<1) v (I>1.)
       THEN
           C := 2*I
       ELSE
           GO TO OUT ;
      A1(2,3) := A1(7,8) + AA+B *CC ;
     END
```

Forward move


(1)
Input error
(2)
(3)
First derived error
(4)
(5)

Second derived error

Third derived error

Fourth derived error

(6)

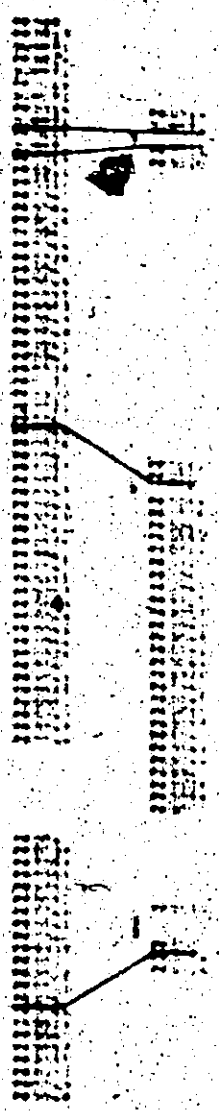
Fifth derived error

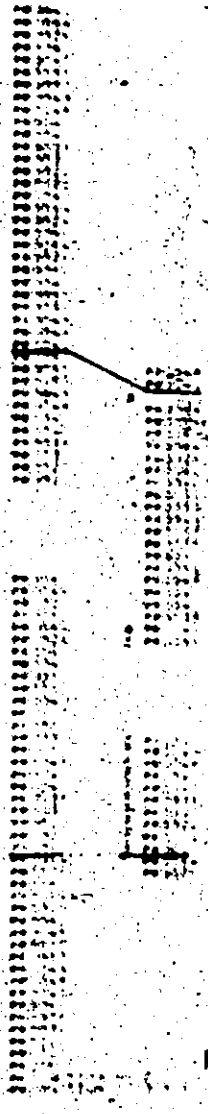
Sixth ,, derived error

APPENDIX J

PARSER OUTPUT -- Rhodes 3

```
*RHODES 3*
START
    BEGIN
      *RHODES P=3*
      INTEGER TOP1,BOT1,TOP2,BOT2,CODE PES ;
      *
      P := 3 ;
      *RHODES P=4*
      A := 3*(J+K) ;
      Z := 4 + J ;
      *RHODES P57*
      I := J-4
      K := K/2 ;
      P := 3 ;
    END
```




Forward Move

(1)

First input error

(2)

Second input error

First derived error from second input error

Second derived error from second input error

Third input error

(3)

(4)

First derived error from third input error

(5)

Second derivid error from
third input error

APPENDIX K

PARSER OUTPUT -- Rhodes 4

```
*RHODES **
START
      BEGIN*
       *RHODES P65*
       IF (I=2) * (J=3.)
         THEN
         THEN
              GO TO L ;
       A := B ;
      END
```

(1)

APPENDIX L

PARSER OUTPUT -- Systems Test

```
*SYSTEMS TEST*
START
      BEGIN*
       *ERROR WITHIN NESTED BEGIN BLOCK*
       IF A=5
         THEN
              BEGIN*
                   P=7;
                   P=S;
              END;
       A := 3 ;
       *; BEFORE ELSE*
       IF A = 6
         THEN
              A := A + 3 ;
         ELSE
              A := A/2 ;
       A := 1 ;
       *END MISSING FOR NESTED BEGIN BLOCK*
       IF A = 7
         THEN
              BEGIN*
                   B := 0 ;
                   Z := Z + 1 ;
                   IF Z = 6
                     THEN
                          BEGIN*
                               Z :=   ;
                               P := B + ;
                          *END MISSING*
              END ;
       A := 0 ;
       *GO TO AS GOTO*
       GOTO LAB1 ;
LAB1 ;
       R := 1 ;
      END
```

LOOPING

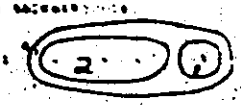
First input error

Derived error from first input error

Second input error

Derived error from second input error

Third input error



(1)

Fifth input error

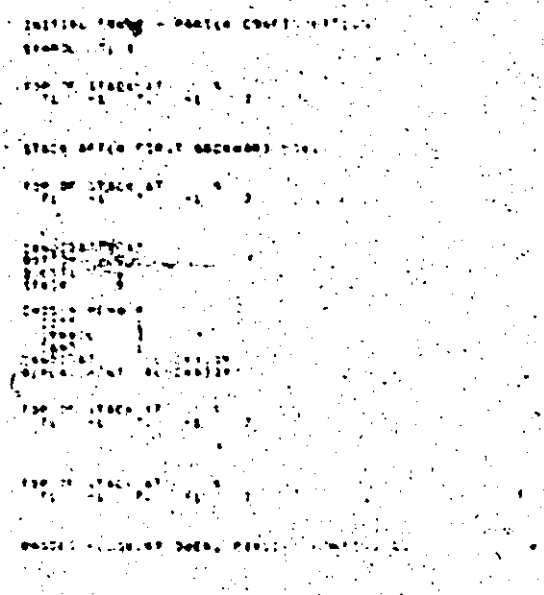
(2)

Derived error from fifth input error.

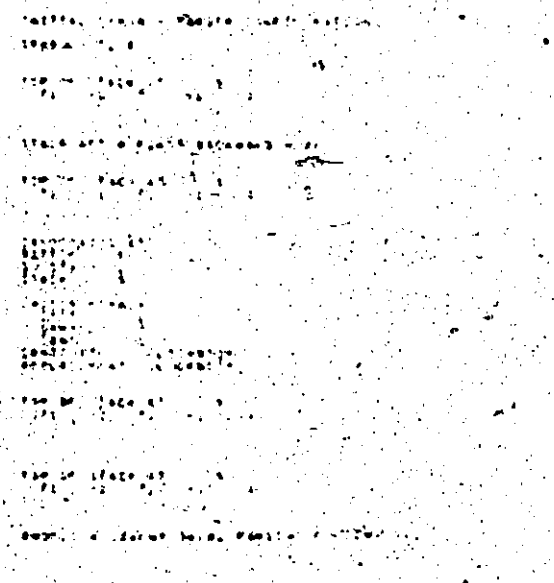
(3)

Fourth input error

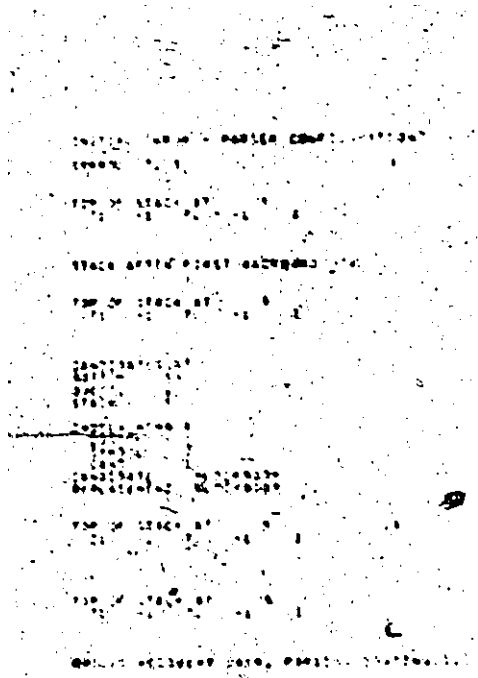
(4)



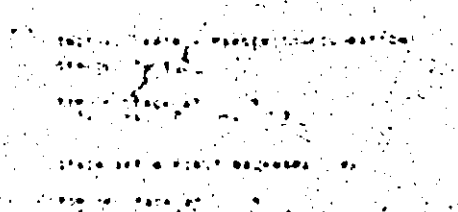
First derived error from fourth input error



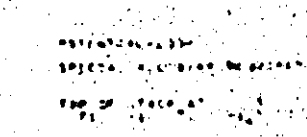
Second derived error from fourth input error



Third derived error from
fourth input error



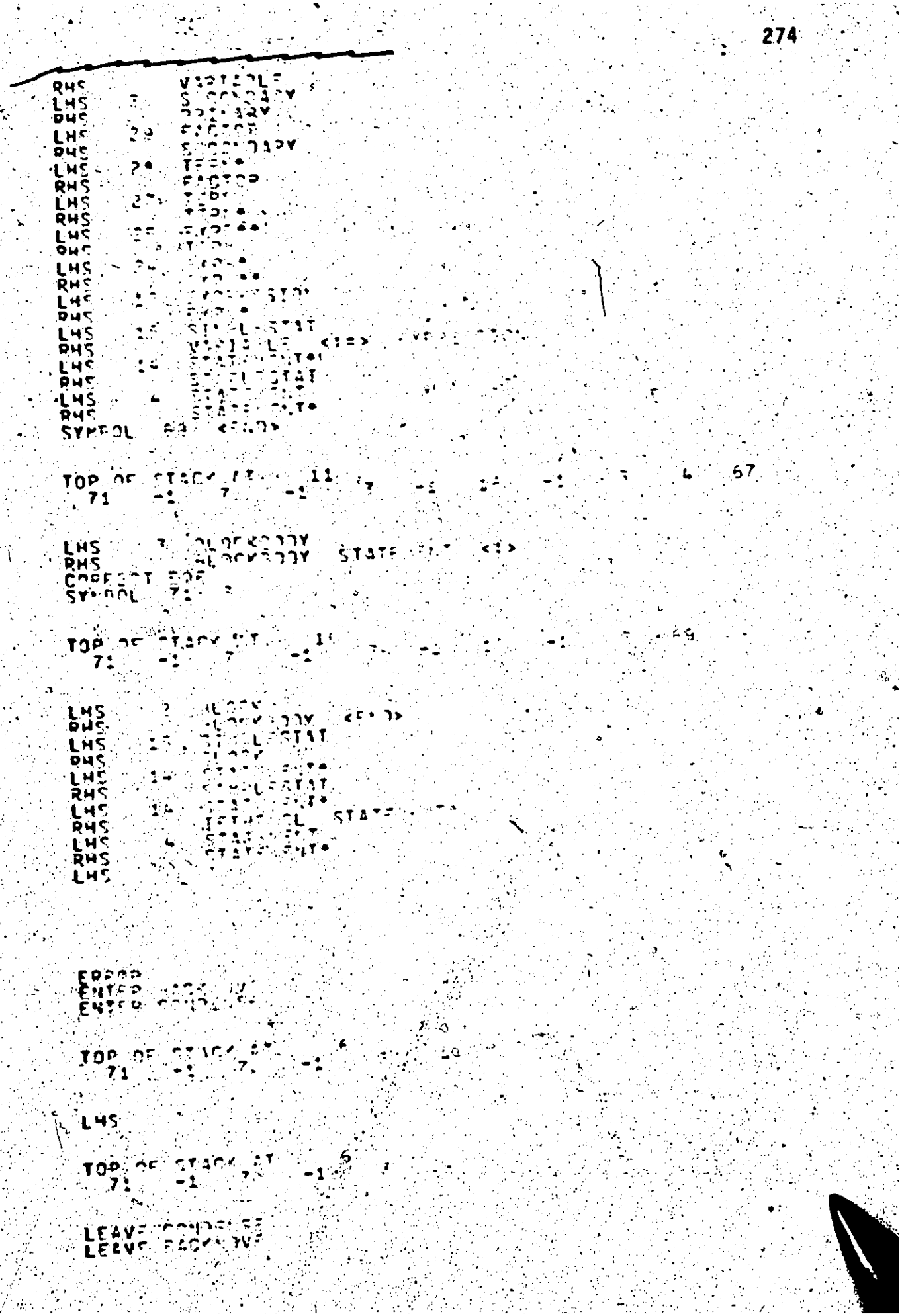
Fourth derived error from
fourth input error



LOOPING detected

APPENDIX M

ADDITIONAL PARSER OUTPUT FOR SYSTEMS TEST



CANDIDATES ST
BOTTO[illegible] 5
QUESTS
STACK 6

CALLS TO [illegible] FOR [illegible]

LHS
LHS [illegible] PROGRAM

REPLACEMENT SETS

AT	CANDIDATE 1	2	3
1	F	F	F
2	F	F	F
3	T	F	F
4	T	F	F
5	F	F	F
6	F	F	F
7	T	F	F
8	F	F	F
9	F	F	F
10	F	F	F
11	F	F	F
12	F	F	F
13	F	F	F
14	F	F	F
15	F	F	F
16	F	F	F
17	F	F	F
18	F	F	F
19	F	F	F
20	F	F	F
21	F	F	F
22	F	F	F
23	F	F	F
24	F	F	F
25	F	F	F
26	F	F	F
27	F	F	F
28	F	F	F
29	F	F	F
30	F	F	F
31	F	F	F
32	F	F	F
33	F	F	F
34	F	F	F
35	F	F	F

NON TERMINAL NUMBER [illegible]

COST OF LSC PHRASES AND THE EMPTY WORD 6

CANDIDATE AT COST

1	2	12
1	3	8
1	4	14
1	5	7
1	6	12
1	7	5
1	8	109
1	9	7
1	1	10

CHOSEN PHRASE
COST
SYMBOL
CAND
CANDIDATE LOCKBODY STATEMENT
REPLACEMENT LOCKBODY

Fourth input error

TOP OF STACK AT 6
72 -1 7 -1 3 4

TOP OF STACK AT 5
72 -1 7 -1 3

PHRASE RECOVERY OVER, PARSING CONTINUES.

TOP OF STACK AT 5
72 -1 7 -1 7

LHS

7

ERROR 4
ENTER BACKTRACK
ENTER CONTINUE

TOP OF STACK AT 5
72 -1 7 -1 7

LHS

TOP OF STACK AT 5
72 -1 7 -1

LEAVE CONTINUE
LEAVE BACKTRACK

CANDIDATES AT
BOTTOM
QUEST:
STACK 5

CALLS TO REDUCE FOR PHRASE LEFT STACKABILITY

LHS
LHS : PROGRAM

REPLACEMENT SETS

NT	CANDIDATE 1	2	3
1	F	F	F
2	F	F	F
3	T	F	F
4	F	F	F
5	F	F	F
6	T	F	F
7	F	F	F
8	F	F	F
9	F	F	F
10	F	F	F
11	F	F	F
12	F	F	F
13	F	F	F
14	F	F	F
15	F	F	F
16	F	F	F
17	F	F	F
18	F	F	F
19	F	F	F
20	F	F	F
21	F	F	F
22	F	F	F
23	F	F	F
24	F	F	F
25	F	F	F
26	F	F	F
27	F	F	F
28	F	F	F
29	F	F	F
30	F	F	F
31	F	F	F
32	F	F	F
33	F	[illegible]	[illegible]

NON TERMINAL NUMBER [illegible] IS DELETED

```
COST OF LSC PHRASES AND THE EMPTY WORD

CANDIDATE    NT    COST
                      9
   1                 11
   1          3      1
                      3
                      4
   1                  1
                      5
                      3
                      3

CHOSEN  ERROR
  COST
  SYMBOL
  CAND
CANDIDATE      LOCKBODY
REPLACEMENT    BLOCKBODY

TOP OF STACK AT      5
  71    -1    7    -1    3

TOP OF STACK AT      5
  71    -1    7    -1    3

RHODES RECOVERY OVER, PARSING CONT...

TOP OF STACK AT      5
  71    -1    7    -1    3

LHS

ERROR
ENTER BACKDOWN
ENTER CONDENSE

TOP OF STACK AT      5
  71    -1    7    -1

LHS

TOP OF STACK AT      5
  71    -1    7    -1    3

LEAVE CONDENSE
LEAVE BACKDOWN
```

First derived error from fourth input error.

No change – LOOPING

etc.

REFERENCES

Aho, A. V., Denning, P. J., and Ullman, J. D. "Weak and Mixed Strategy Precedence Parsing", J.ACM, 19, 2 (1972), 225-243.

Aho, A. V., and Peterson, T. G. "A Minimum Distance Error Correcting Parser for Context Free Languages", Siam Journal of Computing, 1, 4 (1972), 305-312.

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

--------. "Linear Precedence Functions for Weak Precedence Grammars", International Journal of Computer Mathematics, Section A, 3 (1972), 149-155.

--------. "Error Detection in Precedence Parsers", Mathematical Systems Theory, 7, 2 (1973), 97-113.

Barnes, K. R. "Exploratory Steps Towards a Grammatical Manipulation Package". Computer Science Technical Report 73/5, McMaster University, 1973.

Bell, J. R. "A New Method for Determining Linear Precedence Functions for Precedence Grammars", C.ACM, 12, 10 (1969), 567-569.

Brooker, R. A., Morris, D., and Rohl, J. S. "Experience with the Compiler Compiler", Computer Journal, 9, 4 (1967), 345-349.

Conway, M. E. "Design of Separable Transition Diagram Compiler", C.ACM, 6, 7 (1963), 396-408.

Damerau, F. "A Technique for Computer Detection and Correction of Spelling Errors", C.ACM, 7, 3 (1964), 171-176.

Dean, A. L., Jr. Some Results in the Area of Syntax Directed Compilers. Wakefield, Mass.: Computer Associates, Inc., 1964.

Earley, J. "An Efficient Context Free Parsing Algorithm", C.ACM, 13, 2 (1970), 94-102.

Eggers, B. "Error Reporting, Error Treatment and Error Correction in ALGOL Translation", part II, 2nd Annual Meeting G. I. Karlsruhr (October 1972).

Elspas, B., Green, M. W., and Levitt, K. N. "Software Reliability", Computer, 4, 1 (1971), 21-27.

Freeman, D. N. "Error Correction in CORC -- the Cornell Computing Language". Ph.D. dissertation, Cornell University, 1963.

Graham, S. L. "Precedence Languages and Bounded Right Context Languages". Computer Science Department, Stanford University, Stan-CS-71-223, 1971.

Gries, D. Compiler Construction for Digital Computers. New York and London: John Wiley and Son, 1971.

Griffiths, T. V., and Petrick, S. R. "On the Relative Efficiencies of Context Free Grammar Recognisers", C.ACM, 8, 5 (1965), 289-300.

Hedrick, G. E. "User Error Analysis and Automatic Correction for Compiling". Ph.D. dissertation, Iowa State University, 1970.

Hext, Jan. B. "Recovery from Error", Computers and Automation, 16, 4 (1967), 29-31.

Hopcroft, J. E. and Ullman, J. D. "Error Correction for Formal Languages". Digital Systems Lab. Report 52, Princeton, N.J., Princeton University, 1966.

Irons, E. T. "An Error Correcting Parse Algorithm", C.ACM, 6, 11 (1963), 669-673.

James, E. G., and Partridge, D. P. "Adaptive Correction of Program Statements", C.ACM, 16, 1 (1973), 27-37.

James, L. R. "A Syntax Directed Error Recovery Method". Technical Report CSRG-13, Computer Systems Research Group, University of Toronto, 1972.

LaFrance, J. E. "Optimisation of Error Recovery in Syntax Directed Parsing Algorithms", SIGPLAN, notice number 5 (1970), 2-17.

--------. "Syntax Directed Recovery for Compilers". Ph.D. dissertation, University of Illinois at Urbana Champaign, 1971.

Leinius, R. P. "Error Detection and Recovery for Syntax Directed Compiler Systems". Ph.D. dissertation, University of Wisconsin, 1970.

Levy, J. P. "Automatic Correction of Syntax Errors in Programming Languages". Technical Report 71-116, Department of Computer Science, Cornell University, Ithaca, N.Y., 1971.

Lyon, G. "Least Errors Recognition of Mutated Context Free Sentences in Time $n^3 \log n$", Sixth Princeton Conference on Information Science and Systems (1972), 115-118.

--------. "Syntax Directed Least Error Analysis for Context Free Languages -- a Practical Approach", C.ACM, 17, 1 (1974), 3-14.

Martin, D. F. "Boolean Matrix Methods for the Detection of Simple Precedence Grammars", C.ACM, 11, 10 (1968), 685-687.

--------. "A Boolean Matrix Method for the Computation of Linear Precedence Functions", C.ACM, 15, 6 (1972), 448-454.

Morgan, H. L. "Spelling Corrections in Systems Programs", C.ACM, 13, 2 (1970), 90-94.

Moulton, P. G., and Muller, M. E. "DITRAN, a Compiler Emphasizing Diagnostics", C.ACM, 10, 1 (1967), 45-52.

McAfee, J. and Presser, L. "An Algorithm for the Design of Simple Precedence Grammars", J.ACM, 19, 3 (1972), 385-395.

McGruther, G.T. "An Approach to Automating Syntax Error Detection, Recovery and Correction for LR(k) Grammars". M.S. dissertation, Naval Postgraduate School, Monterey, California, 1972.

Peterson, T. G. "Syntax Error Detection, Correction and Recovery in Parsers". Ph.D. dissertation, Stevens Institute of Technology, Hoboken, N.J., 1972.

Rhodes, S. P. "Practical Syntactic Error Recovery for Programming Languages". Technical Report 15, University of California, Berkeley, 1973.

Schneider, V. B. "A Translator System for the Euler Programming Language". Technical Report 69-85, Maryland University Computer Science Centre, 1969.

Smith, W. B. "Error Detection in Formal Languages", Journal of Computer and Systems Science, 4 (1970), 385-405.

Souza, C. R., and Scholtz, R. A. "Syntactical Decoders, and Backtracking Grammars". The Aloha Systems Technical Report A69-9 AD701796, University of Honolulu, Hawaii, 1969.

Teitelbaum, R. "Context Free Error Analaysis by Evaluation of Algebraic Power Series". Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., 1973.

Walker, A., Redish, K. A., and Wood, D. "A Simple Binary Tree Display Algorithm". Computer Science Technical Report 73/10, McMaster University, 1973.

Wirth, N. "PL360, a Programming Language for the 360 Computers", J.ACM, 15, 1 (1968), 37-76.

--------. "The Programming Language PASCAL", Acta Informatica, 1 (1971), 35-63.

Wirth, N. and Weber, H. "Euler, a Generalisation of Algol and its Formal Definition, Part I", C.ACM, 9, 1 (1966), 13-25.

--------. "Euler, a Generalisation of Algol and its Formal Definition, Part II", C.ACM, 9, 2 (1966), 89-99.

Younger, D. H. "Recognition and Parsing of Context Free Languages in time n^3", Information and Control, 10, 2 (1967), 189-208.

Youngs, E. A. "Error Proneness in Programming". Ph.D. dissertation, University of North Carolina, 1970.

Zimmerman et al. "SLANG, a Language and Compiler". University of Wisconsin Computing Centre, Staff Document #4, 1967.